UNIVERSITY OF BERGAMO

School of Doctoral Studies

Doctoral Degree in Engineering and Applied Sciences

XXXII° Cycle

SSD: ING-INF/05

# Data-at-Rest Protection and Efficient Access Control in the Cloud

Advisor

Prof. Stefano Paraboschi

Doctoral Thesis

Marco ROSA

Student ID 1014425

Academic year 2018/2019

# Abstract

Cloud storage services offer a variety of benefits that make them extremely attractive for the management of large amounts of data. These services, however, raise some concerns related to the proper protection of data that, being stored on servers of third party cloud providers, are no longer under the data owner control. The research and development community has addressed these concerns by proposing solutions where encryption is adopted not only for protecting data but also for regulating accesses. Depending on the trust assumption on the cloud provider offering the storage service, encryption can be applied at the server side, client side, or through an hybrid approach. In this thesis, we introduce and implement a novel hybrid approach, named EncSwift. EncSwift relies on client side encryption for protecting data-at-rest, and on server-side encryption to enforce efficient access revocation.

An interesting evolution of these data protection solutions is represented by a different family of encryption techniques to be applied on the client side, i.e., the all-or-nothing transforms (AONTs). An AONT provides stronger security guarantees on the data it wraps, and can be exploited for enforcing efficient access revocation without requiring the support of the cloud provider. In this thesis, we introduce a novel AONT technique, i.e., Mix&Slice, and present an interesting application of AONTs to Decentralized Cloud Storage (DCS) networks. With regard to this, we present an approach enabling data owners to keep data confidentiality and availability under control, limiting the owners intervention with corrective actions when availability or confidentiality is at risk.

Finally, another contribution of this thesis consists in a proposal targeting efficient access control on data aggregations, when relying on a trusted provider. Indeed, despite the availability of information, situations like fragmented ownership and legal frameworks hinder data processing, requiring companies to design complex human-driven processes in order to gather, aggregate, and process data in a compliant way. Our proposal in this domain addresses this lack of automation with an access control mechanism extending the XACML policy language, and enforcing a novel decision process.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Document structure

This thesis is organized in three parts. Part I describes the problem of protecting data-at-rest while ensuring efficient access control on those data. The work presented in this part represents also one of the contributions of the University of Bergamo to the ESCUDO-CLOUD project (`www.escudocloud.eu`). The author's contributions to the ESCUDO-CLOUD project are manifolds. They mainly focus on the design and implementation of techniques aimed at protecting data stored on cloud providers.

- Chapter 1 introduces the overall scenario, and discusses on the trust assumptions, together with existing solutions and current limitations.

- Chapter 2 presents EncSwift, i.e., a solution for enforcing both protection on data-at-rest and efficient access revocation on resources stored on a cloud provider based on OpenStack Swift.

- Chapter 3 describes the implementation of EncSwift, and discusses on technical choices and challenges for its integration in Swift.

- Chapter 4 describes the integration of EncSwift with an industrial key manager provided by one of the partners of the ESCUDO-CLOUD project, British Telecom.

- Chapter 5 presents some research works related to EncSwift.

- Chapter 6 presents some concluding remarks, and introduces considerations that are developed in the following part of the document.

Part II focuses on a particular kind of data encryption techniques: all-or-nothing transform (AONT). With regard to AONT, many different algorithms have been proposed. This part of thesis presents one of them, proposed by the University of Bergamo in the context of the ESCUDO-CLOUD project, and discusses on an interesting application of data protection by means of AONT in decentralized cloud storage solutions.

- Chapter 1 presents another contribution of the University of Bergamo to the ESCUDO-CLOUD project, i.e., an AONT technique for data-at-rest protection: Mix&Slice.
- Chapter 2 considers data protection in a different setting, represented by a Decentralized Cloud Storage (DCS). In this chapter, AONT and erasure codes are used to provide data confidentiality and availability. The work presented in this chapter consists in one of the contributions of the University of Bergamo to the European project MOSAICrOWN (`www.mosaicrown.eu`).
- Chapter 3 describes some works related to AONT and its integration into DCS networks.
- Chapter 4 concludes this part of thesis.

Part III describes the research activity that the author did during his internship at SAP Labs France. This part of thesis deals with access control from a more classical perspective, i.e., at the decision level, and aims at integrating a classical approach to a modern context (e.g., machine learning and data analytics).

- Chapter 1 presents the core of the internship project, i.e., a novel evaluation strategy for requests of data aggregations, based on an extension of the XACML policy language and supplied with an engine.
- Chapter 2 compares our approach with the state of the art.
- Chapter 3 concludes this part of thesis, and presents interesting future works.

## 1.2 Publications

This section presents a list of papers (already published and/or currently under revision), that set the basis of this thesis.

Papers in Proceedings of International Conferences and Workshops:

- Enrico Bacis, Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Marco Rosa, and Pierangela Samarati. "Access Control Management for Secure Cloud Storage" in 12th EAI International Conference on Security and Privacy in Communication Networks (SecureComm 2016) [Part I, Chapter 2 and Chapter 3]

- Enrico Bacis, Sabrina De Capitani di Vimercati, Sara Foresti, Daniele Guttadoro, Stefano Paraboschi, Marco Rosa, Pierangela Samarati, and Alessandro Saullo. "Managing Data Sharing in OpenStack Swift with Over-Encryption" in 3rd ACM Workshop on Information Sharing and Collaborative Security (WISCS 2016) [Part I, Chapter 3]

- Enrico Bacis, Marco Rosa, and Ali Sajjad. "EncSwift and Key Management: An Integrated Approach in an Industrial Setting" in 3rd Workshop on Security and Privacy in the Cloud (SPC 2017) [Part I, Chapter 4]

- Enrico Bacis, Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Marco Rosa, and Pierangela Samarati. "Mix&Slice: Efficient Access Revocation in the Cloud" in 23rd ACM Conference on Computer and Communication Security (CCS 2016) [Part II, Chapter 1]

- Enrico Bacis, Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Marco Rosa, and Pierangela Samarati. "Dynamic Allocation for Resource Protection in Decentralized Cloud Storage" in 2019 IEEE Global Communications Conference (GLOBECOM 2019) [Part II, Chapter 2]

Chapters in books:

- Enrico Bacis, Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Marco Rosa, and Pierangela Samarati. "Protecting Resources and Regulating Access in Cloud-based Object Storage". From Database to Cyber Security: Essays Dedicated to Sushil Jajodia on the Occasion of his 70th Birthday [Part I, Chapter 1 and Chapter 2]

Under revision at the time of writing:

- Marco Rosa, Francesco Di Cerbo, and Rocìo Cabrera Lozoya. "Declarative Access Control for Aggregations of Multiple Ownership Data" [Part III, Chapter 1]

## 1.3 Acknowledgments

The research leading to the results documented in this thesis was supervised by Prof. Stefano Paraboschi (Università degli Studi di Bergamo) and has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644579 (EscudoCloud) and No 825333 (Mosaicrown).

# Part I

# Protecting Resources and Regulating Access in Cloud-based Object Storage

# Chapter 1

# Data Protection in Different Trust Scenarios

## 1.1 Introduction

The ever increasing availability of off-the-shelf cloud storage platforms has contributed to the growing of the *Storage-as-a-Service* (*SaaS*) market, with an increasing trend for users and companies to offload their (possibly sensitive or confidential) data and resources. There are several reasons for using cloud storage services such as the benefits in terms of availability, scalability, performance, and costs as well as the ability to easily share data with other users. However, this trend also introduces several security and privacy risks that can slow down the widespread adoption of storage services (e.g., [32, 50, 73]). In fact, by relying on third parties for the storage of their data and resources, users and companies lose their control over them: how can users and companies trust that their data are properly protected when stored on a third-party server? The research and development communities have dedicated many efforts in designing solutions for addressing this concern (e.g., [32]). Encryption is at the basis of many of these techniques: when data are encrypted they are visible only to the users who know the encryption key. Encryption has then been adopted not only as a valid solution for protecting data confidentiality (even against adversaries with access to the physical representation of the data, including the cloud providers themselves), but also for supporting selective sharing of such data [31]. In this case, the idea consists in encrypting different portions of the data with different keys and then sharing the encryption keys only with the users that have the authorization for accessing the corresponding encrypted data. Figure 1.1 illustrates the typical reference scenario when considering cloud storage infrastructures.

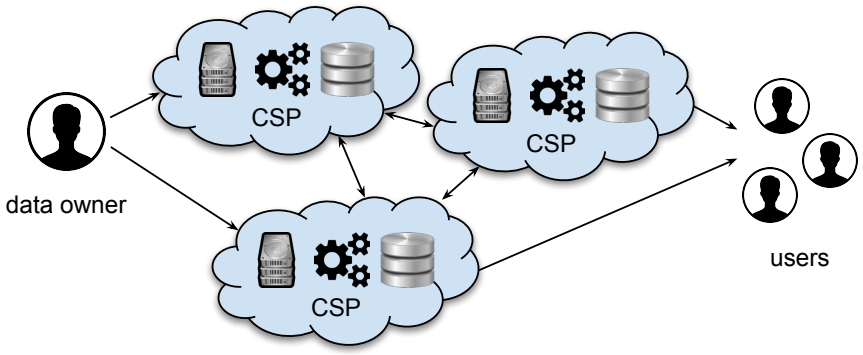



Figure 1.1: Reference scenario

As it is visible from the figure, there are three main entities involved in this scenario: the *data owner* who wishes to outsource the management of her data to a third party, the *cloud providers* (CSPs) offering storage services, and other *users* who may need to access the data stored on cloud providers.

A fundamental aspect that needs to be considered when applying encryption to protect data is the *trust assumption* on the cloud providers in charge of storing and managing the data. Cloud providers can be *trusted*, *honest-but-curious*, or *lazy/malicious*. A trusted provider is fully trusted to access and manage the data that it stores. A honest-but-curious provider is trustworthy for providing services but should not be allowed to know the actual data content. A lazy or malicious provider is neither trusted nor trustworthy and therefore its behavior should be controlled. Depending on the trust assumption, encryption can be applied following three different strategies: *server-side*, *client-side*, *hybrid*. Server-side encryption means that the encryption of the data is managed directly by the cloud provider, which stores and manages also the encryption keys. In this case, the cloud provider guarantees that the data are stored in an encrypted format. However, whenever the cloud provider's services require direct visibility of the plaintext data for access execution, the provider can decrypt the data. Since the cloud provider has full visibility on the data, it can also enforce access restrictions. Server-side encryption can be applied only when the cloud provider is fully trusted. Client-side encryption means that users encrypt their data before storing them on external cloud providers. In this case, the encryption keys are stored and managed by the owner of the data and cloud providers cannot access the data in plaintext form, which limits the functionality that they can offer. Also, access control restrictions need to be enforced by the data owner who has to mediate all access requests to the data. This clearly

| Encryption type / Trust Assumption | *server-side* | *client-side* | *hybrid* |
|---|---|---|---|
| *trusted* | ✓ | ✓ | ✓ |
| *honest-but-curious* | × | ✓ | ✓ |
| *lazy/malicious* | × | ✓ | × |

✓: applicable; ×: not applicable

Figure 1.2: Encryption scenario depending on the trust assumption on cloud providers

reduces the advantages of outsourcing the management of data to a third party. Client-side encryption can be applied under any trust assumption on the cloud provider. However, it is usually adopted when the cloud providers are honest-but-curious or lazy/malicious. In the hybrid approach, the encryption of the data is performed both at the client-side and at the server-side with the consequence that there are two sets of encryption keys: one managed by the data owner and another one managed by the cloud provider. The rationale behind the hybrid scenario is that client-side encryption protects the data from cloud providers while server-side encryption efficiently enforces changes in the access control policy without the involvement of the data owner. Clearly, this approach can be applied only when cloud providers are honest-but-curious (or trusted) but cannot be applied when the cloud provider is lazy/malicious since there is no guarantee that the provider applies the required encryption operations. Figure 1.2 summarizes the applicability of the three encryption strategies according to the trust assumptions that characterize the cloud providers.

The goal of the first part of this thesis is to provide an overview of the current encryption-based solutions for protecting and enforcing selective access over data stored in the cloud. In particular, for each of the three encryption strategies discussed above, we first describe its salient aspects along with the main advantages and disadvantages. We then describe a representative system that applies the considered strategy.

## 1.2 Server-side Encryption

With server-side encryption, the cloud provider protects data in storage with an encryption layer that it can remove when needed to perform access and query execution (i.e., the cloud provider manages both the data and the encryption keys). In this case, users placing data in the cloud have complete

trust that the cloud provider will correctly manage the outsourced information.

### 1.2.1 Discussion

Being fully trusted, the management of data is completely delegated to the cloud provider itself. From the point of view of the users, the main advantage of this solution is that they can use all the functionality offered by the server for querying the outsourced data. Furthermore, the data owner can delegate the cloud provider to enforce access control policies for regulating access to data. From the point of view of the cloud provider, server-side encryption allows it to use *deduplication techniques* to avoid the storage of multiple copies of the same data, thus saving storage space. Basically, a cloud provider keeps the hash of every resource it is storing. When a user uploads a resource, the cloud provider computes the hash of the resources and checks whether the computed hash corresponds to the hash of a resource it already stores. If this is the case, the cloud provider discards the storage request and provides a link to the resource already stored.

Although many of the most well-know public cloud storage providers use server-side encryption (e.g., Dropbox, Amazon, and Google), this solution is not always feasible and introduces security risks. In fact, since the encryption keys are stored with the data, an adversary can exploit possible vulnerabilities of the cloud provider to obtain both the encrypted data and the encryption keys, thus obtaining the access to the plaintext version of the data themselves. Furthermore, the cloud provider might be forced by authorities to provide the stored data in their plaintext form. With respect to the data deduplication techniques commonly adopted by cloud providers, they can be exploited for violating data confidentiality. As an example, suppose that an adversary knows that a certain resource is stored on the cloud provider but does not know the value of some specific bytes (e.g., one value of a csv file). The adversary might try to generate as many resources as the possible combinations for the missing bytes and to upload each of them, one at a time. When the upload operation is not performed, the adversary knows that the uploaded file corresponds to the one already stored and therefore knows the value of the missing bytes. We note that these considerations apply to both public clouds and private clouds (i.e., cloud solutions built internally by a company).

Examples of public storage services based on server-side encryption are *Dropbox* [35], *Amazon Simple Storage Service (S3)*, and *Google Cloud Storage (GCS)*. All these services typically store the encryption keys in their proprietary key management system and mainly differ in the pricing schema.

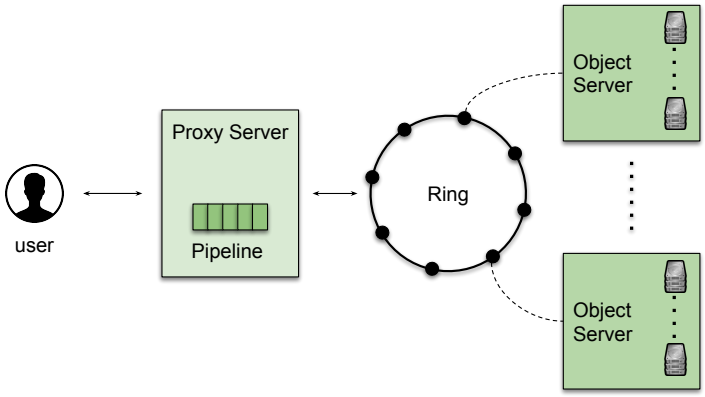



Figure 1.3: OpenStack Swift architecture

Although the companies ensures that no access is performed on users' data, they could potentially access all the data they store. In the following, we present OpenStack Swift as an example of cloud solution offering server-side encryption.

### 1.2.2 Case Study: OpenStack Swift IBM Key Rotation

A well-known open source cloud computing platform that adopts server-side encryption is OpenStack (`http://www.openstack.org`). OpenStack manages large pools of computing, storage, and networking resources, all controlled by administrators through a dashboard. OpenStack consists of several components including an object storage system, called *Swift*. The architecture of Swift is composed of a *Proxy Server*, a *Ring*, and an *Object Server* (Figure 1.3). The Proxy Server is the key component of Swift and is responsible for processing requests coming from users and interacts with all other components. The Ring determines the physical device where a file should be located. In other words, it is responsible for mapping names and physical location of data. The Object Server is a blob storage (i.e., a storage that can manipulate unstructured data) in charge of storing, retrieving, and deleting objects on disks. Each object is stored as a binary file, and its metadata are stored as extended attributes of the file. Objects stored in Swift are organized in *containers*, which loosely corresponds to directories of common file systems. Containers are organized in *tenants* (or accounts). For interacting with Swift, a user sends a valid request to the Proxy Server. The request is then processed by a pipeline of middlewares, and each of them can enrich, filter, or drop metadata. In case the request reaches the end of the pipeline,

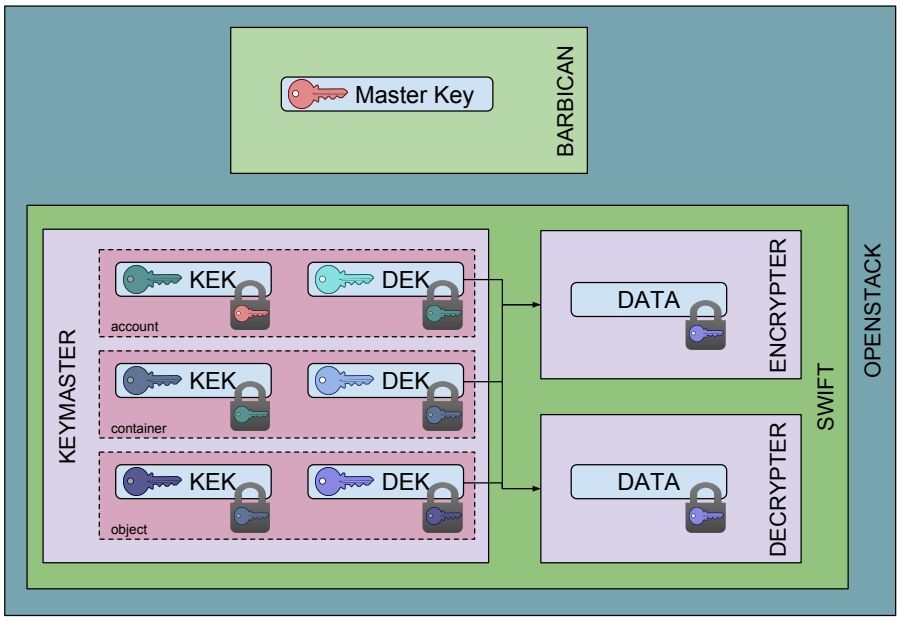



Figure 1.4: Swift-KeyRotate: Key organization

it is dispatched to the relevant Object Server based on the information contained in the Ring. Once the request is processed by the Object Server, a response is sent to the user, processed again by the middlewares of the Proxy Server but in reverse order.

OpenStack Swift has supported server-side encryption to protect data at-rest (both objects content and metadata) since the Ocata[1] release. To this purpose, three new middlewares have been added: *encrypter*, *decrypter*, and *keymaster*. Encrypter and decrypter are middlewares in charge of performing encryption and decryption operations on data and metadata. Keymaster is responsible for deciding whether a resource should (or should not) be encrypted and which encryption key should be used[2]. Swift supports a variety of keymaster implementations, including *Swift-KeyRotate*[3] proposed by IBM. The Swift-KeyRotate is a hierarchical key management system that manages three types of keys: a top-level *Master Key*; *Data Encryption Keys* (DEKs), used to decrypt and encrypt user/system metadata and user data; and *Key Encryption Keys* (KEKs), used internally in the keymaster middleware to protect other KEKs and DEKs. As data are hierarchically organized in accounts, containers, and objects, also KEKs and DEKs are hierarchically

[1] https://github.com/openstack/swift/blob/master/CHANGELOG
[2] http://specs.openstack.org/openstack/swift-specs/specs/in_progress/at_rest_encryption.html
[3] https://github.com/ibm-research/swift-keyrotate

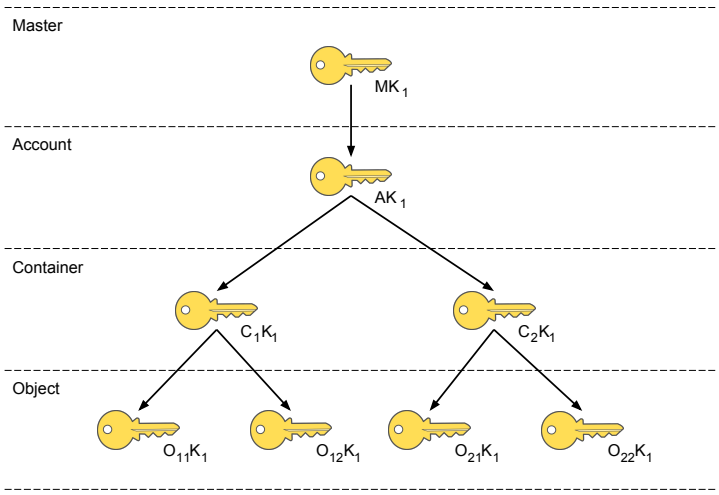



Figure 1.5: Swift-KeyRotate: An example of KEK hierarchy with two containers and four objects

organized according to the account/container/object hierarchy (Figure 1.4). More precisely, a KEK and a DEK are generated for each account, container, and object. DEKs associated with accounts and containers are used to encrypt the metadata of the accounts and containers, respectively. DEKs associated with objects are used to encrypt both objects and their metadata. The Master Key (which is stored in the Barbican system, the secret storage of OpenStack) is used to encrypt the KEK associated with an account. Then, the KEK associated with an entity (i.e., an account, a container, or an object) is used to encrypt the DEK associated with the same entity and the KEKs associated with the entities of the level below (if any). Figure 1.4 illustrates the hierarchical organization of KEKs and DEKs. When a user authenticates to OpenStack via Keystone (the identity server of OpenStack), the user is associated with an account and therefore she can access a Master Key that is retrieved from Barbican through the user's authentication token.

Good key management practice requires a periodic key rotation, meaning that encryption keys must be periodically changed. The rotation of the Master Key stored in Barbican is similar to the approach adopted by systems for industrial key-lifecycle management [16, 37]. However, in Swift-KeyRotate, it is not sufficient to rotate the Master Key since an adversary could have stored the key of a lower level and then could be still able to obtain access to all the underlying data. Key rotation is then performed on all levels and is also needed to securely delete objects. We note that key rotation involves only the KEKs while the DEKs are generated when the corresponding entity is created and are never changed. As an example, consider two containers,

$C_1$ and $C_2$, each of which includes two objects, $\{o_{11}, o_{12}\}$ and $\{o_{21}, o_{22}\}$, respectively. Figure 1.5 illustrates the corresponding KEK hierarchy: nodes of the hierarchy represent keys and an arc from a key $k$ to key $k'$ means that $k'$ is encrypted using $k$ (e.g., in the figure an arc from $\mathrm{MK}_1$ to $\mathrm{AK}_1$ means that the account KEK is encrypted via the Master Key). Suppose that a user wishes to delete object $o_{11}$. In this case, new KEKs have to be generated for all entities in the key hierarchy that are on the path to object $o_{11}$ (i.e., container $C_1$, account $A$, and the master). Furthermore, the KEKs of all entities whose parent KEKs have been changed are re-encrypted with the new parent key. In our example, the KEK $O_{12}K_1$ of object $o_{12}$ is encrypted with the new KEK associated with container $C_1$, say $C_1K_2$, the KEK $C_2K_1$ of container $C_2$ is encrypted with the new KEK of account key $A$, say $AK_2$, and the account key $AK_2$ is encrypted with the new Master Key, say $MK_2$.

## 1.3 Client-side Encryption

With client-side encryption, the data owner encrypts her data before outsourcing them to a cloud provider. The encryption keys are therefore stored at the client-side and are never exposed to the cloud provider, which cannot decrypt the outsourced data. This solution is typically applied when the cloud provider is honest-but-curious or lazy/malicious.

### 1.3.1 Discussion

Like for the server-side encryption, this solution has some advantages and disadvantages for users and the cloud provider. From the point of view of the users, the main advantage is an increase of the spectrum of cloud providers to which a data owner can outsource her data. In fact, since the data are encrypted at the client-side, the data owner can also leverage the services of less reputable cloud providers, which are typically cheaper than well-known cloud providers. The main disadvantages are that the data owner has to directly manage the encryption keys and has to enforce access control restrictions as well as changes in the access control policy. In this scenario, access control can be enforced using an approach based on *selective encryption* [31]. Intuitively, selective encryption means that the data owner encrypts different portions of her data using different keys and discloses to each user only the encryption keys used to protect the data they can access. Whenever the access control policy changes, the data owner must download the involved data, decrypt and re-encrypt them with a new encryption key, re-upload the new encrypted data, and share the new encryption key with authorized

users. Clearly, such an approach puts much of the work at the data owner side, introducing a bottleneck for computation and communication. Another disadvantage is that both the client and the server storing the data may be the subject to attacks from an adversary. Common client-side attacks include, for example, the *man-in-the-browser* attack, in which an adversary takes control over a part of the browser (e.g., browser extension hijacking) to replace the cryptography algorithms used by the cloud provider with algorithms controlled by the adversary. This attack can also compromise the key-generation and the client-side integrity checks without the client being aware of it. The adversary might also try to compromise the server to use it as a vehicle to send malicious code to the client. For services that provide access via browser, in fact, the server still plays a central role by providing the JavaScript code that encrypts the data before upload. If an adversary is able to replace this code with a malicious one, the adversary can compromise the confidentiality of the outsourced data collection.

From the point of view of the cloud provider, the main advantage is that the cloud provider should not be worried about the protection of data, which is guaranteed by client-side encryption. The main disadvantage is that deduplication techniques cannot be used since the same plaintext data are encrypted by different data owners using different keys, thus generating different ciphertexts. A possible approach for addressing this issue consists in using *convergent encryption,* a cryptosystem that can generate identical ciphertexts from identical plaintext data. While interesting, this techniques is still vulnerable to the brute force attack described in Section 1.2.1.

Examples of cloud storage services supporting client-side encryption are SpiderOak and MEGA [25]. In the following, we describe the MEGA system.

### 1.3.2 Case Study: Mega

MEGA system supports browser-based User Controlled Encryption (UCE), meaning that resources are automatically encrypted on the user's device before they are stored on MEGA cloud service [46]. Client-side encryption uses different encryption keys managed by the data owner: a *Master Key* is a user's key used to protect the symmetric *file key* adopted for encrypting a file that is stored on MEGA; a *user password* is then used to encrypt the Master Key. File keys encrypted with the Master Key as well as the Master Key encrypted with the user password are stored on MEGA. Different files are encrypted with different file keys and therefore the knowledge of a file key allows a user to decrypt only the file encrypted with such a key. This mechanism enforces selective encryption, as illustrated in the previous section. Note that an adversary compromising a storage server of MEGA

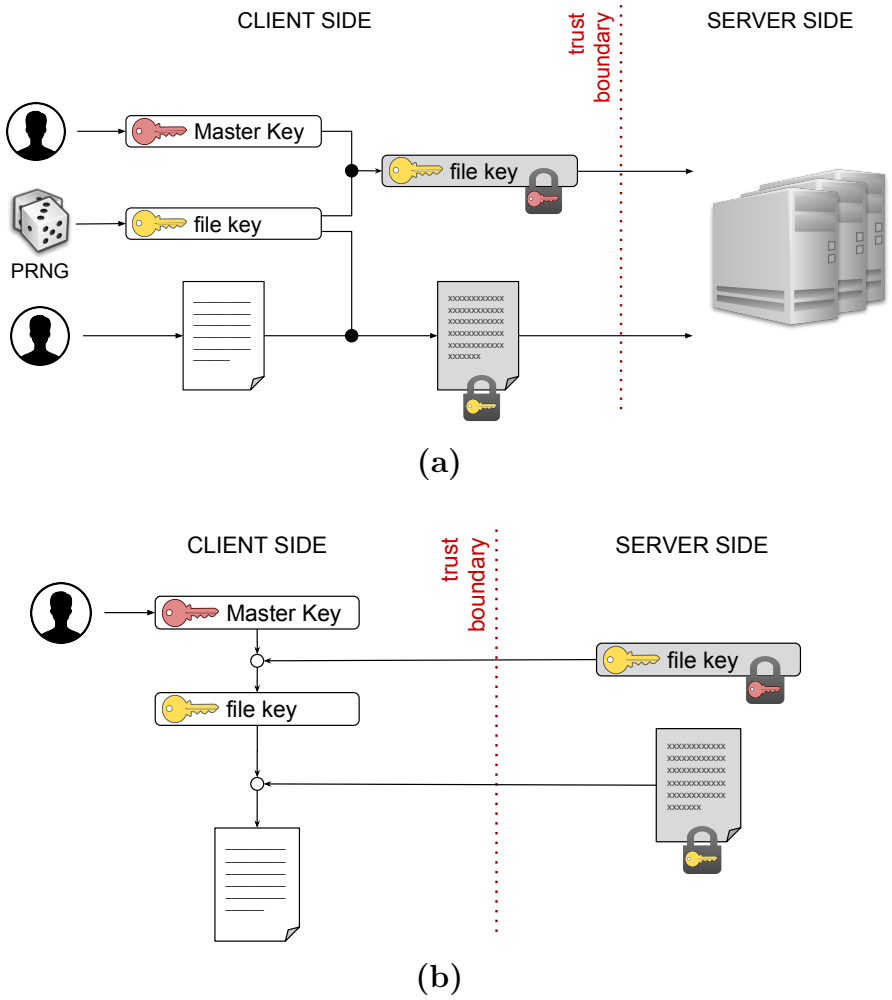



Figure 1.6: MEGA upload (a) and download (b) process

cannot decrypt the encrypted files stored on the node since the encryption key is managed at the client-side. Furthermore, MEGA uses HMAC to provide integrity guarantee to the file stored in MEGA store node. In this way, an adversary with access to a MEGA store node and the file key of a file cannot replace the file without the original user who has uploaded the file noticing that it has been changed.

Figure 1.6 illustrates the MEGA encryption and decryption processes. When a user wishes to store in the MEGA system a resource, a new file key is generated with the support of a cryptographically strong random number with entropy coming from both HTML5 APIs and mouse/keyboard entropy pool. The file is then encrypted with the file key and AES-128 and the result-

ing ciphertext is uploaded on MEGA. The encryption operation is performed either by the MEGA client or directly in the browser, using JavaScript. The file key is encrypted with the Master Key that in turn is protected with the user password. The resulting encrypted keys are then uploaded on MEGA (Figure 1.6(a)). When a user wishes to access a given file, she first provides her password, which is used to decrypt the Master Key. The file key of the file of interest is then decrypted, using the Master Key, and it is used to decrypt the file. Post-download integrity checks are performed via a chunked variant of the Counter with CBC-MAC (CCM) mode, which is an encryption mode only defined for block ciphers with a block length of 128 bits. Note that MEGA supports *end-to-end* encryption, meaning that encryption and decryption operations are performed at the client side.

With respect to the ability of supporting deduplication, MEGA can apply a deduplication process only when a user copies/pastes a file within her cloud drive or when the file is shared with another user who imports it. In fact, even if two (or more) users upload the same encrypted file, it will appear different since the file is encrypted using different keys.

Resource sharing is supported using two different strategies. The first strategy consists in sharing a public link that will allow a user receiving it to decrypt the corresponding resource, as the file key used to encrypt the file is included in the link (it is important to note that the link is generated at the client side and not at the server side). With this strategy, the public link can be shared with anyone who may not necessarily have a MEGA account. The second strategy is only applicable between MEGA users and is based on asymmetric encryption (RSA-2048). Each user is associated with a public key and a private key both stored on MEGA: the public key is stored in plaintext and the private key is stored in encrypted form, using the Master Key of the user as encryption key. When a user, say $A$, wishes to share a resource with another user, say $B$, $A$ encrypts the corresponding file key with the public key of $B$ and the resulting ciphertext is stored on MEGA. When $B$ wishes to access the resource, she first retrieves from MEGA her encrypted private key, decrypts it using her Master Key and the resulting plaintext private key is used to decrypt the file key that $B$ can use for decrypting the file of interest. To provide access revocation to users who were previously given access to the file key, MEGA applies a classical access control policy defined by the data owner. A revoked user is therefore prevented access to the encrypted files. Note that MEGA is trusted to correctly enforce the access control policy defined by the data owner.

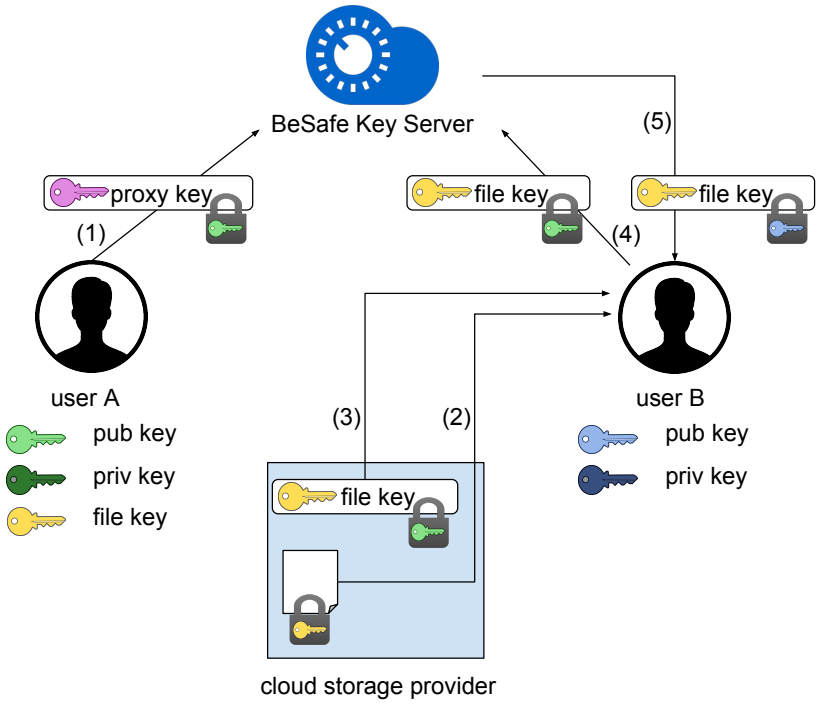



Figure 1.7: BeSafe proxy re-encryption architecture

## 1.4 Hybrid encryption

The hybrid approach combines client-side encryption with server-side encryption to improve efficiency in data management. Hybrid approaches are usually based on different layers of encryption with some encryption keys managed at the client side and other encryption keys managed at the server side. The latter keys are needed by the cloud provider to correctly enforce changes in the access control policy.

### 1.4.1 Discussion

The main advantage of the hybrid approach is the efficient enforcement of changes in the access control policy without impacting the confidentiality of the resources. In fact, while with client-side encryption changes in the access control policy must be enforced by the data owner (Section 1.3), with a hybrid approach such changes can be enforced directly by the cloud provider. This approach can therefore be applied only when the cloud provider is honest-but-curious, since the provider has to correctly enforce the changes as dictated by the data owner. An example of commercial solution adopting this approach

is BeSafe SkyCryptor[4], a commercial platform providing end-to-end encryption. BeSafe is based on a honest-but-curious proxy (BeSafe Key Server) performing a proxy re-encryption [7, 17] on encryption keys, and on a public cloud storage provider storing the encrypted data. Proxy re-encryption is a cryptographic technique that transforms a ciphertext generated with a key $k$ into a ciphertext that can be decrypted using a different key $k'$, without the need for decryption the original ciphertext. Hence, it can be performed also by a party not trusted for the plaintext content of the data. Each user of the BeSafe SkyCryptor has a pair of public and private keys. Whenever a user wants to store a resource at the public cloud provider, the resource is first encrypted at the client side using a symmetric encryption key, called *file key*. The encrypted resource and the file key, encrypted with the public key of the user, are then stored on the cloud provider. Resources can be shared only among users with a BeSafe account. Figure 1.7 shows an example of sharing between user $A$ and user $B$. User $A$ first generates a new *proxy key*, encrypts such a key with her public key, and sends the resulting ciphertext to the BeSafe Key Server (1). $B$ downloads the encrypted resource (2) from the public cloud storage provider, along with the corresponding encrypted file key (3). The encrypted file key is then sent to the BeSafe Key Server (4) that proxy-re-encrypts it using the proxy key generated by $A$. The result of the proxy re-encryption is sent to $B$ (5) who can decrypt it through her private key for retrieving the file key and then can use the retrieved file key to decrypt the resource [52].

Differently from the previous sections, we do not present a case study here: due to its complexity and size, we leave the dissertation of the hybrid encryption case study, i.e., EncSwift [10, 11], to the next chapters.

[4]https://besafe.io/

# Chapter 2

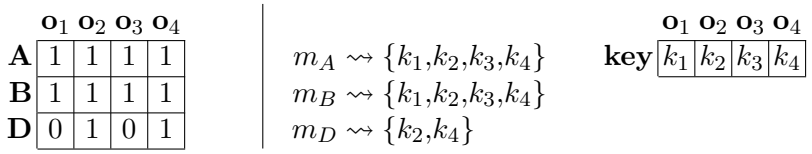
# EncSwift

EncSwift is a solution for providing data-at-rest encryption and enforcing access control when relying on an honest-but-curious cloud provider. This tool is based on OpenStack Swift where, as already discussed in Section 1.2.2, data are hierarchically organized in accounts, containers, and objects. The access control enforcement mechanism implemented by EncSwift is based on selective encryption (introduced in Section 1.3.1) and over-encryption approaches [30, 26].

## 2.1 Basic Concepts

We consider a scenario where users wish to outsource data to an external cloud service provider (CSP) and selectively share their data with others. Different data (owned by the same user) may be accessible by different sets of users. Every data owner has an access control policy specifying authorizations on her data.

As mentioned before, we assume that the CSP is based on the OpenStack framework, which includes the Swift module as an object storage service. Swift enforces discretionary access control restrictions over the objects it stores by associating a read access control list and a write access control list with each container and tenant in the system. These access control lists identify the users who can read and write the container/tenant. To enforce access control restrictions, Swift relies on *Keystone* for users authentication. As already mentioned, Keystone is an OpenStack component acting as identity server, which provides a central directory of users mapped to the OpenStack services they can access. Besides, *Barbican* offers a RESTful API designed for the secure storage, provisioning, and management of secrets such as passwords, encryption keys, and X.509 certificates.




Authorization policy $\mathcal{A}_{\mathbf{A}}$ (a)

Policy-based encryption $\mathcal{E}_{\mathbf{A}}$ (b)

Figure 2.1: An example of authorization policy defined by user Alice (a) and corresponding policy-based encryption (b)

We recall that we assume the cloud service provider to be *honest-but-curious*, that is, trusted to correctly manage the data (i.e., trustworthy) but not trusted for accessing the content of objects. Consistently with our focus on data confidentiality, in this chapter we are concerned with the representation and enforcement of an access control policy regulating read access to objects. We note however that our approach can be extended to the consideration of write authorizations [27]. In the following, $acl(o)$ denotes the read access control list of object $o$ and $\mathcal{A}_u$ is the set of read access control lists defined by user $u$ for her objects. Figure 2.1(a) illustrates an example of authorization policy defined by user Alice. In this example, we assume that there are three users, Alice (A), Bob (B), and Dave (D), and four objects ($o_1$, $o_2$, $o_3$, and $o_4$) owned by Alice. In the matrix in Figure 2.1(a), entry $[u,o]$ has value 1 if $u$ is authorized to read $o$ (i.e., $u \in acl(o_i)$) and 0 if $u$ is not authorized to read $o$ (i.e., $u \notin acl(o_i)$).

The work presented in this chapter is based on the policy-based encryption and over-encryption approach proposed in [29, 30], and aims at their representation and enforcement with Swift, which also require some re-definition and adjustment of these concepts. Essentially, each user is associated with a symmetric key, and each object is encrypted using a symmetric key that depends on the access control policy. Keys are organized in such a way that a user $u$ can derive (via public tokens), all and only the keys of the objects $o_i$ she is authorized to access (i.e., $u \in acl(o_i)$). Yet, policy updates, i.e., changes in the acl connected to a resource, would require to change the key and re-encrypt objects affected by revocation. This would entail download and re-upload operations by owners, which could become cumbersome and affect the performance of the system. In order to avoid such a situation, policy updates are enforced by super-imposing a second layer of encryption on the encrypted object itself. Hence, every object can then have a first layer of encryption (*BEL, Base Encryption Layer*) imposed by the data owner for

protecting the confidentiality of the data from unauthorized users as well as from the CSP, and a second layer of encryption (*SEL, Surface Encryption Layer*) applied by the CSP for protecting the object from users who are not be authorized to access the object but who might know the underlying BEL key. A user will be able to access an object only if she knows both the SEL key and the BEL key with which the object is encrypted. In the following, we use notation $\mathcal{E}_u$ to denote the policy-based encryption equivalent to the authorization policy $\mathcal{A}_u$ defined by user $u$. Figure 2.1(b) illustrates the policy-based encryption equivalent to the authorization policy in Figure 2.1(a). In this figure, keys $m_A, m_B, m_D$ are the symmetric keys of the users and keys $k_1, k_2, k_3, k_4$ are the symmetric keys used to encrypt the objects. Notation $m_x \rightsquigarrow k_y$ represents the fact that key $k_y$ is derivable from key $m_x$. In the remaining sections, we first describe how a policy-based encryption can be realized in Swift (Section 2.2), and then illustrate how to enforce policy updates (Section 2.3).

## 2.2 Access Control Enforcement in Swift

Our approach translates the authorization policy defined by a user into a policy-based encryption that relies on the use of different keys and ad-hoc structures supporting the client-based Swift encryption. In this section, we describe such keys and ad-hoc structures (which are stored as traditional Swift objects), and then illustrate how policy-based encryption can be implemented.

### 2.2.1 Keys and User-Based Repositories

Our approach is based on the definition and management of different keys. There are (symmetric) keys associated with objects for objects' encryption (enforcing the self-protection mentioned in the introduction). Also, each user is associated with a (symmetric) key as well as with two pairs of asymmetric keys to support identity management and signature, respectively. Finally, authorizations are realized by encrypting object keys with user keys. This allows users to retrieve the key of objects they are authorized to access, providing the same functionality that public-tokens provided in [29, 30].

We describe the different keys and their characteristics and functionality in the following.

**Data Encryption Key (DEK) $\mathbf{k_i}$.** Every object $o_i$ is protected by symmetric encryption using a DEK $k_i$. Each DEK $k_i$ has a given size, is asso-

ciated with an encryption algorithm, and has an identifier, denoted $id(k_i)$, that identifies the key among all the keys used in the system.

**Master Encryption Key (MEK) $\mathbf{m_u}$.** Every user $u$ has a personal symmetric master encryption key $m_u$. The knowledge of this key permits to access, directly or indirectly, all the objects that user $u$ is authorized to see. Given the user identity loss that would derive from a compromise of the MEK, it is assumed that the user keeps the MEK only on the client-side, never exposing it to the server or to other users.

**User encryption key pair $\langle \mathbf{p_u}, \mathbf{s_u} \rangle$.** Each user $u$ is associated with an asymmetric key pair $\langle p_u, s_u \rangle$ for encryption (our implementation adopts RSA). As we show later on, the availability of asymmetric cryptography supports the realization of a cooperative cloud storage service, where each user may make her objects available to other users. Note that in most application domains, the correspondence between a user identity and a public key is supported by certificates issued by a trusted Certification Authority. Swift can instead benefit from the availability of Keystone, which already centralizes the management of user identities, and the public key is assumed to be available in the user profile managed by Keystone.

**User signing key pair $\langle \mathbf{sp_u}, \mathbf{ss_u} \rangle$.** Each user $u$ is associated with an asymmetric key pair $\langle sp_u, ss_u \rangle$ for signing messages (our implementation adopts EC-DSA). The reason for having a signing key pair is that it is common in security systems to separate the encrypting and signing identities. This improves security and flexibility, giving the option to use a dedicated cryptographic technique for each function. Signatures are used to guarantee the integrity of objects and of the information that users adopt for deriving the DEKs. Like for asymmetric encryption, the public key for signatures is also stored in the Keystone profile of users.

**Key Encryption Key (KEK).** A KEK is at the basis of the mechanism that translates the access control policy defined by a user into an equivalent policy-based encryption. DEKs are encrypted and stored in the form of Key Encryption Keys (KEKs), which should not be confused with the KEK used in the Swift-KeyRotate approach illustrated in the previous chapter (Section 1.2.2). For each container that a user is authorized to access, there is therefore a KEK that the user can decrypt to obtain the DEK used for encrypting the objects in the container. As we will see in the following sub-section, there are two variants of KEKs, depending on the cryptographic

technique used to protect them: symmetric KEKs, encrypted with the MEKs of users, and asymmetric KEKs, encrypted with the public keys of users. The KEKs that allow a user $u$ to derive the keys of the objects she is authorized to access are stored in a user-based repository, denoted $\mathcal{R}_u$. Each KEK is characterized by the following information: a KEK identifier, the identifier of the protection key, the identifier of the encrypted key, a timestamp, the identifier of the creator (only for asymmetric KEKs), an authentication code, and the encrypted key. The authentication code is used to verify the integrity of a KEK and is generated with the symmetric key of the user who creates the KEK (in case of symmetric KEK) or with the private signing key of the creator (in case of asymmetric encryption). Functions are available that allow the user to extract from her repository the KEK associated with a given protected key identifier.

The identifier of the DEK used to protect an object is maintained in the descriptor of the object itself. Such a piece of information is needed, whenever a user accesses an object, to retrieve the right KEK that allows the user to derive the corresponding DEK. Analogously, the descriptor of a container includes the identifier of the key to be used to encrypt the objects that will be inserted in the container. At initialization time, the key identifier in the descriptor of the objects stored in a container coincides with the key identifier in the container descriptor. As we will discuss in Section 2.3, due to policy changes, the key associated with a container may change and objects in the container may still be protected with a previous container key.

### 2.2.2 Policy-Based Encryption

All users in the system can define an access control policy for the objects they own. We now describe how the authorization policy $\mathcal{A}_u$ defined by user $u$ is translated into an equivalent policy-based encryption $\mathcal{E}_u$ using the keys illustrated in the previous section.

User $u$ creates as many containers $C_1, \ldots, C_m$ as needed and, for each of them, creates a DEK $k_i$, $i = 1, \ldots, m$, using a robust source of entropy. Consistently with Swift working, we assume that all objects in a container have the same acl. User $u$ then encrypts all objects in a container $C_i$ with the DEK $k_i$ of the container and stores them in $C_i$, which will have therefore the same acl for all the objects in it. Each DEK $k_i$ is encrypted with the MEK $m_u$ of the user who created the container and the resulting KEK is stored in the user's repository $\mathcal{R}_u$. For each user $u_j$ in the acl corresponding to container $C_i$, user $u$ encrypts DEK $k_i$ with $u_j$'s public key $p_{u_j}$ and signs

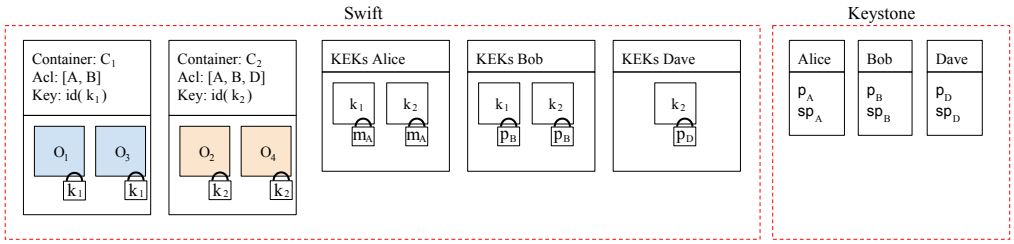



Figure 2.2: Policy-based encryption $\mathcal{E}_A$ equivalent to the authorization policy $\mathcal{A}_A$ in Figure 2.1(a)

it using $ss_u$, thus producing an asymmetric KEK usable by $u_j$. This KEK is stored in $u_j$'s repository $\mathcal{R}_{u_j}$.

**Example 1.** *Consider the authorization policy of Alice in Figure 2.1(a). Figure 2.2 shows how this policy is translated into an equivalent policy-based encryption. Alice creates two containers $C_1$ and $C_2$ and stores objects $o_1$ and $o_3$ both encrypted with key $k_1$ in $C_1$, objects $o_2$ and $o_4$ both encrypted with $k_2$ in $C_2$. She then creates her KEKs as well as the KEKs that Bob and Dave can use to access the objects for which they are authorized. In particular, Alice encrypts DEKs $k_1$ and $k_2$ with her MEK $m_A$ and stores the resulting KEKs in her repository $\mathcal{R}_A$. Then, she encrypts DEK $k_1$ with Bob's public key $p_B$ and DEK $k_2$ with public keys $p_B$ and $p_D$ of Bob and Dave, respectively. The resulting KEKs are stored in repositories $\mathcal{R}_B$ and $\mathcal{R}_D$, respectively. The figure also illustrates the profiles of Alice, Bob, and Dave managed by Keystone. These profiles contain the public keys of the users.*

When a user $u_j$ wishes to access an object $o_l$, the object descriptor is first accessed to retrieve the identifier of the DEK used to encrypt $o_l$. This identifier is then used to retrieve the corresponding KEK from repository $\mathcal{R}_{u_j}$ and then derive the DEK $k_l$. Derivation will require user $u_j$ either to use her own MEK $m_{u_j}$ (for symmetric KEK), or to apply the private encryption key $s_{u_j}$ (for asymmetric KEK). To improve the efficiency of the subsequent accesses to the key and simplify the procedure, once a DEK provided by another user is extracted from an asymmetric KEK, the KEK is replaced in the repository by a symmetric KEK built using the user own MEK. For instance, suppose that Bob requires access to object $o_1$. Bob first retrieves from the descriptor of object $o_1$ the identifier $id(k_1)$ of DEK $k_1$. Then, it retrieves from $\mathcal{R}_B$ the corresponding KEK, decrypts it using his private key $s_B$ and uses the retrieved DEK for decrypting $o_1$. Furthermore, Bob replaces

the original asymmetric KEK with a symmetric KEK obtained by encrypting $k_1$ with his master key $m_B$.

When a new object $o_l$ is inserted into a container $C_i$, user $u$ retrieves the descriptor of the container and looks for the identifier $id(k_i)$ of the corresponding DEK $k_i$. The user will then look in her repository $\mathcal{R}_u$ for the KEK associated with $id(k_i)$ and will extract the corresponding DEK. The DEK will be used to encrypt object $o_l$ that will be given to Swift and DEK $id(k_i)$ will be inserted into the object descriptor. For instance, suppose that Alice inserts a new object $o_5$ in $C_2$. Since the DEK associated with $C_2$ is $k_2$, Alice encrypts $o_5$ with $k_2$, inserts $id(k_2)$ in the descriptor of $o_5$, and stores the encrypted version of $o_5$ in $C_2$.

## 2.3 Policy Updates

Since the authorization policy regulating access to objects in Swift is enforced through a policy-based encryption, every time the authorization policy changes, also the encryption policy needs to be re-arranged accordingly. Updates to the authorization policy include the insertion and deletion of users, objects, and authorizations. The insertion of a user requires the generation of her master key, user encryption key pair, and signing key pair, and the insertion of her public keys in Keystone. The removal of a user requires only the removal from Keystone of her public (encryption and signing) keys. The removal of an object instead requires its deletion from the container including it. We then focus on granting and revoking authorizations, and on the insertion of new objects. For simplicity, but without loss of generality, we consider policy updates that involve a single user $u_i$ and a single container $C$ (the extension to a set of users and of containers is immediate).

### 2.3.1 Enforcement of Policy Updates

We now illustrate how granting and revoking authorizations as well as the insertion of a new object with its authorization policy can be enforced. Recall that authorization policies operate at the granularity of container. Then, grant and revoke operations modify the set of users authorized to access a container $C$, and hence all the objects that it stores. Also, the insertion of an object in a container implies that it inherits the container acl.

**Grant authorization**. If user $u$ grants $u_i$ access to container $C$ (and hence to the content of all its objects), she simply needs to create an (asymmetric) KEK enabling $u_i$ to derive the DEK $k$ of the container and to store it in the repository $\mathcal{R}_{u_i}$ of user $u_i$. For instance, with reference to the authorization

policy in Figure 2.1(a), to grant Dave access to container $C_1$, Alice needs to create a KEK enabling Dave to derive $k_1$.

**Revoke authorization**. If user $u$ revokes from $u_i$ access to container $C$ (and hence to all its objects), it is not sufficient to delete the KEK that allows $u_i$ to derive the DEK $k$ of the container, as the revoked user $u_i$ may have accessed the KEK before being revoked and may have locally stored its value. A straightforward approach to revoke user $u_i$ access to container $C$ consists in replacing the DEK of the container with a new key $k_{new}$. However, this would require the owner $u$ of the container to download from the server all the objects in $C$, decrypt them with the original DEK $k$, encrypt them with the new DEK $k_{new}$, and then re-upload the encrypted objects, together with the KEKs necessary to authorized users to derive $k_{new}$. This would cause a significant performance and economic cost to user $u$. To limit such an overhead, we adopt over-encryption. Hence, when a user $u$ revokes from another user $u_i$ the authorization to access the objects in a container $C$, $u$ updates $C$'s acl and asks the storing server to over-encrypt the objects in $C$ with a SEL key $k^s$ that only non-revoked users can derive. Each container is then associated with a DEK $k$ at the BEL enforcing the initial authorization policy, and possibly also with a DEK $k^s$ at the SEL enforcing revocations. Also, there is a KEK for each user initially authorized for $C$ enabling her to compute $k$, and a KEK for each non-revoked user enabling her to compute $k^s$. For instance, consider the authorization policy in Figure 2.1(a), and assume that Alice wants to revoke from Bob the access to $C_2$. As illustrated in Figure 2.3, objects $o_2$ and $o_4$ are over-encrypted with a SEL key $k^s$. Also, the KEK enabling Bob to compute $k_2$ is dropped from $\mathcal{R}_B$, while the KEKs enabling Alice and Dave to compute $k^s$ are created and inserted into $\mathcal{R}_A$ and $\mathcal{R}_D$, respectively.

**Insert object**. When a new object $o_j$ is inserted into a container $C$, the object inherits the acl of the container. To enforce such an authorization policy, the object owner $u$ can simply decide to encrypt $o_j$ in the same way as the objects already in the container. However, if the authorization policy regulating access to the container has already been modified, this would require to encrypt $o_j$ with both the DEK at the BEL $k$ and the DEK at the SEL $k^s$ associated with the container. Since the policy of object $o_j$ has never been updated, the adoption of the SEL might be an overdo. We therefore propose to adopt a new DEK $k_{new}$ at the BEL to protect objects that are inserted into a container on which revoke operations had been applied. As a consequence of the revoke operation (and the new acl associated with the container), a new DEK BEL key (and the corresponding KEKs) corresponding to the new acl is generated for the container, and used for objects

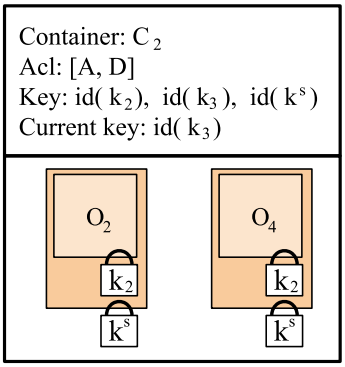



Figure 2.3: An example of revocation triggering over-encryption

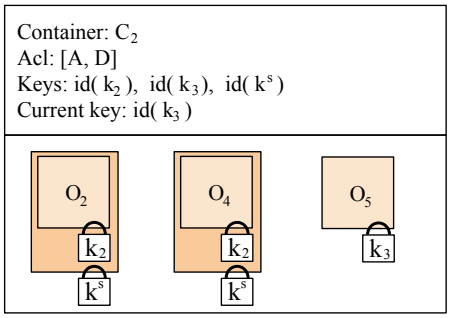



Figure 2.4: An example of insertion of an object into an over-encrypted container

that will be inserted into the container after the revoke operation. While for existing objects over-encryption is needed to guarantee protection from the revoked user, new objects can be encrypted with the new key known only to the users actually authorized for them. To enable non-revoked users to derive the new (current) key of the container, an (asymmetric) KEK enabling them to derive the new key is added to their repositories. Consider, as an example, container $C_2$ illustrated in Figure 2.3, which is encrypted with $k_2$ at the BEL and with $k^s$ at the SEL because of the revoke of Bob. Assume now that Alice needs to insert a new object $o_5$ into $C_2$. Object $o_5$ will be encrypted at the BEL with key $k_3$, generated when Bob has been revoked access to $C_2$ (together with the KEKs enabling Alice and Dave to compute $k_3$ from their own private key). Figure 2.4 illustrates the content of container $C_2$ after the insertion of $o_5$.

# Chapter 3

# The EncSwift Tool: Architecture and Design Choices

In this chapter, we illustrate the design of EncSwift, a solution (introduced in the previous chapter) that implements access control restrictions over data stored in OpenStack Swift without the intervention of the data owner as well as of the cloud provider. The core building block of EncSwift is the *Encryption Layer* module, which is in charge of managing encryption and decryption operations of Swift objects to enforce access control.

## 3.1 Key Management

The adoption of policy-based encryption for access control enforcement requires the definition and management of different keys, already introduced in Section 2.2.1. In that section we defined keys associated to each user, i.e., a RSA key pair, a Signature key pair, and a Master Key, and keys associated to data (depending on their acl), i.e., DEKs and KEKs. The first step of the realization of an hybrid encryption tool capable both to protect resources and to enforce access revocation, is to design an efficient key management solution. Keys associated to users are static and require a minimal storage effort, whereas keys associated to data can explode in number: we recall that each DEK is associated to a container (and thus to its acl), and that at every policy update a new DEK is generated for that container, together with a KEK for each user in the acl. In this section we focus on the realization of a key management system that permits to reduce the number of keys in case of frequent policy updates (Section 3.1.1), and to securely store them (Section 3.1.2).

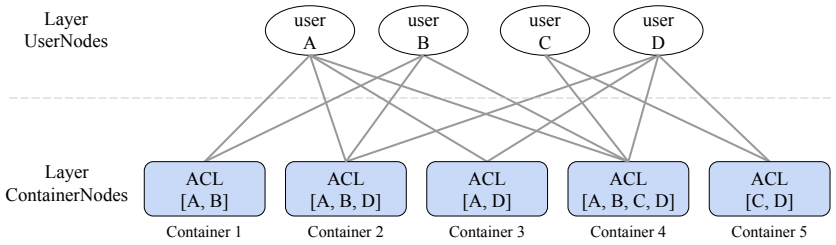



Figure 3.1: An example of a configuration with 4 users and 5 containers with different acls

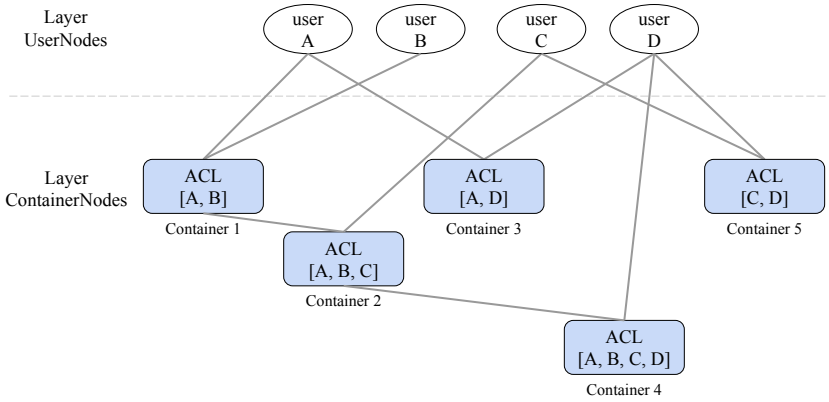



Figure 3.2: Graph with DEK derivation from other DEKs

### 3.1.1 Key Derivation Structure

The enforcement of sharing restrictions is driven by the definition of symmetric and asymmetric KEKs. To share access to a container with others, the data owner only needs to generate an asymmetric KEK for each of the users in the container acl, and a symmetric KEK for herself. These KEKs will permit all the authorized users (and the container owner) to retrieve the DEK of the container. Figure 3.1 graphically illustrates a scenario characterized by four users ($A$, $B$, $C$, $D$), represented by white nodes, and five containers (Container $1, \ldots,$Container 5), represented by colored nodes. Edges in the graph represent KEKs: an edge between a user and a container is a KEK corresponding to the encryption of the DEK of the container with the user public key. For instance, the two edges between users $A$ and $B$ correspond to two KEKs that allow users $A$ and $B$, respectively, to obtain the DEK used for encrypting the objects in Container 1.

To reduce the number of KEKs in the system (i.e., of edges in Figure 3.1), we introduce KEKs that encrypt a DEK with another DEK. A DEK can be

used to protect another DEK only if the latter can be accessed by a superset of users than the former. For instance, Figure 3.2 represents a set of KEKs that enforce the same authorization policy as Figure 3.1. Here, user $A$ should first obtain the DEK of Container 1 to be able to retrieve the DEK of Container 2, which in turn will enable her to obtain the DEK of Container 4.

### 3.1.2 Catalog Structure

Since the number of KEKs can be considerably high, they cannot be stored at the user side. We then define a *catalog* for each user, storing all and only the KEKs that the user needs to know for accessing the objects she is authorized to read. The catalog is stored in a Swift container, specifically created for this purpose. The containers storing users catalogs are stored in a specific tenant, used only for this purpose. In the following, we will refer to containers storing users catalogs as *meta-container* and to the corresponding tenant as *meta-tenant*. Note that, while we have a meta-container for each user, the meta-tenant is unique for the system.

When creating a new user, the Encryption Layer will initialize a meta-container for the user catalog. When creating a container, the data owner will create a symmetric KEK for herself, and an asymmetric KEK for each of the users in the acl of the container. These KEKs are inserted into the catalog of the corresponding user.

Note that, to improve efficiency in key derivation, a user can replace an asymmetric KEK with the corresponding symmetric counterpart in her catalog after the first access. This practice reduces the size of the KEK and makes future derivations more efficient.

## 3.2 Client-Only Encryption

This section describes two alternative architectures to perform policy-based encryption that do not require any modification to server components. In both architectures the encryption and decryption are performed at the client side. The *App-aware architecture* (Section 3.2.1) requires adjustments to the application that uses Swift, while the *Proxy architecture* (Section 3.2.2) is totally transparent to the application but requires an additional trusted proxy server, which may cause delays. Section 3.2.3 illustrates an approach, which can be integrated with both the App-aware and Proxy architectures, enabling users to store RSA and signature key pairs in the cloud.

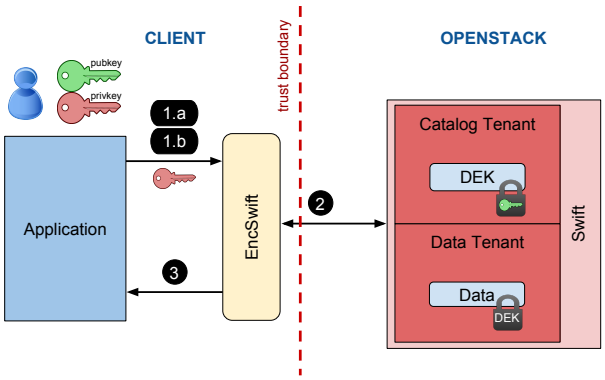



Figure 3.3: App-aware architecture

### 3.2.1 App-aware Architecture

Many applications that use Swift as storage service, also adopt the *python-swiftclient* library[1] that provides several high-level APIs for the communication between applications and Swift server. In the App-aware architecture, the EncSwift component that is responsible for encrypting and decrypting the resources is obtained by modifying of the *python-swiftclient* library (see Section 3.4).

Figure 3.3 illustrates the App-aware architecture for policy-based encryption as well as the steps followed by the client to access a remote object. The process to access an (encrypted) object operates as follows.

- *Step 1*: EncSwift Encryption Layer receives the user's request (*step 1.a*) and her private key (*step 1.b*).

- *Step 2*: EncSwift retrieves, from the user's catalog stored in Swift, the KEK corresponding to the DEK used to protect the object of interest, and decrypts it to obtain such a DEK. Also, EncSwift retrieves the encrypted object and decrypts it using the retrieved DEK.

- *Step 3*: When the object has been decrypted by EncSwift, it is forwarded to the application client.

Note that the process necessary to upload an object operates in a similar way. Since the APIs now need also the user's keys to encrypt/decrypt the data, the API parameters have been enriched. Hence, the application needs to be modified to invoke the enriched APIs.

[1] `https://github.com/openstack/python-swiftclient`

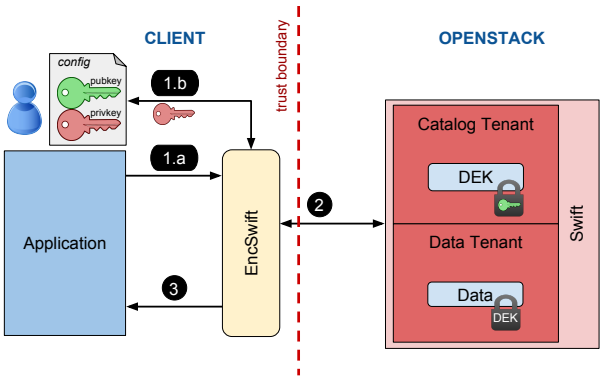



Figure 3.4: Proxy architecture

To prove the applicability of EncSwift in existing applications, we implemented a library that has been directly integrated into *SwiftBrowser*[2] (a simple web-app based on the Django web framework that permits to access Swift resources). The *python-swiftclient* library used by this application has been replaced with our modified version, to enrich its functionality with the Encryption Layer protection. In this way the user can interact with EncSwift using a browser.

During the login phase, the username, the tenant name, and the password must be provided. Then a session is created and managed by *SwiftBrowser* to store relevant data, including the authorization token issued by Keystone (the OpenStack identity service) that is used in place of the username and password.

Once the user has logged into the system, the GUI shows the list of containers that she is authorized to access. Using the web interface, the user can upload and download objects, access the object list of a specific container, and manage the container acls (i.e., grant/revoke authorizations).

The integration of EncSwift in SwiftBrowser preserves full compatibility with SwiftBrowser, which does not use any encryption layer.

### 3.2.2 Proxy Architecture

The App-aware architecture expects a modification of the application client. An adaptation to this architecture that provides enhanced transparency is the *Proxy architecture*, illustrated in Figure 3.4. To avoid changing the API parameters, in the Proxy architecture the EncSwift Encryption Layer reads a configuration file that defines the path of the user's keys. Then, the user does

[2]https://github.com/cschwede/django-swiftbrowser

not need to provide to the Encryption Layer her own private key every time she wants to retrieve an object from Swift. The steps followed by the client to access an object are the same as the ones illustrated for the App-aware architecture (Figure 3.3), with the difference that in the Proxy architecture the private key is provided by the configuration file rather than by the user (*step 1.b*).

With this approach the application can be fully unaware of the presence of the Encryption Layer. This permits to plug the Encryption Layer in every application without any modification, since the application can continue to use the regular APIs.

Note that the Encryption Layer can also be deployed as a trusted proxy server, which receives all the requests by clients and forwards them (after encryption of the objects) to the Swift storage service.

This architecture presents several advantages compared to the App-aware architecture. First, the proxy server providing EncSwift functionality offers the same APIs provided by Swift. This guarantees that all the existing libraries (e.g., JOSS Java OpenStack Storage [53]) and applications (e.g., Cyberduck [36]) can benefit from EncSwift functionality without any modification to their source code. Second, the trust boundary can be moved by placing the proxy server (Encryption Layer) in different locations. We then have a centralized proxy for an entire organization (i.e., all the users that belong to the same trust boundary) by running it on a trusted server (i.e., inside the organization network) or on a trusted cloud provider, while the data can be outsourced to a lower grade (and generally cheaper) cloud provider. In this case the entire organization is the EncSwift user. The proxy server can also be run directly by the client to keep the trust boundary as close as possible to the user.

The disadvantage of this architecture is that it relies on the presence of an additional trusted server running the Encryption Layer, which may introduce overheads and delays into the system.

### 3.2.3 Key Storage in the Cloud

One of the clearest trends of the past few years has been the adoption by users of mobile devices, replacing personal computers as the reference platform for carrying out their daily activities. The management of keys on these kind of devices seems to be unfeasible. Hence, the user has to outsource her own private and public keys, to store securely her own data on the cloud.

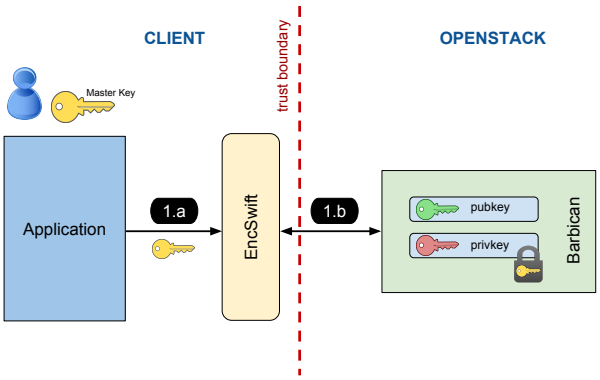



Figure 3.5: Key storage in the cloud

OpenStack already offers a component, *Barbican*[3], for the secure storage of private information. Instead of locally managing her RSA and signature encryption keys, the user only has to store a symmetric *Master Key* (MK) used to retrieve the private RSA and signature keys from Barbican. The Master Key is then used to encrypt the RSA and signature keys of the user. Figure 3.5 illustrates the architecture for key management, which can be integrated with both the App-aware and the Proxy architectures changing the working of *step 1* in the object download process. To download an object, the user then first provides to EncSwift her own Master Key (*step 1.a*). EncSwift uses such a key to decrypt the user's private RSA key retrieved from Barbican (*step 1.b*). Steps 2 and 3 then operate as illustrated for the App-aware and for the Proxy architectures discussed above.

## 3.3 Policy Evolution: Implementation of Over-Encryption

Over-encryption [30] prevents object re-encryption by requiring the server to add a further layer of encryption. Intuitively, every object has a first layer of encryption (*Base Encryption Layer*, BEL) imposed by the data owner to enforce the initial access control policy and to protect the confidentiality of the objects content from the server. A second layer of encryption (*Surface Encryption Layer*, SEL) is imposed by the server to enforce policy updates. Only users knowing both the BEL and the SEL encryption keys of an object can read its plaintext content.

[3] `https://wiki.openstack.org/wiki/Barbican`

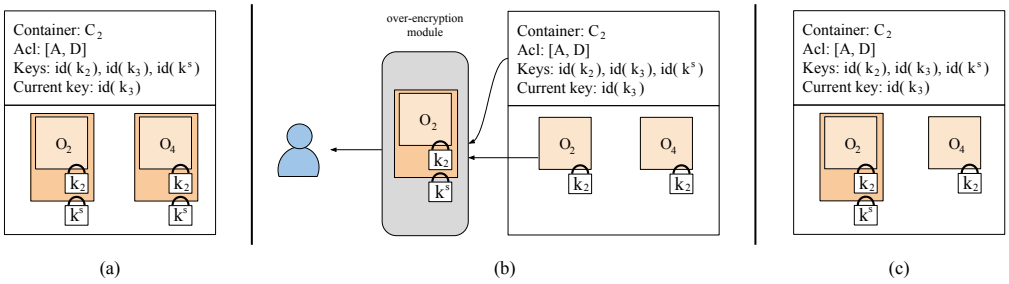



Figure 3.6: An example of implementation of a revoke operation using immediate (a), on-the-fly (b), and opportunistic (c) over-encryption

We note that, for efficiency reasons, objects inserted into a container after a policy change will be encrypted with a BEL key that reflects the new acl of the container. This limits the adoption of SEL to only the objects directly involved in a revoke operation. To this purpose, after a revoke operation, the data owner generates a new BEL DEK for the container, and inserts into the catalog of non-revoked users the KEKs necessary to retrieve the new DEK.

The implementation of over-encryption for the enforcement of revoke operations in Swift can operate in different ways, depending on the time at which SEL encryption is applied, which can be: materialized at policy update time (*immediate*), performed at access time (*on-the-fly*), or performed at the first access and then materialized for subsequent accesses (*opportunistic*). In the following, we elaborate on each of these strategies.

**Immediate Over-Encryption**. The storing server applies over-encryption when a user revokes the authorization over container $C$ to a user $u_i$. Immediate over-encryption requires the user to define, at policy update time: the SEL DEK $k^s$ necessary to protect the objects in the revoked container $C$, and the KEKs necessary to authorized users (and to the server) to derive $k^s$. Also, the objects in container $C$ will be over-encrypted. The server will then immediately read from the storage the objects in $C$, re-encrypt their content (possibly removing SEL encryption), and write the over-encrypted objects back to the storage. Hence, immediately after the policy update, the objects in $C$ are stored encrypted with two encryption layers. Every time a user needs to access an object in $C$, the server will simply return the stored version of the requested object. Figure 3.6(a) illustrates container $C_2$ in Figure 2.2 after Bob has been revoked access to $C_2$, when adopting immediate over-encryption.

Immediate over-encryption causes a considerable cost at policy update time, which is however significantly lower than the cost that would be paid if over-

encryption is not used. The advantage of immediate over-encryption lies in its simplicity in the management of `get` requests by clients, because objects will be returned by the server as they are stored. This approach can be an interesting option in scenarios where policy updates are extremely rare and the overall size of objects is modest.

**On-the-fly Over-Encryption**. The storing server applies over-encryption on-the-fly, that is, every time a user accesses an object. Then, even if the owner of the container asks the server to over-encrypt the objects in $C$, the server only keeps track of this request, but it does not re-encrypt stored objects. When a user needs to access an object in $C$, the server possibly over-encrypts the object before returning it to the user. Figure 3.6(b) illustrates the adoption of on-the-fly over-encryption when Alice accesses object $o_2$, after Bob has been revoked access to container $C_2$ in Figure 2.2. As it is visible from the figure, the server over-encrypts $o_2$ with $k^s$, which can be computed by Alice and Dave but not by Bob, before sending the object to the requesting user.
When adopting on-the-fly over-encryption, keys can be managed according to the following two strategies.

- *Static key generation*: the owner of the container defines, at revoke time, the SEL DEK $k^s$ necessary to protect the objects in the revoked container $C$, and the KEKs necessary to non-revoked users (and to the server) to derive $k^s$.

- *Dynamic key generation*: the server generates a fresh SEL DEK $k^s$ for every `get` request involving an object in the revoked container $C$. Also, it creates and makes available to the requesting user a KEK enabling her to derive $k^s$. At revoke time, the owner of the container only needs to communicate to the server the container $C$ subject to the revoke operation and the revoked user.

In terms of performance, if the same user makes repeated requests for objects in the same container (i.e., protected with the same DEK), dynamic key generation may require a greater amount of work. On the other hand, if the number of requests for the objects in a container is significantly lower than the number of KEKs produced by the static approach for the same container, the dynamic approach is more efficient. The profile of key management for the two alternatives presents significant differences, but key management operations exhibit negligible computational and I/O costs compared to the management of the objects themselves. This is the reason why in the experiments (Section 3.5), focusing on the overall object management cost, we do

not distinguish between static and dynamic key generation.
The advantage of on-the-fly over-encryption is that over-encryption is applied only when needed. However, if an object is asked multiple times during a period when the policy is stable, the server will incur a higher cost than immediate over-encryption, due to the multiple applications of encryption on the same object. On-the-fly over-encryption can then be an interesting option in scenarios where the ratio between accesses and revoke operations is low.

**Opportunistic Over-Encryption**. This approach aims at combining the advantages of both immediate over-encryption and on-the-fly over-encryption. It presents a similarity with the *Copy-On-Write* approach commonly used by operating systems to improve the efficiency of copying operations. Analogously to the immediate approach, opportunistic over-encryption requires the owner, when a user is revoked access to a container, to define both the SEL DEK $k^s$ necessary to protect the objects in the revoked container $C$, and the KEKs necessary to authorized users (and to the server) to derive $k^s$. Similarly to the on-the-fly approach, the server over-encrypts an object $o_j$ in the revoked container $C$ only when it is first accessed. However, instead of discarding it, the result of over-encryption is written back to storage for future accesses.
The management of opportunistic over-encryption is more complicated than the approaches illustrated above. In fact, after multiple policy updates and object insertions, a container may include objects associated with different BEL and SEL keys. Therefore, the object descriptor must specify also its state (i.e., not over-encrypted, over-encrypted with the most up-to-date SEL key, over-encrypted with an old SEL key). When a user needs to access an object $o_j$, the server first checks its descriptor. If $o_j$ is protected only at BEL and it has been subject to a revoke operation, the server derives the most recent SEL key and over-encrypts $o_j$ on-the-fly, storing then the result. If $o_j$ is protected also at the SEL with the most up-to-date key (or it is encrypted only at the BEL and no revoke operation affected the container), it is returned to the requesting user. Finally, if $o_j$ is protected at the SEL with an outdated key (e.g., because another revoke operation has been performed after $o_j$ has been last accessed), the server decrypts $o_j$ with the old SEL key, re-encrypts it with the new one, and stores the result. Note that KEKs enabling to derive old SEL keys can be dropped from repositories only when no object is protected with those keys. Figure 3.6(c) illustrates container $C_2$ in Figure 2.2 after Bob has been revoked access to $C_2$ and Alice has accessed object $o_2$. As it is visible from the figure, object $o_2$ is protected at both the BEL and SEL, while $o_4$ is encrypted only at the BEL as it has not been

| | |
|---|---|
| **bel_key_id** | Identifier of the current BEL DEK of the container, used to encrypt new objects uploaded in the container |
| **sel_key_id** | Identifier of the SEL DEK of the container; it is empty if over-encryption is not necessary for the container |

Figure 3.7: Metadata added to the container header

| | |
|---|---|
| **bel_key_id** | Identifier of the BEL DEK used to encrypt the object |
| **sel_key_id** | Identifier of the SEL DEK used to encrypt the object in storage; it is empty if the object does not need over-encryption or when using on-the-fly over-encryption mode |

Figure 3.8: Metadata added to the object header

accessed yet.
The critical advantage of opportunistic over-encryption is that it shows good adaptability to a variety of scenarios. In some peculiar combinations of policy update frequency, size of data collection, and access profile by clients, the other solutions may be preferable. However, based on our experimental results, we expect that this solution will be preferred in the majority of scenarios.

## 3.4 Implementation

This section describes the major design choices in the implementation of EncSwift.

### 3.4.1 Container and Object Headers

To enforce the access control policy, each object and container is associated with BEL and SEL DEKs. Each container is associated with at most one SEL DEK. In fact, every time a revoke operation implies over-encryption, a SEL DEK is generated and is associated with the container. SEL KEKs are updated according to policy changes and are always available in the catalog of each authorized user. Differently from SEL, each container may

include objects encrypted with different BEL DEKs as they may have been inserted into the container at different times (and hence with different acls. A BEL DEK therefore must be associated also with each object. However, each container is associated with only one current BEL DEK, reflecting the current acl of the container.

Figures 3.7 and 3.8 show the metadata added to the container and object headers, to allow users to retrieve the DEK necessary to encrypt/decrypt each object in the system.

### 3.4.2 Encryption Layer APIs

Our application offers a new set of routines that should be used in substitution of the official ones provided by *python-swiftclient* to take advantage of our policy-based encryption functionality. When the final user invokes one of these routines, the Encryption Layer in EncSwift, which is in charge of the dialog with the different modules of the OpenStack environment, manages it. In the following, we refer our discussion to the Encryption Layer APIs of the on-the-fly mode, with the note that the immediate and opportunistic modes operate in a similar way.

#### Create User Method

To create a new user, the Encryption Layer invokes the Keystone standard *create_user* method and generates a new RSA key pair and a new signature key pair, and stores them both on the local storage and on Barbican (encrypting the RSA and signature private keys with the user's Master Key). Then, the Encryption Layer communicates with Swift to create the meta-container with the standard *put_container* method. It then sets the acl of the meta-container to include the user only, using the traditional *post_container* method. Finally, the catalog is generated and stored in the meta-container.

#### Put Container Method

To create a new container, the Encryption Layer generates a new BEL DEK and produces the corresponding set of KEKs according to the acl of the container. Clearly, when a container is created, no SEL is required since the container (and its objects) are protected only with BEL encryption. The container is then inserted into Swift using the traditional *put_container* method, properly initializing the metadata in the header of the container (Figure 3.7).

### Put Object Method

To insert a new object into a container, the traditional *put_object* method is modified to guarantee that the object is uploaded in encrypted (in contrast to plaintext) form, using a key that enforces the container acl. To this aim, the Encryption Layer retrieves, from the header of the container, the identifier of the BEL DEK. If the user who invokes the put object method is authorized for the container (i.e., she appears in the acl), the Encryption Layer asks Swift for the user's catalog, to retrieve the KEKs necessary to obtain the BEL DEK associated with the container. The Encryption Layer then uses such a BEL DEK to encrypt the object and puts it in the container, invoking the traditional *put_object* method.

### Get Object Method

To retrieve an object from a container, it is first necessary to retrieve the identifier of the BEL DEK and SEL DEK used for the object. The Encryption Layer then asks Swift to retrieve the stored object, using the traditional *get_object* method. The Swift server then verifies whether the object has to be protected with SEL encryption. If the **sel_key_id** of the container is empty or if the **bel_key_id** of the container is the same as the **bel_key_id** of the object, over-encryption is not necessary. In this case, either the policy of the container has never been updated, or the container has not been subject to revoke operations after the insertion of the object. The object is then returned to the client. At this point, two (or four) decryptions occur at the client side: one to obtain the DEK (at BEL and, if necessary, also at SEL) from the KEKs, and one to decrypt the object (at BEL and, if necessary, also at SEL).

### Post Container Method

To grant or revoke users access to a container, we use the *post_container* method, which is used to change the meta-information associated with a container and then also its header.

When a user invokes the *post_container* method, the Encryption Layer first checks if the post operation requests to change the acl (and whether it is a grant or a revoke operation). If this is the case, the Encryption Layer retrieves the container header, to obtain the acl and the identifier of the current BEL DEK and of the SEL DEK. When a user is added to a container acl, the Encryption Layer computes the BEL KEKs and SEL KEKs necessary to the granted user to access the BEL DEK and SEL DEK, respectively. These KEKs are then stored in the catalog of this user. When

a user is removed from a container acl, over-encryption is required. Hence, the Encryption Layer creates a new SEL DEK (possibly substituting the old one) and updates the catalogs of the users in the current acl with the new KEKs (possibly removing the KEKs used to obtain the old SEL DEK). In the immediate mode, the server also re-encrypts the objects with the new SEL DEK.

### 3.4.3 Catalog Management Service

To guarantee the security and the consistency of users catalogs, they can be updated only by users with administrative privileges. Since also users who do not have these privileges may need to update catalogs (i.e., when granting other users access to their containers), we introduce an always-listening *Catalog Management Service*. This service provides an API that users can invoke to update catalogs when a container acl is changed. The Catalog Management Service checks if the request is valid and guarantees a correct update of all the catalogs of the users involved in the grant/revoke operation. In this way, there is only one authorized entity that can change the users' catalogs. When a user creates a container, she also invokes the Catalog Management Service API, providing the KEKs that must be inserted into the catalogs of the users in the container acl.

Since the version of Keystone that we used for our implementation requires administrative privileges to obtain the user ID that corresponds to a username and vice versa, and Swift acls include user IDs, our Catalog Management Service also converts user IDs into usernames and vice versa.

### 3.4.4 Swift Middleware Pipeline

To apply over-encryption it is also necessary to modify the server-side Swift service, which is based on *WSGI*, Web Server Gateway Interface (a modular interface between the web server and the web application) and on Paste Python Framework. WSGI permits to define a pipeline where several components (the middleware) process and modify the request before it reaches the main web server component (e.g., the Proxy node), and the response they get from it. To add over-encryption functionality to Swift, we introduced two new components, the *key_master* and the *encrypt* components, in the Proxy node pipeline. The *key_master* component is in charge of retrieving the correct DEKs, while the *encrypt* component applies SEL encryption before returning the object to the client. Including these components in the Proxy node pipeline has the advantage of leveraging the work done by the components preceding it in the pipeline (e.g., authentication and request validation).

## 3.5 Experimental Results

We discuss the experimental results performed for evaluating the practical applicability of our proposal. We performed different series of experiments aimed at evaluating the following aspects:

- the benefits of the use of over-encryption compared to a system where policy changes are enforced by the client downloading, re-encrypting, and re-uploading the objects involved (Section 3.5.1);
- the performance of the immediate, on-the-fly, and opportunistic options (Section 3.5.2);
- the performance of a batch and a streaming option for the execution of encryption by the server (Section 3.5.3);
- the performance at the client-side for the removal of the two encryption layers for over-encrypted objects (Section 3.5.4).

The experiments were executed on two PCs with Linux Ubuntu 16.04, 16 GB RAM DDR3, 4-core i7-4770K CPU, 256 GB SSD disk. The client and the server were connected with a 100 Mbps network channel.

### 3.5.1 Comparison between Client Re-Encryption and Over-Encryption

We compare different options of over-encryption with a scenario where a policy update on a container is enforced by the data owner through the download, re-encryption and upload of the whole container. For this set of experiments, we consider a container with 1000 files of size 1 MB. Client side re-encryption does not require server work (except for the download and upload request, which are the same in every scenario) and is necessary only for revocations.

Figure 3.9 compares the overall time required for the management of a policy update followed by a number of `get` requests. The line on top corresponds to the configuration without over-encryption. In the lower part, we have the lines that describe the time required when using over-encryption, considering the on-the-fly approach and the opportunistic approach with uniform distribution of access requests (corresponding to $\alpha = 1$). We also report the time exhibited by the management of a sequence of direct `get` requests, where no encryption is applied to the objects. The graph shows that the lower lines are all one near to the other, proving that over-encryption has a small overhead.

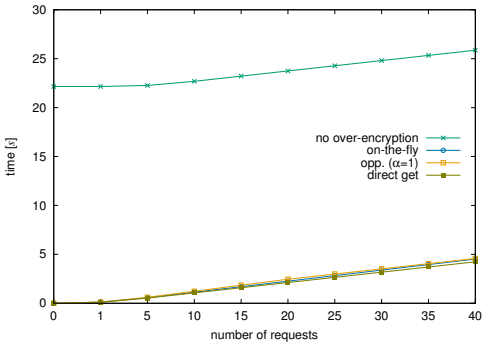



Figure 3.9: Overhead of all the solutions

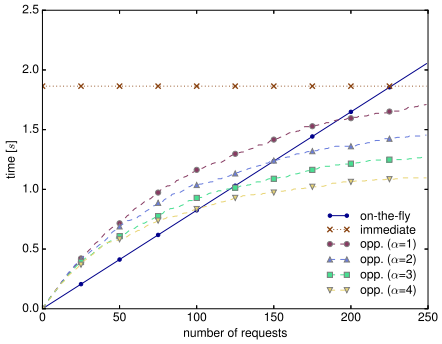



Figure 3.10: Cumulative server work with different over-encryption approaches

### 3.5.2 Analysis of Over-Encryption Approaches

We compare the performance of immediate, on-the-fly, and opportunistic approaches. For this set of experiments, we consider a container with 100 files of size 1 MB. We focus on the time required for the processing on the server module, without considering the time required for the transfer of data across the network. This permits to focus on the component that is most influenced by these options (the network is typically a bottleneck and it hides the difference between the approaches, as shown in Figure 3.9). Figure 3.10 reports the cumulated execution time associated with a sequence of requests, for the three over-encryption approaches.

The immediate option requires, at policy update time, to read all the objects in the container, possibly decrypt them, and encrypt and write them back. This creates an immediate overhead at policy update, before the first request. Subsequent requests do not require a specific processing by this module, which manages the `get` requests with a direct mapping to the retrieval of the over-encrypted representation of the object. Figure 3.10 represents the immediate approach with a horizontal line.

The on-the-fly option requires to apply SEL encryption on every returned object. The cost is then identical for all the requests. Figure 3.10 shows that the on-the-fly option is associated with a constant growth.

For the opportunistic approach, the cost depends on the number of files in the container that are accessed more than once. When an object is accessed for the first time after the policy update, the server will have to encrypt it at the SEL level and then save its new representation. This adds to the encryption cost the cost for the storage of the new version. Subsequent

requests for the same object will be managed as a simple `get` of the over-encrypted representation of the object. The frequency of repeated accesses has then an impact on the efficiency of this approach. In our experiments, we therefore consider request profiles associated with power law distributions [39] with varying values for the $\alpha$ parameter, from 1 to 4. A value of $\alpha$ equal to 1 corresponds to a uniform distribution, where all the requests have an equal probability of asking any of the objects in the container; increasing values of $\alpha$ lead to an increasingly skewed distribution of requests. The analysis shows that for the first requests the cost associated with the opportunistic approach is greater than that of the on-the-fly approach. As requests continue to be executed, the opportunistic approach becomes increasingly more efficient compared to the on-the-fly approach. The advantage increases as the profile becomes more unbalanced. The worst case is represented by the uniform distribution, which still becomes more efficient after 180 requests.

From this experimental analysis we conclude that the choice of the over-encryption approach has to consider a few aspects. In terms of pure performance, the opportunistic approach always dominates the immediate approach. The choice between the on-the-fly and the opportunistic approach has to evaluate the frequency of policy updates, the number of access requests generated between each policy update, and the profile of access requests. For scenarios where policy updates are relatively frequent compared to the frequency of access requests, and the profile is uniform, the on-the-fly approach can be the most efficient solution. In these scenarios, a choice should be made between the static and dynamic key generation. This choice will have to take into account design and configuration aspects, with the static generation requiring a greater upfront processing, but then more efficient computation, and the dynamic generation minimizing setup costs, but requiring a DEK and a KEK creation for every access request. In domains with a profile opposite to that leading to the on-the-fly approach, the opportunistic approach can prove to be the best option.

In addition to performance, there are design and security requirements that may have an impact on the choice. In terms of design, the opportunistic approach requires a more complex procedure, whereas the immediate and on-the-fly approaches both map to a simpler implementation. With respect to security, the immediate approach (for all the objects) and the opportunistic approach (for objects that have already been accessed since the last update) offer greater protection, because a revoked user who may have access to the Swift storage infrastructure would not be able to access the plaintext of the objects, whereas in the on-the-fly approach such an attack would succeed for a revoked user. System administrators will then have to make a choice based

on the consideration of a number of parameters. Our expectation is that in most scenarios administrators will select the opportunistic approach.

### 3.5.3 Streaming and Batch Encryption

We performed a set of experiments aimed at comparing the execution time of a number of `get` requests when two different kinds of encryptions are used by the server: *Streaming* and *Batch*. They both use the AES-CTR encryption mode. Streaming encryption makes use of the WSGI structure of the Swift servers, and it consists in encrypting every chunk of the file as it is obtained from the proxy server. On the contrary, Batch encryption consists in encrypting the whole file after it is returned from the proxy server and before it is sent to the client. In these experiments, files of the same size are inserted into a container, which has the total size of 1 GB. We studied the benchmark of Streaming and Batch encryption applied to the on-the-fly approach against the direct `get` call that does not apply any encryption.

As it is visible from Figure 3.11, compared to the direct `get` call, Streaming encryption adds an overhead between 1% and 3%, whereas Batch encryption adds an overhead between 7% and 15%. It is then clear that Streaming encryption is more efficient, both because of shorter response times and because it has a lower memory usage, since it does not have to load the entire object in RAM before encrypting it. Note that the encryption of the chunks could also be parallelized, further reducing the overhead compared to the direct `get` call.

### 3.5.4 Application of Two Encryption Layers

When over-encryption is used, the client has to decrypt the downloaded objects twice, using the same encryption algorithm with two distinct keys. The simplest approach for the implementation of these two decryptions consists in first removing the SEL layer on the full object and then removing the BEL layer. Such an approach is not the most efficient option, because the portion of the object that has been SEL-decrypted (and still BEL-encrypted) will have to either be temporarily stored in RAM or on mass memory. This is similar to the analysis for Streaming and Batch encryption for the server, where Streaming encryption proves to be more efficient.

We started from these considerations and investigated the joint application of SEL and BEL decryptions. We were also interested in evaluating the performance profile of decryption on the client and in evaluating the impact of the hardware support offered for the execution of cryptographic functions.

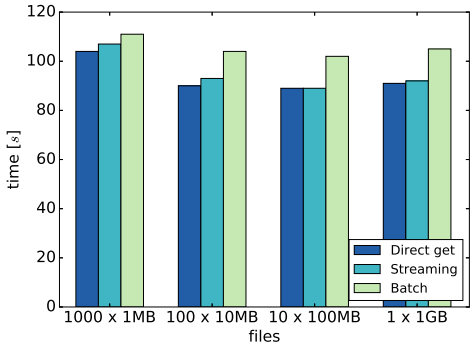



Figure 3.11: Comparison of the overhead caused by Streaming and Batch on-the-fly approaches with respect to the direct `get` call

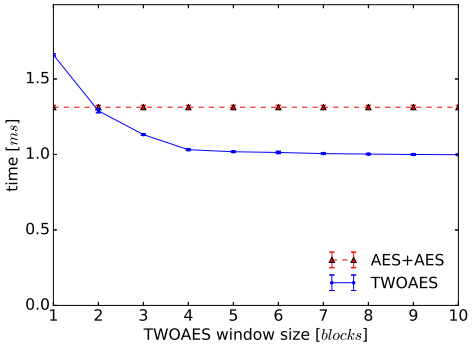



Figure 3.12: BEL+SEL encryption performance on a 1MB file using two subsequent AES invocations and TWOAES

| | **without AES-NI** | | | **with AES-NI** | | |
|---|---|---|---|---|---|---|
| | **ECB** | **CBC** | **CTR** | **ECB** | **CBC** | **CTR** |
| **128 bits** | 253 MB/s | 215 MB/s | 154 MB/s | 1857 MB/s | 408 MB/s | 284 MB/s |
| **256 bits** | 192 MB/s | 170 MB/s | 133 MB/s | 1301 MB/s | 336 MB/s | 248 MB/s |

Figure 3.13: AES encryption rate for the modes *ECB*, *CBC*, and *CTR* using the *pycrypto* library without and with AES-NI

In particular, we verified the impact of the AES-NI (Intel AES New Instruction set) instructions available on Intel processors. A first set of experiments, reported in Figure 3.13, showed that the encryption performance of AES-NI compared to an AES software implementation (we used the one available in OpenSSL) is around 7 times faster.

We then focused on the application of two decryptions. Our expectation was that the consecutive application of a SEL decryption and BEL decryption on the same block would have produced a benefit, as it would have avoided to pay the penalty of a transfer outside the CPU cache of the data. As shown in Figure 3.12, where AES-NI instructions were used, we instead observed that the performance of the interleaved decryption depends on the number of consecutive blocks processed with each key. The worst performance is observed when after each block there is a switch of encryption key. Further investigation allowed us to verify that the source of this behavior was an optimization by the C compiler that avoided to execute a write to

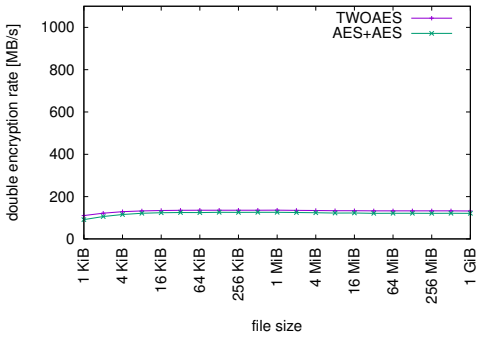



Figure 3.14: Re-encryption using AES

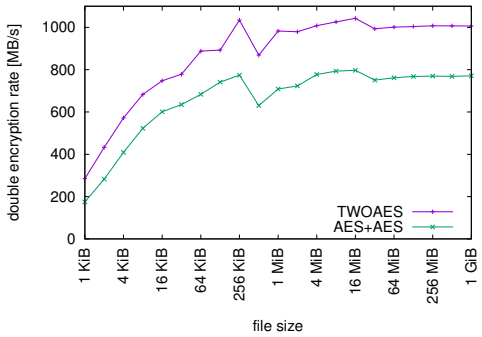



Figure 3.15: Re-encryption using AES-NI

the registers storing the key value when no changes had occurred to the key since the previous execution. When the switch from the application of the SEL decryption to the BEL decryption occurs after a number of blocks, the cost of the key setup is amortized over a number of blocks, but the blocks remain in the CPU cache after the first decryption and the second decryption becomes more efficient.

We then compared the execution times for the (a) serial application of SEL and BEL decryption (a full SEL decryption, followed by a full BEL decryption) and (b) interleaved SEL and BEL decryption, with the application of the two decryptions 8 blocks at a time. Figures 3.14 and 3.15 report the results of these experiments when not using AES-NI and when using AES-NI, respectively. The greater performance of hardware-accelerated AES emphasizes the impact that the CPU/RAM interface has on performance. Figure 3.14 indeed shows that the difference between the two approaches when hardware acceleration is not used is limited. Figure 3.15 shows that the 20% benefit observed is persistent across objects with a variety of sizes.

This approach is then the one that has to be applied whenever two layers of decryption have to be removed. It is also important to note that the throughput that can be obtained in the application of two decryptions (a few GB/s) is orders of magnitude greater than the bandwidth available for the network connections between a client and the Swift provider. This confirms the applicability of over-encryption in this scenario.

# Chapter 4

# EncSwift and Key Management: An Integrated Approach in an Industrial Setting

In a dynamic system with frequent policy updates, the use of encryption brings not negligible complexity since keys must be shared and revoked in an efficient and trusted way. Indeed, as widely explained in the previous chapters, as though granting access to a resource only requires to share its encryption key, access revocation is a more relevant problem. To this aim, we have analyzed how over-encryption can provide a clear benefit both in terms of performance and security. Yet, intuitively, in case of frequent policy updates, our approach would benefit from an efficient key management service able to manage a potentially large number of encryption keys. The purpose of this chapter is to introduce a new efficient key management service, i.e., *BT KMS*, and investigate its integration with EncSwift, in a scenario with frequent policy updates, and thus stressed by the generation and administration of a large number of encryption keys.

## 4.1 BT Key Management Service

The main purpose of the BT Key Management Service (KMS) is to allow users to securely manage their encryption keys and certificates securely, either in a cloud based or in an on-premise environment. Furthermore, in a cloud based setting, the BT KMS can be provisioned and managed via the BT Service Store [37], which enables its users to create isolated and com-

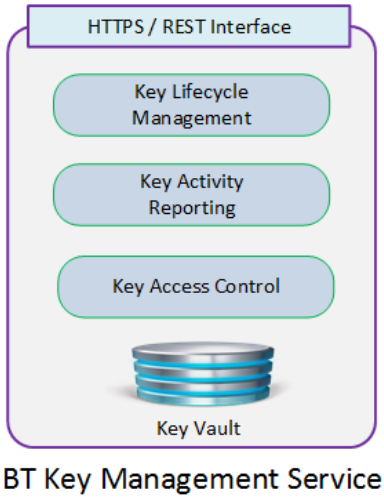



Figure 4.1: Architecture of the BT KMS

partmentalized key management domains. This allows the users to manage their keys for use in multiple cloud platforms in a secure multi-tenant environment. The main components of the BT KMS are shown in Figure 4.1 and are described in more detail below.

- *HTTPS/REST Interface*: it offers an intuitive web-based management console for enterprise-scale administration, as well as Single Sign-On capabilities that can be seamlessly integrated with LDAP based user identity management services. It also supports management through the use of a full-featured REST based API.

- *Key Lifecycle Management*: it provides complete lifecycle management of the keys, including key creation, storage, import, export, rotation, revocation and deletion. Thus users have complete control over every aspect of a key during its existence. It also supports multiple API standards like PKCS#11, Microsoft Extensible Key Management (EKM) and OASIS KMIP, which helps with the automation of various operational tasks.

- *Key Activity Reporting*: it implements the ability to audit and report on all activities relating to keys, so that the users are able to generate comprehensive and granular audit logs of encryption key and certificate management activities. This is an essential requirement for a lot of compliance standards, and BT KMS complies with the FIPS 140-2 Level 1 standard.

- *Key Access Control*: lastly, it is tightly integrated with an access control capability that governs the rules and policies for releasing the relevant data encryption keys to the authorized users and processes. It also enforces the storage and retrieval to and from the key vault, which provides high availability storage and backup of the keys.

One of the core requirements from users of cloud based services is maximum and transparent control and ownership of their data in the cloud ecosystem. This can be realized by utilizing the BT KMS, as it enforces the user-based management of the cryptographic keys, so that only the users are able to authorize policy-based release of keys to trusted applications. Even the cloud service provider on which the BT KMS is deployed has no view or control of the users' keys and other security credentials. This approach also keeps the keys separate from the data that they are protecting.

Each instance of the BT KMS contains a public/private key pair (RSA Key Pair), which is used to protect the Key Encrypting Keys (KEKs) and other sensitive security objects stored within the Key Vault. The RSA Key Pair can be changed periodically without any impact to the keys and credentials being protected by it. A KEK is a random AES-256 key which is used to protect a Data Encryption Keys (DEK) while they are at-rest in the Key Vault or in transit from the BT KMS to the client application. The DEKs are AES-256 keys that are used for the actual encryption of the objects, and are always encrypted at-rest and in transit. This minimizes the exposure of the DEK, as well as the need for its frequent rotation.

## 4.2 Integration between BT KMS and EncSwift

In the previous chapter (Section 3.2.2), we illustrated an architecture to perform policy-based encryption on the client-side, relying on an external component (Barbican) in order to store encryption keys in the cloud. In this section, we analyze possible challenges and enhancements to the EncSwift solution in case of adoption of the BT KMS in place of Barbican. Obviously, a research tool such as EncSwift would benefit from the integration of a full real industrial key management service. Indeed, BT KMS is designed to bring good performance and to be scalable in a real industrial scenario, and thanks to its flexibility and the wide number of communication protocols it supports, BT KMS would fully fit into the EncSwift architecture. Nevertheless, this integration would require a low implementation effort in order to fix some differences in the key usage. First of all, both BT KMS and EncSwift

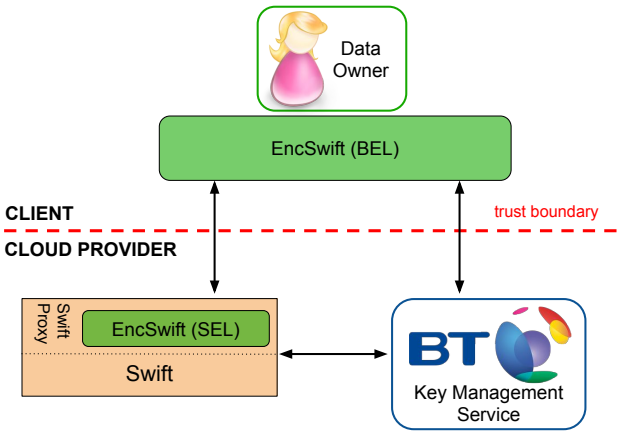



Figure 4.2: Architecture of EncSwift and BT KMS integration

requires to store KEKs. Yet, in EncSwift KEKs are produced as symmetric DEKs asymmetrically encrypted with users' RSA keys, on the contrary BT KMS supports encryption of DEKs with a symmetric KEKs, that are encrypted themselves with users' RSA keys. A possible integration of the two architectures would require to choose one of the key treatment alternatives.

In our analysis, we adopt the EncSwift key structure, and deprecate the use of BT KMS encrypting keys, i.e., DEKs are asymmetrically encrypted with users' RSA keys. An architectural overview of the integration is shown in Figure 4.2. EncSwift still relies on an OpenStack Swift object storage, but all the encryption keys are stored in an instance of BT KMS instead of Barbican. Moreover, the EncSwift SEL middleware implemented to enforce over-encryption can communicate with BT KMS in order to store and retrieve the SEL DEK.

# Chapter 5

# Related work

The design of encryption techniques for data stored in the cloud is a large research area, with a considerable variety of topics and proposals. A significant amount of work has been dedicated to the design of techniques that support the efficient search and retrieval of encrypted data (e.g., [83]). Techniques have been designed that let the data be available only to users with specific properties (e.g., ABE [22, 44]). Another important line of research focuses on protecting access privacy (e.g., [33, 34, 77]). In the first part of this thesis, we focus on techniques adopting encryption in order to protect resources stored in the cloud. We analyze server-side and client-side encryption approaches already in use in existing cloud storage frameworks, and focus on recent hybrid encryption techniques recently raising interest in the industrial and academic communities.

Several proposals have contributed to the design of solutions for the protection of outsourced data with reference to current cloud frameworks. In [2], OpenStack security issues are extensively analyzed. The confidentiality of objects stored in Swift is considered as a significant aspect, but no specific technical solution is presented. A subsequent work by the same authors [1] describes an approach for the encryption of objects in Swift. In [55] another approach for server-side encryption is presented, with the goal of protecting "data-at-rest" (i.e., an approach for making the object representation on storage devices protected against physical accesses). In these approaches, keys are never seen by clients and they do not consider the support for container acls. Then, they do not have to look at the management of the encryption policy and its evolution.

A number of proposals have considered the application of encryption on the client-side. In [85], a service is presented that maps a file system to an encrypted representation on Amazon S3. The proposal does not support the sharing of files among distinct users and acls are not considered. In [54], an

architecture for sharing encrypted objects outsourced to a cloud provider is presented. Revocation is considered as important and difficult and the proposed solution enforces it by limiting access to encryption keys for revoked users. In [86], an extensive architecture for the management of a cloud-based data sharing system is proposed. Resources are protected with keys that are consistent with the policy and significant attention is paid to revocation. The approach used is based on proxy re-encryption and lazy re-encryption. Proxy re-encryption relies on expensive cryptographic techniques that allow a server to convert a representation of a resource encrypted with a key to one associated with a different key, without letting the server executing the transformation be able to access the plaintext of the resource. Proxy re-encryption supports expressive encryption schemes, which allow attribute-based selection. Over-encryption uses standard symmetric encryption, which does not support those features but exhibits better performance. Lazy re-encryption shares some features with our opportunistic over-encryption approach, as it saves on re-encryptions by applying them only after an access request is made to the object, but the motivation is different. The advantage of lazy re-encryption is due to the ability to avoid re-encryptions for resources that are not accessed between a number of policy updates. The same benefit is also valid in our opportunistic approach, but in those scenarios our on-the-fly approach can be preferable.

Over-encryption has been proposed to effectively and efficiently enforce policy updates over encrypted outsourced data [29, 30]. This solution considers the presence of a single data owner, and it has been extended to consider multiple users owning (and willing to share) data [28]. This approach differs from the solution we proposed as it relies on Diffie-Hellman, while our approach is based on the definition of symmetric and asymmetric KEKs. Also, these proposals consider a generic resource management scenario, with no specific connection to existing cloud frameworks.

# Chapter 6

# Discussion and conclusions

The design of efficient techniques for protecting the confidentiality and regulating access to data stored at external cloud providers has been the subject of several efforts in the research as well as industrial community. In this chapter, we have presented an overview of recent approaches that protect the confidentiality of the data through encryption as well as enforce access control restrictions. These techniques mainly differ in how encryption is enforced, which depends on the trust assumption on the cloud provider. Interesting evolution of these encryption-based data protection techniques are related to the use of All-or-Nothing Transforms (AONT) for enforcing changes in the access control policy without requiring the support of the cloud provider [12], and the consideration of novel distributed cloud storage systems (e.g., Storj [84], Sia [82] and FileCoin [59]) characterized by the availability of multiple (untrusted) nodes that can be used to store resources in a distributed manner.

# Part II

# AONT

# Chapter 1

# Mix&Slice: Efficient Access Revocation in the Cloud

## 1.1 Introduction

In the last years we assisted to an increasing interest in encryption techniques. When dealing with data protection, more and more solutions rely on encryption, both on data in transit (e.g., SSL) and on data-at-rest. Robust encryption has become computationally inexpensive, enabling its introduction in domains that are traditionally extremely sensitive to performance (like cloud-based applications and management of large resources). Moreover, encryption provides protection against the service provider itself, which may not be considered authorized to know the content of the resources it stores. In addition to this, encryption solves the need of having a trusted party for policy enforcement: resources enforce self-protection, since only authorized users, holding the keys, will be able to decrypt them.

*All-or-nothing transform* (AONT) is a family of encryption techniques, first proposed by Rivest in [71], which provide better security guarantees with respect to simple encryption. AONT ensures that if even a single block of the AONT-encoded resource is missing, no portion of the resource can be reconstructed, even if the encryption key is known.

The goal of this part of thesis is to present AONT approaches used for protecting data-at-rest, introducing a novel solution that can also be employed to enforce efficient access revocation, i.e., Mix&Slice [12]. We then focus on an interesting application of AONT to a new scenario, i.e., decentralized cloud storage, as a way to balance availability and security.

## 1.2 Basic Idea

In this chapter, we present a novel approach to enforce access revocation that provides efficiency, as it does not require expensive upload/re-upload of (large) resources, and robustness, as it is resilient against the threat of users who might have maintained copies of the keys protecting resources on which they have been revoked access.

The basic idea of our approach is to provide an encrypted representation of the resources that guarantees complete interdependence (*mixing*) among the bits of the encrypted content. Such a guarantee is ensured by using different rounds of encryption, while carefully selecting their input to provide complete mixing, meaning that the value of each bit in the resulting encrypted content depends on every bit of the original plaintext content. In this way, unavailability of even a small portion of the encrypted version of a resource completely prevents the reconstruction of the resource or even of portions of it. Brute-force attacks guessing possible values of the missing bits are possible, but even for small missing portions of the encrypted resource, the required effort would be prohibitive. Rivest's all-or-nothing transform [71] considers similar requirements, but the techniques proposed for it are not suited to our scenario, because they are based on the assumption that keys are not known to users, whereas in our scenario revoked users can know the encryption key and may plan ahead to locally store critical pieces of information.

Trading off between the potentially clashing need of connecting all bits of a resource to provide the wished interdependency of the content on one side, and the potential huge size of the resources and need to maintain a possible fine-granularity of access within the resource itself on the other side, we apply the idea of mixing content within portions of the resource, enforcing then revocation by overwriting encrypted bits in every such portion. Before mixing, our approach partitions the resource in different, equally sized, chunks, called *macro-blocks*. Then, as the name hints, it is based on the following concepts.

- *Mix:* the content of each macro-block is processed by an iterative application of different encryption rounds together with a carefully designed bit mixing, that ensures, at the end of the process, that every individual bit in the input has had impact on each of the bits in the encrypted output.

- *Slice:* the mixed macro-blocks are sliced into fragments so that fragments provide complete coverage of the resource content and each fragment represents a minimal (in terms of number of bits of protection,

which we call *mini-block*) unit of revocation: lack of any single fragment of the resource completely prevents reconstruction of the resource or of portions of it.

To revoke access from a user, it is sufficient to re-encrypt one (any one) of the resource fragments with a new key not known to the user. The advantage is clear: re-encrypting a tiny chunk of the resource guarantees protection of the whole resource itself. Also, the cloud provider simply needs to provide storage functionality and is not required to play an active role for enforcing access control or providing user authentication. Our Mix&Slice proposal is complemented with a convenient approach for key management that, based on key regression, avoids any storage overhead for key distribution.

In [12], the evaluation of Mix&Slice is also supplied with its security analysis, together with an implementation written in C and applied to OpenStack Swift. In this thesis, we decided to only present the basic principles and building blocks of Mix&Slice, in order to leave more space to the application of AONT solutions (such as Mix&Slice) to decentralized cloud storage.

## 1.3 Mix & Slice

### 1.3.1 Blocks, mini-blocks, and macro-blocks

The basic building block of our approach is the application of a symmetric block cipher. A symmetric cryptographic function operating on blocks guarantees complete dependency of the encrypted result from every bit of the input and the impossibility, when missing some bits of an encrypted version of a block, to retrieve the original plaintext block (even if parts of it are known). The only possibility to retrieve the original block would be to perform a brute-force attack attempting all the possible combinations of values for the missing bits. For instance, modern encryption functions like AES guarantee that the absence of $i$ bits from the input (plaintext) and of $o$ bits from the output (ciphertext) does not permit, even with knowledge of the encryption key $k$, to properly reconstruct the plaintext and/or ciphertext, apart from performing a brute-force attack generating and verifying all the $2^{\min(i,o)}$ possible configurations for the missing bits [3].

Clearly, the larger the number of bits that are missing in the encrypted version of a block, the harder the effort required to perform a brute-force attack, which requires attempting $2^x$ possible combinations of values when $x$ bits are missing. Such *security parameter* is at the center of our approach and we explicitly identify a sequence of bits of its length as the atomic unit on which our approach operates, which we call *mini-block*. Applying block

encryption with explicit consideration of such atomic unit of protection, and extending it to a coarser-grain with iterative rounds, our approach identifies the following basic concepts.

- *Block*: a sequence of bits input to a block cipher (it corresponds to the classical block concept).
- *Mini-block*: a sequence of bits, of a specified length, contained in a block. It represents our *atomic unit* of protection (i.e., when removing bits, we will operate at the level of mini-block removing all its bits).
- *Macro-block*: a sequence of blocks. It allows extending the application of block cipher on sequences of bits larger than individual blocks. In particular, our approach operates *mixing* bits at the macro-block level, extending protection to work against attacks beyond the individual block.

Our approach is completely parametric with respect to the size (in terms of the number of bits) that can be considered for blocks, mini-blocks, and macro-blocks. The only constraints are for the size of a mini-block to be a divisor of the size of the block (aspect on which we will elaborate later on) and for the size of a macro-block to be a product of the size of a mini-block and a power of the number of mini-blocks in a block (i.e., the ratio between the size of a block and the size of a mini-block). In the following, for concreteness and simplicity of the figures, we will illustrate our examples assuming the application of AES with blocks of 128 bits and mini-blocks of 32 bits, which corresponds to having 4 mini-blocks in every block and therefore operating on macro-blocks of size $32 \cdot 4^x$, with $x$ arbitrarily set. In the following, we will use *msize*, *bsize*, *Msize* to denote the size (in bits) of mini-blocks, blocks, and macro-blocks, respectively. We will use $b_j[i]$ ($\mathrm{M_j}[i]$, resp.) to denote the $i$-th mini-block in a block $b_j$ (macro-block $\mathrm{M_j}$, resp.). We will simply use notation $[i]$ to denote the $i$-th mini-block in a generic bit sequence (be it a block or macro-block), and $[[j]]$ to denote the $j$-th block. In the encryption process, a subscript associated with a mini-block/block denotes the round that produced it.

### 1.3.2 Mixing

The basic step of our approach (on which we will iteratively build to provide complete mixing within a macro-block) is the application of encryption at the block level. This application is visible at the top of Figure 1.1, where the first row reports a sequence of 16 mini-blocks ($[0], \ldots, [15]$) composing

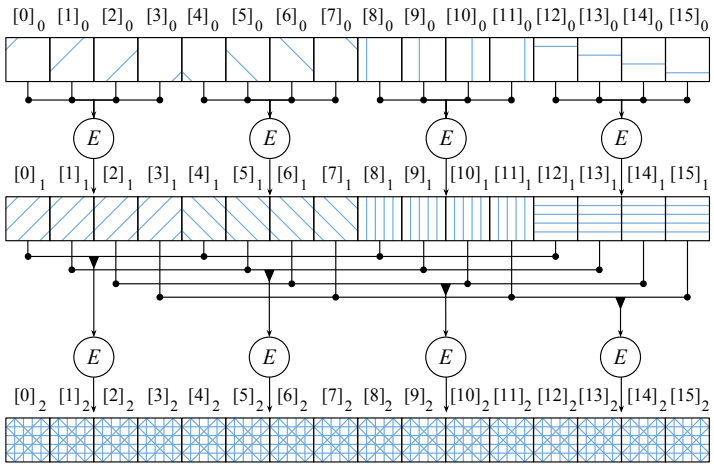



Figure 1.1: An example of mixing of 16 mini-blocks assuming $m = 4$

4 blocks. The second row is the result of block encryption on the sequence of mini-blocks. As visible from the pattern-coding in the figure, encryption provides mixing within each block so that each mini-block in the result is dependent on every mini-block in the same input block. In other words, each $[i]_1$ is dependent on every $[j]_0$ with $(i \text{ div } 4) = (j \text{ div } 4)$.

One round of block encryption provides mixing only at the level of block. With reference to our example, mixing is provided among mini-blocks $[0]_0 \ldots [3]_0$, $[4]_0 \ldots [7]_0$, $[8]_0 \ldots [11]_0$, and $[12]_0 \ldots [15]_0$, respectively. Absence of a mini-block from the result will prevent reconstruction only of the plaintext block including it, while not preventing the reconstruction of all the other blocks. For instance, with reference to our example, absence of $[0]_1$ will prevent reconstruction of the first block (mini-blocks $[0]_0, \ldots, [3]_0$) but will not prevent reconstruction of the other three blocks (mini-blocks $[4]_0, \ldots, [15]_0$). Protection at the block level is clearly not sufficient in our context, where we expect to manage resources of arbitrarily large size and would like to provide the guarantee that the lack of any individual mini-block would imply the impossibly (apart from performing a brute-force attack) of reconstructing any other mini-block of the resource. The concept of macro-block, and accurate extension of block ciphering to operate across blocks, allows us to provide mixing on an arbitrarily long sequence of bits (going much above the size of the block).

The idea is to extend mixing to the whole macro-block by the iterative application of block encryption on, at each round, blocks composed of mini-blocks that are representative (i.e., belong to the result) of different encryptions in the previous round. Before giving the general definition of

**Mix**(M)
1: **for** $i := 1, \ldots, x$ **do** /* at each round $i$ */
2: $span := m^i$ /* number of mini-blocks in a mixing */
3: $distance := m^{i-1}$ /* leg of mini-blocks input to an encryption */
4: **for** $j := 0, \ldots, b-1$ **do** /* each $j$ is an encryption */
/* identify the input to the $j$-th encryption picking, */
/* within each span, mini-blocks at leg *distance* */
5: let *block* be the concatenation of all mini-blocks $[l]$
6: s.t. ($l \mod distance) = j$ and
7: $(j \cdot m)$ div $span = l$ div $span$
8: $[[j]]_i := E(k, block)$ /* write the result as the $j$-th block in output */

Figure 1.2: Mixing within a macro-block M

our approach, let us discuss the simple example of two rounds illustrated in Figure 1.1, where $[0]_1, \ldots, [15]_1$ are the mini-blocks resulting from the first round. The second round would apply again block encryption, considering different blocks each composed of a representative of a different computation in the first round. To guarantee such a composition, we define the blocks input to the four encryption operations as composed of mini-blocks that are at distance 4 (=$m$) in the sequence, which corresponds to say that they resulted from different encryption operations in the previous round. The blocks considered for encryption would then be $\langle [0]_1[4]_1[8]_1[12]_1 \rangle$, $\langle [1]_1[5]_1[9]_1[13]_1 \rangle$,$\langle [2]_1[6]_1[10]_1[14]_1 \rangle$,$\langle [3]_1[7]_1[11]_1[15]_1 \rangle$. The result would be a sequence of 16 mini-blocks, each of which is dependent on each of the 16 original mini-blocks, that is, the result provides mixing among all 16 mini-blocks, as visible from the pattern-coding in the figure. With 16 mini-blocks, two rounds of encryption suffice for guaranteeing mixing among all of them. Providing mixing for larger sequences clearly requires more rounds. This brings us to the general formulation of our approach operating at the level of macro-block of arbitrarily large size (the example just illustrated being a macro-block of 16 mini-blocks).

To ensure the possibility of mixing, at each round, blocks composed of mini-blocks resulting from different encryption operations of the previous round, we assume a macro-block composed of a number of mini-blocks, which is the power of the number ($m$) of mini-blocks in a block. For instance, with reference to our running example where blocks are composed of 4 mini-blocks (i.e., $m$=4), macro-blocks can be composed of $4^x$ mini-blocks, with an arbitrary $x$ ($x$=2 in the example of Figure 1.1). The assumption can be equivalently stated in terms of blocks, where the number of blocks $b$ will be $4^{x-1}$. Any classical padding solution can be employed to guarantee such a requirement, if not already satisfied by the original bit sequence in input.

Providing mixing of a macro-block composed of $b$ blocks with $b$=$m^{x-1}$ requires $x$ rounds of encryption each composed of $b$ encryptions. Each round

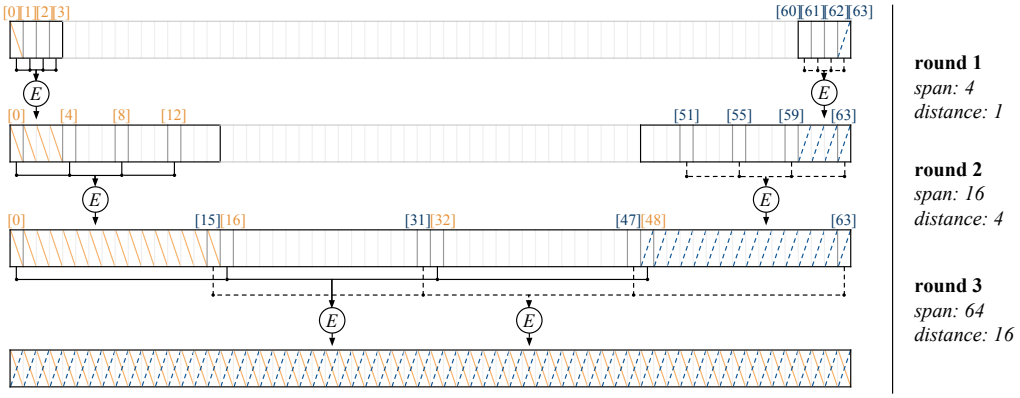



Figure 1.3: Propagation of the content of mini-blocks [0] and [63] in the mix process

allows mixing among a number *span* of mini-blocks that multiplies by $m$ at every round. At round $i$, each encryption $j$ takes as input $m$ mini-blocks that are within the same *span* (i.e., the same group of $m^i$ mini-blocks to be mixed) and at a *distance* ($m^{i-1}$). Figure 1.2 illustrates the mixing procedure. To illustrate, consider the example in Figure 1.1, where blocks are composed of 4 mini-blocks ($m$=4) and we have a macro-block of 16 mini-blocks, that is, 4 blocks ($b$=4). Mixing requires $x = 2$ rounds of encryption ($16 = 4^2$), each composed of 4 ($b$) encryptions operating on 4 ($m$) mini-blocks. At round 1, the *span* is 4 (i.e., mixing operates on chunks of 4 mini-blocks) and mini-blocks input to an encryption are taken at distance 1 within each span. At round 2, the *span* is 16 (all mini-blocks are mixed) and mini-blocks input to an encryption are taken at *distance* 4 within each *span*. Let us consider, as an another example, a macro-block composed of 64 mini-blocks (i.e., 16 blocks). Mixing requires 3 rounds. The first two rounds would work as before, with the second round producing mixing within chunks of 16 mini-blocks. The third round would then consider a *span* of all the 64 mini-blocks and mini-blocks input to an encryption would be the ones at *distance* 16.

At each round $i$, mini-blocks are mixed among chunks of $m^i$ mini-blocks, hence ensuring at round $x$, mixing of the whole macro-block composed of $m^x$ mini-blocks.

Figure 1.3 captures this concept by showing the mixing of the content of the first ([0]) and last ([63]) mini-blocks of the macro-block at the different rounds, given by the encryption to which they (and those mini-blocks mixed with them in previous rounds) are input, showing also how the two meet at the step that completes the mixing. While for simplicity the figure pictures only propagation of the content of two mini-blocks, note that at any step they

(just like other mini-blocks) actually carry along the content of all the mini-blocks with which they mixed in previous rounds. Given a macro-block M with $m^x$ mini-blocks (corresponding to $b$ blocks), the following two properties hold: *1)* a generic pair of mini-blocks $[i]$ and $[j]$ mix at round $r$ with $i$ div $m^r = j$ div $m^r$; and *2)* $x$ rounds bring complete mixing. In other words, the number of encryption rounds needed to mix a macro-block with $m \cdot b$ mini-blocks is $\log_m(m \cdot b)$.

An important feature of the mixing is that the number of bits that are passed from each block in a round to each block in the next round is equal to the size of the mini-block. This guarantees that the uncertainty introduced by the absence of a mini-block at the first round ($2^{msize}$) maps to the same level of uncertainty for each of the blocks involved in the second round, and iteratively to the next rounds, thanks to the use of AES at each iteration. This implies that a complete mixing of the macro-block requires at least $\log_m(m \cdot b)$ rounds, that is, the rounds requested by our technique.

Another crucial aspect is that the representation after each round has to be of the same size as the original macro-block. In fact, if the transformation produced a more compact representation, there would be a possibility for a user to store this compact representation and maintain access to the resource even after revocation (this is a weakness of other solutions discussed in Chapter 3). Since, in our approach, each round produces a representation that has the same macro-block size, the user has no benefit in aiming to attack one round compared to another.

When resources are extremely large (or when access to a resource involves only a portion of it) considering a whole resource as a single macro-block may be not desirable. Even if only with a logarithmic dependence, the larger the macro-block the more the encryption (and therefore decryption to retrieve the plaintext) rounds required. Also, encrypting the whole resource as a single macro-block implies its complete download at every access, when this might actually not be needed for service.

Accounting for this, we do not assume a resource to correspond to an individual macro-block, but assume instead that any resource can be partitioned into $M$ macro-blocks, which can then be mixed independently. The choice of the size of macro-blocks should take into consideration the performance requirements of both the data owner (for encryption) and of clients (for decryption), and the possible need to serve fine-grained retrieval of content. This requirement can be then efficiently accommodated independently encrypting (i.e., mixing) different portions of the resource, which can be downloaded and processed independently.

Encryption of a resource would then entail a preliminary step cutting the resource in different, equally sized, macro-blocks on which mixing operates.

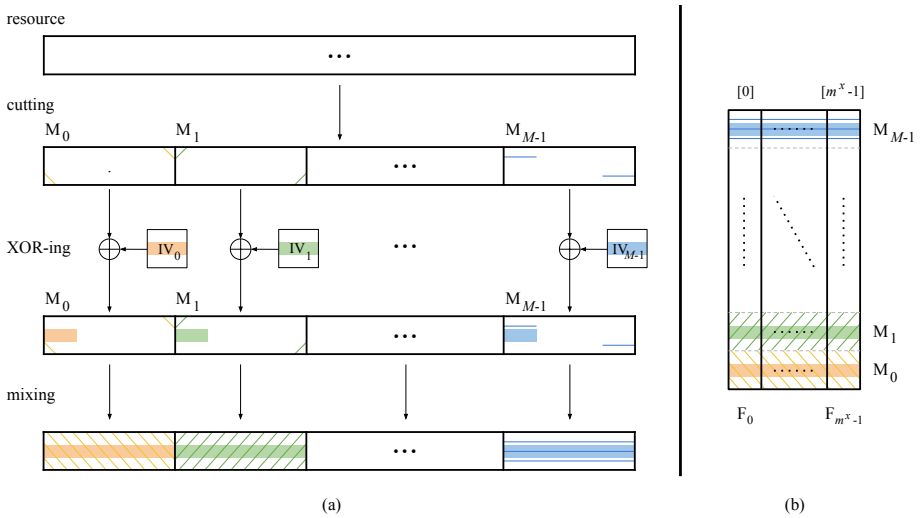



Figure 1.4: From resource to fragments

To ensure the mixed versions of macro-blocks be all different, even if with the same original content, the first block of every macro-block is XORed with an *initialization vector* ($IV$) before starting the mixing process. Since mixing guarantees that every block in a macro-block influences every other block, the adoption of a different initialization vector for each macro-block guarantees indistinguishability among their encrypted content. The different initialization vectors for the different blocks can be obtained by randomly generating a vector for the first macro-block and then incrementing it by 1 for each of the subsequent macro-blocks in the resource, in a way similar to the CTR mode [38]. Figure 1.4(a) illustrates such process.

### 1.3.3 Slicing

The starting point for introducing mixing is to ensure that each single bit in the encrypted version of a macro-block depends on every other bit of its plaintext representation, and therefore that removing any one of the bits of the encrypted macro-block would make it impossible (apart from brute-force attacks) to reconstruct any portion of the plaintext macro-block. Such a property operates at the level of macro-block. Hence, if a resource (because of size or need of efficient fine-grained access) has been partitioned into different macro-blocks, removal of a mini-block would only guarantee protection of the macro-block to which it belongs, while not preventing reconstruction of the other macro-blocks (and therefore partial reconstructions of the resource). Resource protection can be achieved if, for each macro-block

**Encrypt**
1: cut R in $M$ macro-blocks $\mathrm{M}_0, \ldots, \mathrm{M}_{M-1}$
2: apply padding to the last macro-block $\mathrm{M}_{M-1}$
3: $IV$ := randomly choose an initialization vector
4: **for** $i = 0, \ldots, M-1$ **do** /* encrypt macro-blocks */
5: $\mathrm{M}_i[[1]] := \mathrm{M}_i[[1]] \oplus IV$ /* XOR the first block with the IV */
6: **Mix**($\mathrm{M}_i$) /* encrypt the macro-block */
7: $IV := IV + 1$ /* initialization vector for the next macro-block */
8: **for** $j = 0, \ldots, m^x - 1$ **do** /* slicing */
9: $\mathrm{F}_j[i] := \mathrm{M}_i[j]$

Figure 1.5: Algorithm for encrypting a resource R

of which the resource is composed, a mini-block is removed. This observation brings to the second concept giving the name to our approach, which is *slicing*. Slicing the encrypted resource consists in defining different *fragments* such that a fragment contains a mini-block for each macro-block of the resource, no two fragments contain the same mini-block, and for every mini-block there is a fragment that contains it. To ensure all this, as well as to simplify management, we slice the resource simply putting in the same fragment the mini-blocks that occur at the same position in the different macro-blocks. Slicing and fragments are defined as follows.

**Definition 1.3.1** (Slicing and fragments)**.** *Let* R *be a resource and* $\mathrm{M}_0, \ldots, \mathrm{M}_{M-1}$ *be its (individually mixed) macro-blocks, each composed of* $(m \cdot b)$ *mini-blocks. Slicing produces* $(m \cdot b)$ *fragments for* R *where* $\mathrm{F}_i = \langle \mathrm{M}_0[i], \ldots, \mathrm{M}_{M-1}[i] \rangle$*, with* $i = 1, \ldots, (m \cdot b)$*.*

Figure 1.4(b) illustrates the slicing process and Figure 1.5 illustrates the procedure for encrypting a resource R. R is first cut into $M$ macro-blocks and an initialization vector is randomly chosen. The first block of each macro-block is then XOR-ed with the initialization vector, which is incremented by 1 for each macro-block. The macro-block is then encrypted with a mixing process (Figure 1.2). Encrypted macro-blocks are finally sliced into fragments.

## 1.4 Access management

Accessing a resource (or a macro-block in the resource, resp.) requires availability of all its fragments (its mini-blocks in all the fragments, resp.), and of the key used for encryption. Policy changes corresponding to granting access to new users can be simply enforced, as usual, by giving them the encryption key. In principle, policy changes corresponding to revocation of access would instead normally entail downloading the resource, re-encrypting it with a new key, re-uploading the resource, and distributing the new encryption key to

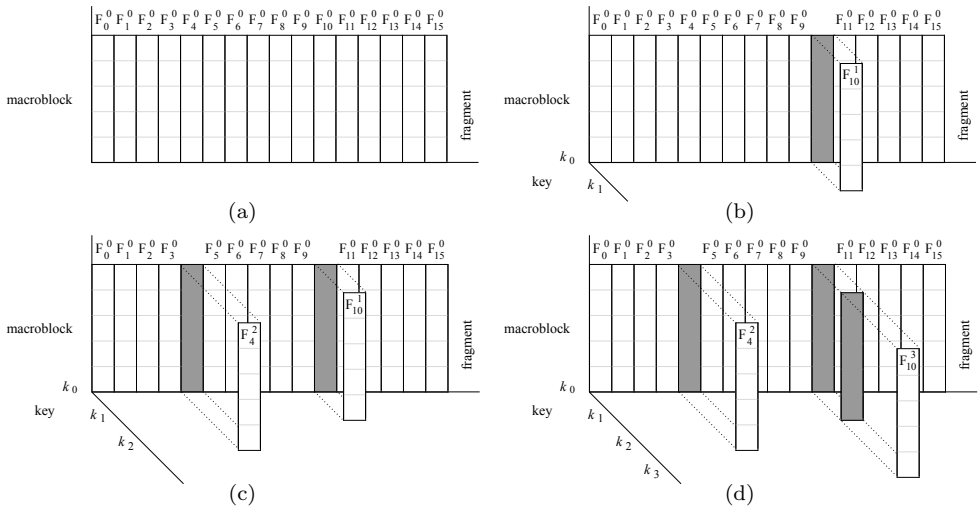



Figure 1.6: An example of fragments evolution

all the users who still hold authorizations. Our approach permits to enforce revocation of access to a resource by simply making any of its fragments unavailable to the users from whom the access is revoked. Since lack of a fragment implies lack of a mini-block for each macro-block of a resource, and lack of a mini-block prevents reconstruction of the whole macro-block, lack of a fragment equates to complete inability, for the revoked users, to reconstruct the plaintext resource or any portion of it. In other words, it equates to revocation.

Access revocations are then enforced by the data owner by randomly picking a fragment, which is then downloaded, re-encrypted with a new key (which will be made known only to users still authorized for the access), and re-uploaded at the server overwriting its previous version. While still requesting some download/re-upload, operating on a fragment clearly brings large advantages (in terms of throughput) with respect to operating on the whole resource. Revocation can be enforced on any randomly picked fragment (even if already re-written in a previous revocation) and a fresh new key is employed at every revoke operation. Figure 1.6 illustrates an example of fragments evolution due to the enforcement of a sequence of revoke operations. Figure 1.6(a) is the starting situation with the original fragments computed as illustrated in Section 1.3. Figure 1.6(b-d) is the sequence of rewriting to enforce revocations, which involve, respectively, fragment $\mathrm{F}_{10}$, re-encrypted with key $k_1$, fragment $\mathrm{F}_4$, re-encrypted with key $k_2$, and fragment $\mathrm{F}_{10}$ again, now re-encrypted with key $k_3$. In the following, we use notation $\mathrm{F}_i^j$ to denote a version of fragment $\mathrm{F}_i$ encrypted with key $k_j$, being

**Revoke**
1: randomly select a fragment $F_i$ of R /* fragment to be rewritten */
2: download $F_i^c$ from the server /* version of the fragment stored */
3: **if** $c > 0$ **then** /* $F_i^0$ has been overwritten in a revocation */
4: derive key $k_c$ /* derive $k_c$ using key regression */
5: $F_i^0 := D(k_c, F_i^c)$ /* retrieve the original version of the fragment */
6: determine the last key $k_{l-1}$ used /* it is stored in R's descriptor */
7: generate new key $k_l$
8: $F_i^l := E(k_l, F_i^0)$
9: upload $F_i^l$ overwriting $F_i^c$ /* overwrite previous version */
10: encrypt $s_l$ with the key of $acl(R)$ /* limits it to authorized users */
11: update R's descriptor /* including the new $s_l$ */

Figure 1.7: Revoke on resource R

$F_i^0$ the version of the fragment obtained through the mixing process. In the figure, the resource is represented in a three-dimensional space, with axes corresponding to fragments, macro-blocks, and keys. The re-writing of a fragment is represented by placing it in correspondence to the new key used for its encryption. The shadowing in correspondence to the previous versions of the fragments denote the fact that they are not available anymore as they are overwritten by the new versions.

Each revoke operation requires the use of a fresh new key and, due to policy changes, fragments of a resource might be encrypted with different keys. Such a situation does not cause any complication for key management, which can be conveniently and efficiently handled with a *key regression* technique [42]. Key regression is an RSA-based cryptographically strong technique (the generated keys appear as pseudorandom) allowing a data owner to generate, starting from a seed $s_0$, an unlimited sequence of symmetric keys $k_0, \ldots, k_u$, so that simple knowledge of a key $k_i$ (or the compact secret seed $s_i$ of constant size related to it) permits to efficiently derive all keys $k_j$ with $j \leq i$. Only the data owner (who knows the private key used for generation) can perform forward derivation, that is, from $k_i$, derive keys following it in the sequence (i.e., $k_z$ with $z \geq i$). Note instead that, not knowing the private key, users cannot perform forward derivation. The cost that users must pay for key derivation is small. On a single core, the computer we used for the experiments (i.e., a PC with Linux Ubuntu 16.04, 16 GB RAM DDR3, i7-4770K CPU, 256 GB SSD disk) is able to process several hundred thousand key derivations per second.

With key regression, every user authorized to access a resource just needs to know the seed corresponding to the most recent key used for it ($s_0$ if the policy has not changed, $s_3$ in the example of Figure 1.6(d)). To this end, there is no need for key distribution, rather, such a seed can be stored in the resource descriptor and protected (encrypted) with a key corresponding

**Access**
1: download R's descriptor and all its fragments
2: retrieve seed $s_l$ used for the last encryption
3: compute keys $k_0, \ldots, k_l$
4: **for** each downloaded fragment $\mathrm{F}_i^x$ **do**
5: **if** $x > 0$ **then**
6: $\mathrm{F}_i^0 := \mathrm{D}(k_x, \mathrm{F}_i^x)$ /* retrieve the original version of fragments */
7: **for** $j = 0, \ldots, M-1$ **do** /* reconstruct and decrypt macro-blocks */
8: $\mathrm{M}_j :=$ concatenation of mini-blocks $\mathrm{F}_i^0[j]$, $i = 0, \ldots, (m \cdot b) - 1$
9: decrypt $\mathrm{M}_j$

Figure 1.8: Access to resource R

to the resource's *acl* (i.e., known or derivable by all authorized users) [6, 29]. Enforcing revocation entails then, besides re-encrypting a randomly picked fragment with a fresh new key $k_i$, rewriting its corresponding seed $s_i$, encrypted with a key associated with the new *acl* of the resource. Figure 1.7 illustrates the revocation process.

To access a resource, a user then first downloads the resource descriptor, to retrieve the most recent seed $s_l$, and all the fragments. With the seed, she computes the keys necessary to decrypt fragments that have been overwritten, to retrieve their version encrypted with $k_0$. Then, she combines the mini-blocks in fragments to reconstruct the macro-blocks in the resource. She then applies mixing in decrypt mode to macro-blocks to retrieve the plaintext resource. Figure 1.8 illustrates the process to access a resource.

Note that the size of macro-blocks influences the performance of both revoke and access operations. Larger macro-blocks naturally provide greater policy update performance as they decrease policy update cost linearly, with limited impact on the efficiency of decryption, since its cost increases logarithmically.

# Chapter 2

# Dynamic Allocation for Resource Protection in Decentralized Cloud Storage

## 2.1 Decentralized Cloud Storage

While the cloud paradigm typically is based on a single service provider, the emergence of blockchains and micro-payment networks has introduced a new interest towards Decentralized Cloud Storage (DCS) networks. DCS networks rely on storage capabilities of network peers and provide an interesting emerging alternative to traditional cloud storage. The main characteristic of a DCS is that storage space is provided by users, which can be anyone from big providers to individuals, that can join the network and offer storage space, typically in exchange of some reward (micro-payment). Examples of DCS systems are Storj [84], SAFE Network Vault [60, 47], IPFS [15, 59], and Sia [82], which enable users to rent out their unused storage and bandwidth in exchange of some crypto-currency. The great advantage is that the price for storage on a DCS is a fraction of the price of a normal provider.

The use of a dynamic network of unknown (and potentially non-trusted) peers clearly introduces the problem of guaranteeing proper protection to resources, ensuring their availability and security (for both confidentiality and integrity). In fact, a DCS is a potentially unstable network, and hence continuous participation of every single node cannot be assumed. Given this, and in line with the distributed nature of DCS services, resources are sliced into many shards, with different shards allocated to different nodes of the network, with replication to guarantee availability. Reconstruction of a resource requires collecting the different shards composing it. Also, nodes participat-

ing in the DCS - which can dynamically join and leave and are anonymous - cannot be considered trusted, hence resource confidentiality needs to be protected against each of them (as well as against possible coalitions) and owners should be able to assess integrity of the shards (and hence resources).

Typically, DCS networks provide security by encrypting the resource at the owner side before slicing it, and provide availability by employing shard replication. Instead of simple replication in the allocation, several DCS networks leverage the dynamic application of erasure codes (e.g., Reed-Solomon). With Reed-Solomon, if a node participating in the service becomes unavailable, its shard is dynamically re-allocated to another node. The advantage of Reed-Solomon with respect to simple replication is that it provides reliability guarantees at a fraction of the storage overhead. However, it has two main drawbacks. First, it is a fixed-rate encoding technique and therefore computing the shard to be re-allocated requires reconstructing the complete resource. Second, if the old node originally storing a (then re-allocated) shard comes back online, more replicas than actually needed would be available for the same shard and the economic cost brought by the involvement of more nodes does not bring a clear advantage for availability. Also, the use of simple encryption for providing confidentiality has some limitations since, although it is true that the unavailability of all shards composing a resource prevents its complete reconstruction, still a subset of the shards may enable reconstruction of a portion of the resource to users knowing the encryption key. For instance, a owner losing control on the encryption key may ask nodes in the DCS to delete the resource, as it would otherwise remain exposed. However, nodes that do not respect the request of deleting their shards may then have access to a portion of the resource.

The contribution of this chapter is twofold. First, we combine AONT, in contrast to simple encryption for protecting resources, and fountain codes, in contrast to traditional erasure codes like Reed-Solomon for computing shards to be distributed to nodes in a DCS. The combined adoption of AONT and fountain codes provides better security and availability guarantees, and lower performance overhead than current solutions. Second, we provide a model for dynamically managing the computation and allocation of shards, avoiding an immediate reaction to a node leaving, if availability guaranteed by the remaining nodes is considered still acceptable. Our approach avoids potentially excessive generation of shards which may turn out to be unnecessary when - as it is often the case - nodes unavailability is only temporary. Avoidance of excessive generation provides advantages not only for performance (as it avoids generating new shards and involvement of new nodes) but also for security (excessively increasing the number of shards in the system may cause a higher vulnerability to malicious coalitions).

## 2.2 Basic concepts

The two basic building blocks enabling the development of our solution are the adoption, at the client side, of *all-or-nothing transform* (AONT) and of *fountain codes*.

**AONT**. AONT [71] is an encryption mode that transforms a plaintext resource into a ciphertext where every bit of the output has strong interdependence on all the bits of the input (e.g., [12, 56, 57]). In other words, AONT ensures that even knowing the encryption key, a resource or a portion of it cannot be reconstructed even when even only a single block is missing. AONT then provides better security guarantees than encryption. An Example of AONT is Mix&Slice, i.e., the technique illustrated in Chapter 1

**Fountain codes**. Fountain codes are a class of erasure codes preventing that the loss of one of the transmitted or stored blocks of a resource causes a data loss. Given a resource $r$, partitioned into $f$ different *fragments*, an erasure code generates a set of $s>f$ *encoded shards* that depend on the resource content and support the reconstruction of $r$ through the combination of a subset of the encoded shards. Fountain codes, unlike other erasure codes (e.g., Reed-Solomon [67]), offer probabilistic reconstruction guarantees, meaning that with a probability $p < 1$, $f$ of the $s$ shards are sufficient for reconstructing $r$. The reconstruction probability $p$ exponentially increases by retrieving additional shards. Although probabilistic, fountain codes have two main characteristics that, as we will discuss in the next section, allow us to profitably use them in the DCS context. First, they are *rateless*, that is, using these codes it is possible to create a new (i.e., different from each other) shard on the fly and therefore the number $s$ of encoded shards is not fixed a priori. Independently from the number $s$ of shards, any subset of (at least) $f$ shards can be used to reconstruct $r$. Second, each shard depends on a subset of (and not on all) the $f$ original fragments of the resource and then only a subset of the original fragments are needed for generating a new shard.

## 2.3 Encoding and allocation strategy

We consider a data owner interested in storing her resources in a DCS network. For simplicity, we refer the discussion to a single resource $r$, since the same approach can be applied to all the resources moved to the DCS.

Given a resources $r$ we assume the data owner to first encrypt $r$ using AONT, to protect confidentiality, and then encode the resulting ciphertext

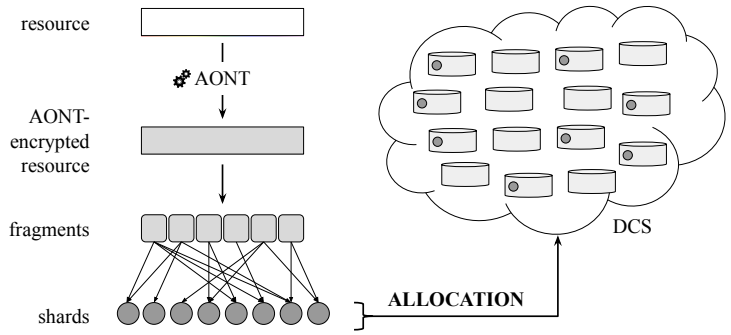
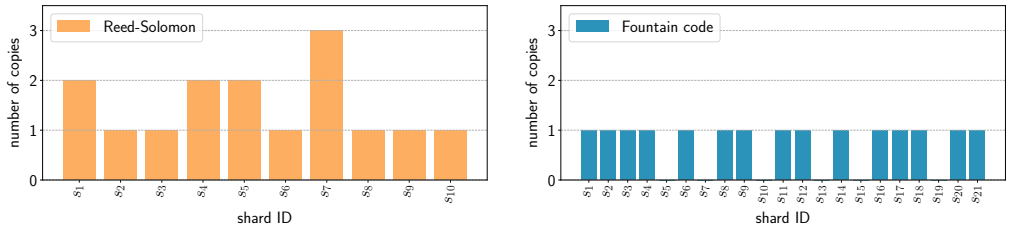



Figure 2.1: Reference scenario

using fountain codes, to provide availability. As illustrated in Figure 2.1, the ciphertext is organized in $f$ original fragments and encoded into a set $\mathcal{S} = \{\mathtt{s}_1, \ldots, \mathtt{s}_s\}$ of $s$ shards allocated to $s$ randomly chosen nodes $\mathcal{N} = \{\mathtt{n}_1, \ldots, \mathtt{n}_s\}$ (i.e., each shard is allocated to a different node). Whenever a user is interested in retrieving the resource content, she contacts an arbitrary subset of $f$ nodes and downloads their shards, which are then combined to reconstruct the resource.

The reason for using AONT, in contrast to encryption, is its ability to prevent even partial reconstruction of the plaintext resource. This property is crucial in the considered scenario, since fountain codes enable the reconstruction of a fragment (i.e., a chunk of the ciphertext) starting from a small subset of shards (i.e., less than $f$ shards). The adoption of AONT then prevents any (small) coalition of malicious nodes to partially reconstruct the plaintext resource content, even if they can reconstruct a fragment from the shards they store.

The choice of using fountain codes, in contrast to simple replication or traditional erasure codes (e.g., Reed-Solomon), is due to their high flexibility in dealing with the intrinsically dynamic behavior of DCS . Since fountain codes are rateless, if a node $\mathtt{n}_\mathtt{i}$ leaves the DCS, the data owner does not need (as it would happen using Reed-Solomon) to reconstruct its shard $\mathtt{s}_\mathtt{i}$ and re-allocate it to a different node to guarantee the same resource availability. Indeed, it is sufficient to generate a new shard $\mathtt{s}_{\mathtt{s}+1}$ (different from $\mathtt{s}_1, \ldots, \mathtt{s}_\mathtt{s}$) and allocate it to a new node $\mathtt{n}_{\mathtt{s}+1}$. After the generation and allocation of $\mathtt{s}_{\mathtt{s}+1}$, any user can still reconstruct the resource content by downloading any subset of $f$ shards from the resulting set $(\mathcal{S} \backslash \{\mathtt{s}_\mathtt{i}\}) \cup \{\mathtt{s}_{\mathtt{s}+1}\}$ of $s$ shards. The generation of a new shard, in contrast to the replication of the unavailable one, provides higher resource availability. Indeed, every shard is unique and equally contributes to the reconstruction of the resource. Hence, when the previously unavailable node $\mathtt{n}_\mathtt{i}$ comes back online, the increased economic




(a) Reed-Solomon (b) Fountain codes

Figure 2.2: An example of shard generation/replication using Reed-Solomon (a) and fountain codes (b) in a dynamic scenario

cost due to the involvement of a larger number $(s+1)$ of nodes comes with an actual advantage in terms of higher availability. In fact, there will be $s+1$ (instead of $s$) different shards stored at $s+1$ different nodes that can all be used for resource reconstruction. Note that the generation of a new shard causes a limited overhead since it implies the download of a subset of the shards in $\mathcal{S}$, without the need to reconstruct $r$ (as required by other erasure codes).

As discussed, the rateless characteristic of fountain codes allows the adaptive adjustment of the number of shards available for reconstructing a resource, thus impacting the availability and security guarantees. We use this characteristic (see Section 2.4) in such a way that, when the availability guarantees go below a given threshold, the owner can generate a new shard. Analogously, when the risk of confidentiality exposure is above a given threshold due to the presence of a high number of shards, the data owner can re-encrypt the resource and generate a new set of $s$ shards.

Figure 2.2 illustrates an example that compares the set of shards in a DCS when using Reed-Solomon and fountain codes. Here, we assume that the data owner partitions the resource in $f$=7 fragments and encodes it in $s$=10 shards and that nodes where the shards are stored leave and then possibly re-join the network. Since Reed-Solomon reacts to a node failure replicating the shard of the failed node, the set $\mathcal{S}$ of shards representing $r$ never changes but the number of copies grows (e.g., $\mathtt{s}_7$ has three copies). Fountain codes do not cause any replication, but a new shard is generated at each failure. Note that the two techniques imply the same economic cost for the owner, since each failure causes the allocation of a (new or duplicated) shard to a node. In the considered example, the owner pays for 15 shards in both scenarios.

## 2.4 Availability and security guarantees

Node failure has an impact on both resource availability and confidentiality. Indeed, when one of the nodes $\mathtt{n_i}$ fails, the encoded shard $\mathtt{s_i}$ it stores cannot be used to reconstruct the resource content. Hence, the user needs to retrieve $f$ out of $s-1$, in contrast to $s$, shards. When $\mathtt{n_i}$ re-joins the DCS, it still stores $\mathtt{s_i}$ and this could provide a positive effect on resource availability, since the shard at the node could be used to reconstruct the resource. However, node $\mathtt{n_i}$ re-joining the DCS could have a negative impact on security, since $\mathtt{n_i}$ could exploit its knowledge of $\mathtt{s_i}$ and collude with other $f-1$ nodes to reconstruct $r$ (or prevent its deletion). Similarly, the generation and allocation of a new shard $\mathtt{s_{s+1}}$ improves resource availability, but also naturally reduces security since there is a higher number of nodes that could possibly collude.

In this section, we analyze the advantages provided by fountain codes on availability guarantees, by studying the probability $P_u$ that a resource becomes unavailable as a consequence of nodes leaving and re-joining the network. We also evaluate the risks that the adoption of our solution causes in terms of security, by studying the probability $P_c$ that a coalition of malicious nodes has enough shards to compromise resource security. These probabilities depend on the probability $p_u$ that a node fails, and on the probability $p_c$ that a node is malicious and interested in colluding with other nodes to breach the confidentiality of the resource. For simplicity, we assume $p_u$ and $p_c$ to be the same for all the nodes.

### 2.4.1 Availability guarantees

When using $s$ shards to encode a resource $r$ split into $f$ original fragments, $r$ becomes unavailable when more than $s-f$ nodes fail (i.e., when less than $f$ shards can be accessed). The probability of such an event to happen is $P_u{=}\sum_{i=s-f+1}^{s} \binom{s}{i} {p_u}^i (1-p_u)^{s-i}$. Probability $P_u$ increases when one of the nodes fails (or leaves the DCS) since the shard stored in it is no longer available, while it decreases any time a new shard is generated or a failed node re-joins the network.

Even if, in principle, the owner should react every time a node $\mathtt{n_i}$ fails by generating a new shard, this practice is expensive and may not even be necessary (e.g., if $\mathtt{n_i}$ re-joins the network in a few hours). The increase in $P_u$ caused by the failure of a node may not be critical in a scenario where nodes dynamically leave and re-join the system frequently. Indeed, the reduction in $P_u$ may be temporary and may not considerably affect the ability of the user to reconstruct $r$. To properly take into consideration these aspects, we

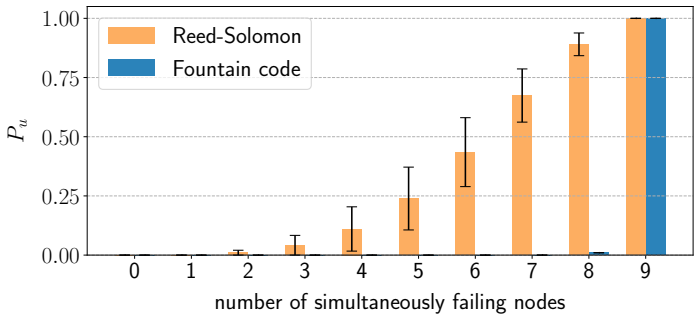



Figure 2.3: Probability that a resource becomes unavailable using Reed-Solomon and fountain codes, assuming $p_u$=0.015, $P_u^{min}$=$10^{-12}$, $P_u^{max}$=$10^{-5}$, $f$=7 fragments, and $s$ in [10,15]

propose a solution where the data owner takes corrective actions (i.e., create a new shard) only when resource availability is considered at risk.

Our solution is based on the definition of two thresholds for $P_u$, $P_u^{min}$ and $P_u^{max}$, identifying the range of values considered acceptable by the owner for guaranteeing the availability of $r$. Intuitively, these thresholds influence the maximum and minimum number of available shards in the system and represent:

- $P_u^{max}$: the maximum probability of failure that the owner can tolerate, which corresponds to the minimum number of shards ($\geq f$) the owner considers desirable to keep resource unavailability under control;
- $P_u^{min}$: the minimum probability of failure fixed by the data owner based on her economic availability, which corresponds to the maximum number of shards the owner can afford.

The data owner does not react every time a node leaves or re-joins the DCS, but only if this event causes $P_u$ to become higher than $P_u^{max}$ or smaller than $P_u^{min}$. If $P_u$ exceeds $P_u^{max}$, the data owner generates a new shard and allocates it to a new node. If $P_u$ goes below $P_u^{min}$, the data owner can terminate the contract with one of the nodes in the system (e.g., with the one that has been off-line the most) and stop paying for its services.

Consider, as an example, a resource partitioned in $f$=7 original fragments and encoded in $s$=10 shards allocated at nodes with probability $p_u$=0.015 of failure. Figure 2.3 compares the probability $P_u$ of unavailability of a resource $r$ when using Reed-Solomon and fountain codes, assuming $P_u^{min}$=$10^{-12}$ and $P_u^{max}$=$10^{-5}$, and varying the number of nodes that the data owner identifies

as unavailable between 0 and 9. The error bars in the figure represent the standard deviation. Note that $P_u$ is initially the same for Reed-Solomon and fountain codes, and evolves in the same manner when nodes leave the DCS. The evolution of $P_u$ when a node re-joins the DCS is instead considerably different: with Reed-Solomon it causes duplication of a shard, with fountain codes it implies the availability of an additional (different) shard. As visible from the figure, the probability that $r$ becomes unavailable is higher using Reed-Solomon than using fountain codes. In fact, the nodes storing shards available in a single copy (e.g., $\mathtt{s}_2$ in Figure 2.2(a)) play a critical role in resource reconstruction. Indeed, when failing, they cannot be immediately substituted. With fountain codes, all nodes are equally critical since they all store different shards that can be interchangeably used to reconstruct resource $r$.

The top chart in Figure 2.4 illustrates an example of evolution of $P_u$, assuming the adoption of fountain codes, considering the corrective actions taken by the owner. The value of $P_u$ grows every time a node leaves and decreases every time a node re-joins the network. As long as $P_u$ is between $P_u^{min}$ and $P_u^{max}$ (the two red dashed lines in the figure), the data owner does not react to node leave and re-join events. In the example, we set $P_u^{min}$ and $P_u^{max}$ in such a way to tolerate the leave and re-join of one node at a time. When $P_u$ reaches $P_u^{max}$ as a consequence of the failure of a node (red triangles 1, 2, and 4 in the figure, indicating unavailability of two nodes), the owner generates and allocates a new shard. The generation of a new shard reduces $P_u$ to a value below $P_u^{max}$ (green circle 1, 2, and 4 in the figure). When $P_u$ reaches $P_u^{min}$ as a consequence of node re-join (red triangle 3 in the figure, indicating re-join of all the three nodes that failed), the owner closes the contract with one of the nodes and $P_u$ returns above the threshold (green circle 3 in the figure).

In the next section, we will illustrate that $P_u$ decreases not only when the data owner creates a new shard, but also when the resource is re-encrypted.

### 2.4.2 Security guarantees against malicious coalitions

Since $f$ shards are sufficient to reconstruct $r$, any coalition of at least $f$ malicious nodes can reconstruct the encrypted content of the resource (and its plaintext representation if the key is exposed) and/or prevent its deletion by retaining their local copy of the shard. The probability that $f$ (or more) nodes collude is $P_c{=}\sum_{i=f}^{s} \binom{s}{i} {p_c}^i (1-p_c)^{s-i}$. The probability $P_c$ that $f$ nodes collude increases whenever the data owner generates a new shard and allocates it to a new node. On the contrary, $P_c$ never decreases. Indeed, we cannot assume that nodes leaving the system will not re-join in the future and that they

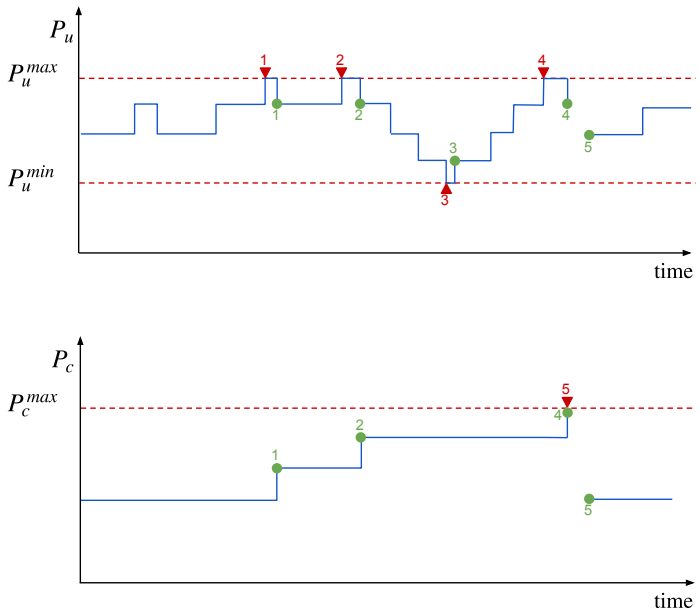



Figure 2.4: An example of evolution of $P_u$ (top chart) and $P_c$ (bottom chart) as a consequence of nodes leaving and re-joining the DCS and of actions taken by the data owner

do not have a copy of the shard initially allocated to them, even in case the data owner closed the contract. A malicious node can keep a copy of the data on purpose, to prevent resource deletion and possibly sell the shard to non-authorized users.

The owner can reduce $P_c$ only by re-encrypting $r$ with a new key. This process is however quite expensive as it requires to locally reconstruct $r$ (downloading $f$ shards), decrypt it, re-encrypt its plaintext content with a new encryption key, encode the resulting ciphertext, and distribute the new set of $s$ shards to $s$ nodes. To limit such overhead, our solution is based on the definition of a threshold $P_c^{max}$ for $P_c$, representing the maximum probability that the owner can tolerate that $r$ is exposed. When, as a consequence of the generation of a shard, $P_c$ exceeds $P_c^{max}$, the owner re-encrypts $r$. Clearly, shards generated before resource re-encryption cannot be used together with shards generated after re-encryption for resource reconstruction, hence nullifying possible misbehaviors by nodes whose contract has been closed before re-encryption.

The bottom chart in Figure 2.4 illustrates the evolution of $P_c$ due to corrective actions taken by the owner for keeping $P_u$ below $P_u^{max}$ (i.e., the generation of new shards). As long as $P_c$ remains below threshold $P_c^{max}$, no

| Shard size | Reed-Solomon | RaptorQ |
|---|---|---|
| **100 kB** | 0.680s ($\sigma = 0.220$) | 0.487s ($\sigma = 0.167$) |
| **1 MB** | 1.071s ($\sigma = 0.306$) | 0.758s ($\sigma = 0.234$) |
| **10 MB** | 5.082s ($\sigma = 3.232$) | 2.552s ($\sigma = 0.962$) |

Figure 2.5: Average time required to download enough shards from the network to reconstruct a resource

corrective action is taken (green circles 1 and 2 in the figure). In the example we set $P_c^{max}$ to tolerate the generation of two shards. When the generation of a new shard causes $P_c$ to reach $P_c^{max}$ (red triangle 5 and green circle 4 in the figure), the owner re-encrypts $r$. Resource re-encryption resets both $P_u$ and $P_c$ to their initial value (green circle 5 in the figure). The lower is $P_u^{max}$, the more frequently shards will be generated and the more frequently the data owner will need to re-encrypt $r$.

## 2.5 Implementation and experiments

To assess the benefit of using fountain codes in contrast to Reed-Solomon used by real world DCS networks, we implemented our proposal on top of Storj [84]. Among the existing DCS, we selected Storj because it is one of the leaders in the DCS scenario, and its open-source implementation easily permits the integration of our solution. Storj relies on Reed-Solomon to guarantee resource availability, and applies encoding and decoding at the owner and user side, respectively. We then modified the Storj client to adopt fountain codes instead of Reed-Solomon. Specifically, we used RaptorQ code (specified in IETF RFC 6330 [63]), which is the latest evolution of Rapid Tornado (Raptor) codes [75]. RaptorQ is among the most effective fountain codes available to date, since the probability of not being able to reconstruct a resource when accessing $f+\varepsilon$ shards is negligible, $0.01^{\varepsilon+1}$ (i.e., 1% accessing $f$ shards, 0.01% accessing $f+1$ shards).

To analyze the performance of our solution, we set parameters $f$=7 and $s$=10 and considered shards of size 100kB, 1MB, and 10MB (i.e., resources of size 700kB, 7MB, and 70MB, respectively). In our experiments, we considered $p_u$=0.015, $P_u^{min}$=$10^{-12}$, $P_u^{max}$=$10^{-5}$, $p_c$=0.025, and $P_c^{max}$=$10^{-6}$. These parameter values support the generation of 22 shards before requiring resource re-encryption. The client used for our experimental evaluation is an Intel i7 4770K with 16 GB DDR3 RAM, running Ubuntu 18.04, connected to a Fast Ethernet network, thus with enough bandwidth to download all the required ($f$=7) shards in parallel. In our tests, we contacted 70 nodes. The histogram bars in Figure 2.6 illustrate the time necessary to download a shard

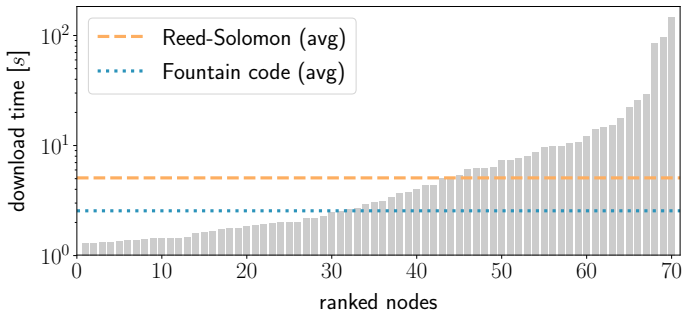



Figure 2.6: Download time of a 10MB shard from each node that has been contacted in the test (in increasing order of time)

of 10MB from each of these nodes. The nodes are reported on the $x$-axis in increasing download time. In the experiments, we focused on download time as the use of AONT causes a negligible overhead, since the bottleneck is represented by the network (e.g., the solution in [12] has 2.5 GB/s throughput, which is orders of magnitude higher than fast network connections).

If the user interested in downloading the resource knows exactly the response time of each node in the DCS, she can choose the optimal pool of nodes, and minimize the time for retrieving the shards necessary to reconstruct $r$. When using RaptorQ, the best strategy consists in contacting the $f$ fastest nodes, since any subset of $f$ shards enables resource reconstruction. Hence, the download time is given by the access time of the $f$-th node in the ranking of the $s$ nodes storing a shard of the resource. When using Reed-Solomon the best strategy consists in contacting the $f$ fastest nodes that store $f$ different shards, which might not be the first $f$ nodes in the ranking in case of duplicated shards. The download time then is given by the access time of the slowest contacted node.

Since usually clients do not know the response times of the nodes in the DCS, the best strategy consists in contacting all the $s$ nodes in parallel, and stop ongoing downloads when the downloaded shards enable resource reconstruction (i.e., after the first $f$ shards have been downloaded for RaptorQ, after $f$ different shards have been downloaded with Reed-Solomon). Figure 2.5 reports the average over 100 runs of the time (and standard deviation $\sigma$) for downloading the shards necessary to reconstruct a resource obtained when using Reed-Solomon and RaptorQ. As expected, RaptorQ provides better response times, with a reduction between 28.32% (with shards of 100kB) and 49.78% (with shards of 10MB), than Reed-Solomon. This difference is due to the fact that the download time is a function of both the download

throughput and the latency of nodes. For smaller shards, latency is more relevant than for larger shards, where node bandwidth represents the factor contributing most to download times.

Figure 2.6 reports the average time necessary to reconstruct the resource in case of shards of 10MB when using RaptorQ (blue line) and Reed-Solomon (yellow line). As illustrated in the figure, RaptorQ requires to acquire shards from ∼45-th percentile of the nodes (30 out of 70 nodes), while for Reed-Solomon requires to acquire shards from ∼65-th percentile (45 out of 70 nodes), causing a delay in resource reconstruction. We conclude that the adoption of fountain codes provides a general advantage, thanks to the possibility for the final user to rely on faster nodes for resource reconstruction, while providing security guarantees.

# Chapter 3

# Related Work

The idea of making the extraction of the information content of an encrypted resource dependent on the availability of the complete resource has been first explored by Rivest [71], who proposed the *all-or-nothing transform* (AONT). The AONT requires that the extraction of a resource where $n$ bits of its transformed form are missing should require to attempt all the possible $2^n$ combinations. The AONT can be followed by encryption to produce an *all-or-nothing encryption schema*. In [71], the author proposes the *package transform*, which realizes an AONT by applying a CTR mode using a random key $k$. The ciphertext is then suffixed with the used key $k$ XOR-ed with a hash of all the previous encrypted message blocks. In this way, a modification on the encrypted message limits the ability to derive the encryption key. This technique works under the assumption that the user who wants to decrypt the resource has never accessed the key before, but fails in a scenario where the user had previously accessed the key and now the access must be prevented (i.e., revocation of privileges on encrypted files). The user, in fact, could have stored key $k$ and so she would be able (depending on the encryption mode used) to partially retrieve the plaintext. Key $k$ can be seen as a digest: it is compact and its storage allows a receiver to access the majority of the file, even if one of the blocks was destroyed.

The problem of providing availability guarantees to data stored in DCS networks has been considered by real-world systems. Storj [84], Sia [82], SAFE Network [47] adopt techniques based on Reed-Solomon, and Enigma [4] relies on fountain codes. Although these DCS networks also pay attention to security, they do not aim at protecting data against coalitions of malicious nodes. Moreover, the network immediately reacts to any configuration change. Our proposal is orthogonal to these solutions, and can be easily integrated in real world DCS networks to balance availability and security guarantees while limiting the data owner's intervention.

The problem of balancing availability and security in DCS networks has been recently addressed in [13]. This proposal is based on the combined adoption of AONT and data replication, and introduces two allocation strategies of shards to nodes. The analysis of the proposed allocation strategies illustrates how they can be tuned to balance availability and security. Unlike our approach, this proposal considers a static scenario and adopts replication.

The combined adoption of AONT and error-correcting code techniques has been recently explored to the aim of protecting outsourced data against possibly curious storage providers and offering high performance (e.g., [14, 68]). *AONT-RS* [68] combines Rivest's AONT [71] with Reed-Solomon, while *AONT-LT* [14] uses Luby Transform code (which is a class of fountain codes [21, 75]) instead of Reed-Solomon. Although these proposals present similarity with our approach, they are specifically focused on static scenarios, where the set of nodes is fixed and nodes are not expected to frequently leave/join the network. Indeed, error-correcting codes are used to enhance performance, in contrast to provide availability guarantees. These solutions are then suited to cloud-of-clouds but not to DCS scenarios.

# Chapter 4

# Conclusions

In this part of thesis we presented the *all-or-nothing transforms* (AONT), in particular we introduced Mix&Slice, an approach for efficiently enforcing access revocation on encrypted resources stored at external providers. Mix&Slice enables data owners to effectively revoke access by simply overwriting a small portion of the (potentially large) resource and is resilient against attacks by users locally maintaining copies of previously-used keys. We then presented an approach aimed at balancing availability and security of resources stored in a DCS, where nodes naturally leave and re-join the system. The proposed solution combines AONT encryption and fountain codes. Our approach is based on the definition of thresholds for both availability and security guarantees, and on the idea that the data owner intervenes only when the thresholds are violated. Our experimental evaluation demonstrates that our proposal considerably reduces the time required to reconstruct resources in a DCS, while limiting the exposure to coalitions of malicious nodes.

# Part III

# Access Control for Data Aggregations

# Chapter 1

# Declarative Access Control for Aggregations of Multiple Ownership Data

## 1.1 Introduction

In the media and business discussions, data are often regarded as "the new oil" [78]: companies can take advantage of the availability of information to obtain insights to improve their business. From predictive maintenance to marketing, business applications make use of analytics, machine learning (ML) and artificial intelligence (AI) to deliver their functionalities, often using public data available on the Internet. Data may be produced by the same subject/entity designing the application or by external sources, from publicly available or commercial datasets. In the current practice, it is assumed to have full control on the input datasets. However, fragmentation in data ownership and/or in usage terms is not an uncommon situation. It is the case, for example, of personal information processing as regulated by GDPR [40]: individuals give their data to a company under an agreement that explicitly lists and describes all data computation purposes and procedures. Data ownership remains with each individual that has right to control her data. Nevertheless, the enforcement of these principles is generally achieved through legal processes and means, thus individuals must trust the receiving company for the compliant processing of their data. Individuals do not have control on how their information are actually used, also with respect to the other pieces of information processed together. Aggregating data of multiple individuals may reveal illegitimate processing (like in the case of Cambridge Analytica [20]) that might be hindered if controls on data aggregations can be available.

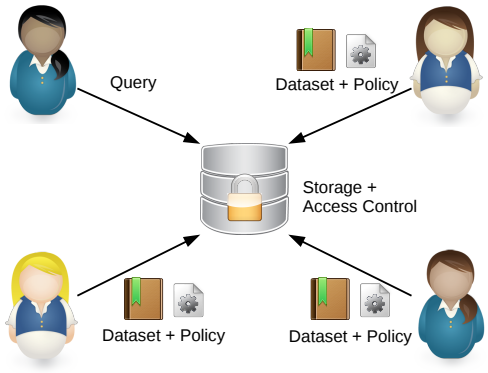



Figure 1.1: Multiple individuals share their information with an entity, trusted for making them available to others and for enforcing their own usage policy

We may observe a pattern, where multiple data owners share data with a number of recipients, trusted to comply with directives and agreements in consuming their data. We believe that trust assumptions on data processing are due to the lack of fine-grained access control mechanisms, able to make decisions on one's information taking into account attributes of the other pieces of information used in the same operation. Our proposal aims at enabling data owners to express machine-enforceable security policies to regulate the usage of their datasets when aggregated and/or analyzed together with other datasets. Data owners' policies are expected to capture Attribute-Based Access Control (ABAC) model constraints, one of the most flexible and powerful access control models. We consider a scenario where multiple data owners share their data with access policies by means of an entity, trusted for the enforcement of their policies when receiving requests for controlled datasets, as depicted in Figure 1.1. To exemplify our idea, we will consider the case where a data scientist uses a ML framework to analyze datasets of multiple owners with different policies. To materialize such ABAC policies, we designed our solution by extending a well-known attribute based access control standard, XACML [70] to benefit from its conceptual framework, popularity, validation and existing implementations. To this scope, we extend the XACML policy language in order to express conditions that regulate simultaneous access to multiple resources. We adopt some performance-effective techniques for XACML evaluation and we supplement the XACML evaluation context with the ability to analyze aggregations of datasets by means of its integration with a ML library.

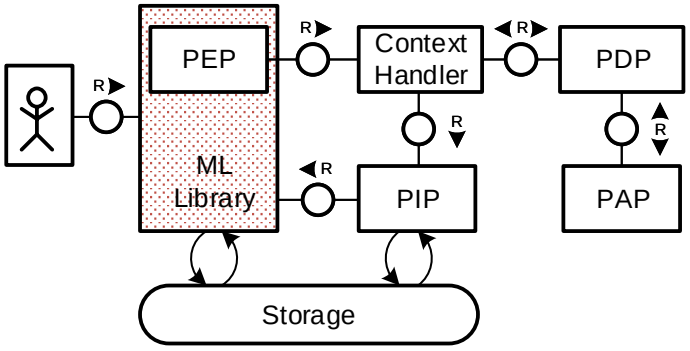



Figure 1.2: The proposed system architecture (including "ML Library" and "Storage" components integrated with those of OASIS XACMLv3 standard)

## 1.2 Basic concepts

In our work, we consider XACML [70] as our reference policy language. XACML is a standard based on XML, published by OASIS that defines a language for access control policies. XACML is a flexible, well-known, widespread, and easy-to-use language. An XACML policy declares if an action, performed by a subject, is permitted or denied on a particular resource (under specific constraints). XACML can be used to materialize constraints coming from multiple (if not any) access control models, such as ABAC [51].

Each XACML policy is composed by a *Target* and one or more *Rules*. The Target defines a combination of attributes related to subjects, actions and resources that determines the applicability of the policy. Also a Rule contains a Target, besides a *Condition*, that indicates whether the *RuleEffect*, i.e., an attribute of the Rule element that consists in either a *Permit* or a *Deny*, can be applied to the request or not. A policy can possibly declare also *Obligations* and *Advices*, that introduce further actions, respectively mandatory or optional, to be accomplished at the response time. Policies may also be aggregated in a *PolicySet*, a collection of *Policy* elements.

In order to be enforced in a system, XACML requires the presence of the following components. **PDP (Policy Decision Point)**: the component that evaluates requests against the applicable policies and makes the access control decisions. **PEP (Policy Enforcement Point)**: the component that receives the request from the user and returns a response, even if it delegates the access control decision to the PDP. Moreover, the PEP is the component in charge of executing Obligations and Advices. **PAP (Pol-**

**icy Administration Point)**: the component that manages the policies and makes them available when needed for evaluation. **PIP (Policy Information Point)**: the component that retrieves further attributes related to the subject, action, resource and environment of the request from their respective repositories. **Context Handler**: the component that is responsible for maintaining a set of attributes, coming from PEP request and PIP, necessary for PDP evaluation.

An example of our reference architecture is shown in Figure 1.2, using the FMC notation [58]. We enforce the use of the PEP, PAP, PDP, Context Handler and PIP as described above, and also rely on a *Storage* module, i.e., a component in charge of storing our datasets, and a *ML Library* module, i.e., a library for performing any operation of aggregation that consists in a ML operation.

As stated above, the PEP is the component that intercepts the user's intent and produces an authorization request to be evaluated; it also materializes the decision made by the PDP to the user. Given a set of subjects $S$, a set of actions $A$, a set of resources $R$, and an environment $e$, we define an authorization request as a tuple $Req = \{s, a, r, e\}$ where $s \in S, a \in A, r \in R$. In this work, we do not focus on the environment, since it does not affect our proposal and its presence is irrelevant for pursuing our goals. Indeed, the environment provides further attributes not related to subjects, actions, and resources, but related to the environment itself (e.g., request/evaluation time, web domain, etc.). Therefore, for simplicity, in the remainder of this paper we will not consider the environment, and we will define a request as a tuple $Req = \{s, a, r\}$. Whenever a PDP is asked to make a decision, it relies on the information made available by the Context Handler, i.e., the *Context*: it consists of a container for all the attributes extracted from the PEP-prepared request and complemented with additional necessary attributes by the PIP. Then, the PDP matches the context with the attributes derived from the policy provided by the PAP.

To do so, the PDP first compares $\{s, a, r\}$ with the policy Target. If they match, then it checks if the Condition is true. If the Condition is true (or there is no Condition), then the effect of that rule (i.e., Permit or Deny) is returned to the PEP, and thereupon returned to the user. Nevertheless, a policy may contain more than one rule, and more than one rule may match the request target and provide a true condition. Therefore, in case more than one rule (and more than one policy, if these rules are derived from different policies) is applicable, a rule (resp. policy) combining algorithm regulates the behavior of the PDP.

### 1.2.1 Combining Algorithms

According to the current XACML specification [69], seven rule-combining algorithms are defined, as follows.

- **First applicable**: consider only the first rule (in order of appearance in the policy) that is applicable to the current request and return its effect.
- **Permit overrides** (resp. **Deny overrides**): in case more than one rule is applicable, return Permit (resp. Deny) if available as at least one of the applicable rule effects.
- **Ordered Permit overrides** (resp. **ordered Deny overrides**): same as Permit overrides (resp. Deny overrides), but applicable rules are evaluated in order.
- **Permit unless Deny** (resp. **Deny unless Permit**): return Permit unless one of the applicable rules returns a Deny (return Deny unless one of the applicable rules returns a Permit, respectively).

These algorithms are applicable also to policies, in the same way they work for rules (in this case they are called policy-combining algorithms). In addition to the seven cases listed above, there is an eight combining algorithm valid only for policies, i.e., **only one applicable**, which states that only one policy must be applicable to the request, otherwise an Indeterminate result is returned.

### 1.2.2 XACML Profile Multi

Another aspect of XACML is relevant for our work. The current XACML specification provides a profile in case a request requires more than one access control decision, e.g., when the request contains repeated attribute categories or hierarchical resources. This profile is the so-called *XACML Multiple Decision Profile* (informally known as *"Multi"*) [69], and an example of request related to this profile is an access request to more than a single resource. We define a request of this kind a *request-Multi*, in accordance to the XACML profile it refers to. Conforming to the current specification, the evaluation of a *request-Multi* is done by dividing this request for multiple repeated attributes into many requests for multiple but not repeated attributes, i.e., a request-Multi is disassembled into many single requests, and each of these requests is evaluated individually. According to this approach, each Context (derived from such individual requests) acts as a sandbox such that the PDP

owns least information for every access control decision. In case of request-Multi, either a combined decision or an individual decision is returned to the user, as determined by the requester through an attribute of the request.

- **Individual Decision profile**: the request-Multi is divided into multiple individual requests, and each request is evaluated individually. Then, as many results as requests are returned to the PEP.
- **Combined Decision profile**: the evaluation of individual requests is same as the Individual profile. Yet, if all the individual decisions are the same (i.e., all *Permit* or all *Deny*), then only one decision is returned. If even a single exception occurs, then the combined result is *Indeterminate*. Moreover, if any of the results, before being combined, contains an Obligation or an Advice, then the combined result is *Indeterminate* by default.

If we consider the case where the repeated attributes are resources, it is clear that each request for a resource is evaluated by itself, without considering any of the attributes of the other resources. As a consequence, there is no possibility of making fine-grained decisions considering the cross-evaluation of attributes, like in case of multiple resource aggregation. Therefore, we believe that such evaluation of requests-Multi can be seen as a limitation in the use of XACML where multiple resources must be accessed at the same time, and where operations of data aggregation and their impact on the evaluation of the request must be taken into consideration. Moreover, both the Individual and Combined Decision profiles can be seen as a limitation in such a scenario, because, according to these profiles, either the access is granted for all the resources (Combined Decision) or for an individual resource by itself (Individual Decision). We trust that, when evaluating datasets with heterogeneous usage terms (and therefore, policies), a more effective approach would consist in identifying *clusters* (i.e., *subsets*) of resources whose access can be granted.

## 1.3 Our approach

Our approach considers the expression and enforcement of security policies in order to regulate the usage of a dataset when aggregated and/or analyzed together with other datasets. We identified a gap in the existing XACML Profile Multi, where a request for multiple resources is always evaluated as the result of the evaluation of individual resource requests. The actual model imposes multiple individual resource request evaluations, and excludes the

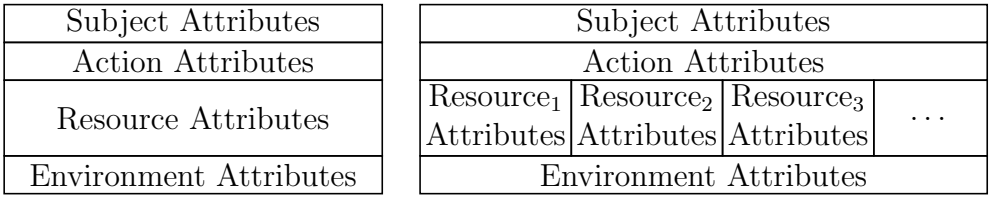



Figure 1.3: *left*: Standard Context structure, *right*: Extended Context structure

possibility to take into account information (i.e., attributes) from the original multiple-resource request. To fill this gap, we extend the XACML Profile Multi, by means of a number of original contributions.

(a) Extension of the XACML policy language with a new condition, i.e., the *OtherData* condition, in order to evaluate attributes of other resources part of the same request-Multi (a request for multiple resources according to XACML Profile Multi). It is detailed in Section 1.3.1.

(b) Definition of a new concept, i.e., the *Extended (Request) Context*, that caters for all attributes of a request-Multi, in order to allow the evaluation of the new condition previously mentioned. It is again described in Section 1.3.1.

(c) Extension of an efficient XACML policy evaluation and management system in order to support in-memory management of the newly defined policy type and efficient target matching. It is illustrated in Section 1.3.2.

(d) Extension of the set of possible evaluation results of a request-Multi, in order to deal with situations where it can be authorized only for a part of the requested resources. It is reported in Section 1.4.

Our approach designs an access control mechanism that requires the enforcement of the architecture illustrated in Figure 1.2 and that controls a set of resources (datasets) stored in the Storage module, and it is structured in the following steps.

1. Creation of an in-memory graph representation for all available policies associated with a set of resources under control by our mechanism, before processing any request evaluation. This step is necessary to create a representation of all policies that permits an efficient evaluation of access requests (be them XACML Profile Multi requests or not). This step is illustrated in Section 1.3.2, and requires an extension of

PAP and PDP. Indeed, the PDP builds a policy graph, and the PAP has to manage this policy graph instead of plain policies.

2. Once a request-Multi for multiple resources is received, the PDP proceeds with a target matching phase (Section 1.4.1), that identifies the applicable policies for the requested resources and their evaluation order, as ordered set of graph visit paths.

3. Subsequently, the PDP has also to evaluate conditions (Section 1.4.2) associated with the identified graph visit paths. The logic of the PDP must be extended in order to consider both traditional XACML conditions as well as our new condition option, and evaluate them against the Extended Context. As a result, the PDP creates a graph representation of the authorized resources and of the applicable constraints among resources, e.g., situations like "resource $a$ has conditions that prevent its authorization if combined with resource $b$".

4. Last, the PDP provides a cluster of resources to the PEP, which has to compute the request-Multi (authorization) enforcement (Section 1.4.3). According to the evaluation type specified in the request, the PEP can authorize the access to either all or a part of the requested resources.

As mentioned, in this paper we consider a subset of requests-Multi, in particular, a request-Multi for multiple resources (i.e., multiple resource categories, in XACML language) and as a use case action, an operation involving data aggregation (e.g., an operation of machine learning). Let $G \subseteq A$ be a set of aggregation operations, we define our request-Multi as $Req = \{s, a, [r_1, \ldots, r_N]\}$, with $s \in S, a \in G$, and $\forall_{i \in 1,\ldots,N} r_i \in R$.

The limitation of the actual model is due to both the decomposition of requests-Multi into individual requests, which are evaluated with isolated Contexts, and to the impossibility of expressing constraints on other resources belonging to the same request-Multi.

In order to overcome these limitations, we propose an approach in which requests-Multi are not evaluated as part of isolated individual requests. Our approach relies on the creation of an *Extended Context*, which encompasses all the attributes of all the entities appearing in the request-Multi. It is depicted in Figure 1.3, in contrast with the standard XACML Context normally used to evaluate Target and Condition. Moreover, we also introduce further constraints in the policies, in order to express relationships that occur among resources.

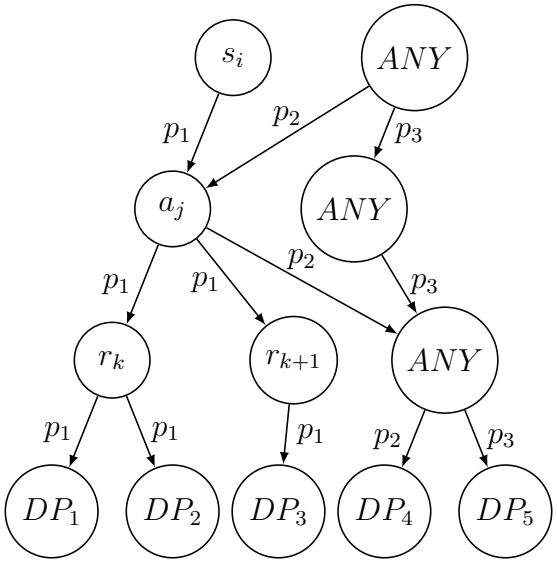



Figure 1.4: Policy graph with visit priorities

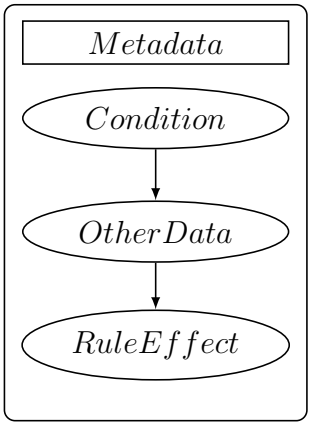



Figure 1.5: Decision Path node structure

### 1.3.1 XACML Extension

Current XACML policies have a fixed structure: every policy declares a target, a set of rules, obligation expressions and advice expressions. As well as policies, also rules have a fixed structure: one target, one condition, and obligation and advice expressions. As introduced in the previous sections, we propose to introduce modifications to XACML constructs to reflect constraints and/or relationship among the resources of a request-Multi. As we want to allow a data owner to express conditions on other resources part of the same request (i.e., inter-entities constraints), it seems natural to propose a new type of XACML Condition. To this purpose, we introduce a mechanism for regulating the relationship between two resources, that we explain in details in Section 1.4. Due to the current limitation, we propose to extend the standard XACML language in order to be able to declare more than one condition per rule, to allow the coexistence of our new and traditional conditions. In order to be compliant with the XACML schema, we need to introduce an identifier for the new condition. Thus, we extend the standard specification also in order to allow users to declare two new optional attributes for a Condition node: *ConditionID* and *OtherData*. *ConditionID* is a string value declaring the identifier of the Condition node. *OtherData*, instead, is a boolean value that indicates whether the Condition must consider the resource of the individual request currently under evaluation (i.e. the default conditions behavior, if its value is false) or the other resources in the Extended Context (otherwise). The OtherData condition evaluation is

detailed in Section 1.4.2. We assume that the default value for OtherData is false, so that our extension can be fully compatible with the current profile. An example of an extended XACML policy is available in the Appendix. In the remainder of this paper, we refer to the normal condition as *Traditional condition* (or simply, *Condition*), and to the new introduced one as *OtherData* (consistently with its attribute set to true).

### 1.3.2 Policy Representation

The introduction of OtherData condition and Extended Context requires adaptations in the traditional XACML processing. Before being able to evaluate them, we have to face the problem of dealing with significant amounts of policies. Previous works [80, 45, 79, 65] showed clear limitations and scalability issues in the actual XACML policy evaluation and management. To overcome this limitation, as a first step, our approach considers a graph-based policy in-memory representation, in order to anticipate bottlenecks that would limit its effectiveness and thus adoption. Adopting the approach presented in [79, 9], we build a directed graph data structure using the attributes appearing in the target sections of a policy and its rules elements, giving to the graph nodes a precise order, and organize our data structure in four distinct levels. We group the attributes according to their XACML category, and consider attributes related to the subjects as the first level, attributes related to the actions as the second level, and attributes related to the resources as the third level. Finally, and differently from the previous works, the fourth level contains the later-described *Decision Paths*. In our approach, we build one global policy graph (that we define as the *policy graph*) using all the available policies in the PAP, and update it whenever a new policy is loaded into the access control system. However, transforming the policies into a policy graph, one could loose the order of evaluation of policies and rules, that is relevant for certain combining algorithms, thus introducing possible inconsistencies. Therefore, in order to support the correct evaluation of the combining algorithms, we introduce a link prioritization procedure. Its result is the *link label*.

**Definition 1.3.1.** *Each link label $p_i$ denotes a graph visiting priority. Lower link values indicate higher routing priority in graph visiting.*

$$\exists i, j \in \mathbb{N} : i < j \Rightarrow p_i > p_j$$

Link labels indicate all valid graph visiting paths and their priority. Therefore, it is possible to represent the policy/rule combining algorithm,

evaluating policies/rules in the same order as they would have been evaluated by a traditional XACML engine. In the literature, [62] and [66] already demonstrated the feasibility of similar approaches. Link labels are assigned while building the graph, and can be easily recalculated when inserting a new policy, according to the combining algorithm of the latter. It is important to notice that link labels are relevant when dealing with policies affecting the same $\{s, a, r\}$. Consistency among multiple policies regulating the same resources is a necessary assumption, as for any other XACML solution. If we consider that in our setting $r$ is a dataset with an owner defining an access policy, there cannot be a situation where another entity can express another policy for the same $r$. Therefore such assumption is sufficient to avoid ambiguities in the policy graph. Figure 1.4 shows an example of our policy graph, composed of a top-down-ordered subject-action-resource layer set with three different navigation paths and priorities as graph routes, expressed as link labels. Looking at the structure of the graph, each node at the bottom level represents a *Decision Path*. A Decision Path is a branch composed by the nodes Condition, OtherData, and RuleEffect of a rule, in addition to metadata such as the IDs of the rule and policy that branch is extracted from, and possible obligations that should be enforced before returning the decision to the user. Figure 1.5 shows the structure of a Decision Path node. Thanks to this organization, our policy graph can decouple the policy evaluation process in two steps: a former *target matching* phase followed by a latter *conditions evaluation*. These two phases are then followed by a final *fine-grained decision* phase, whose goal is to find all allowed clusters of input data.

## 1.4 Request-Multi evaluation

### 1.4.1 Target Matching

**Definition 1.4.1.** *We define as target matching phase, the process of finding the set of all viable $\{s_i, a_j, r_k\}$ visit routes in a policy graph.*

In the target matching phase, the XACML policy target matching process (needed to identify the applicable policies to evaluate a request) becomes a simple graph visit. Graph visit is only allowed if the link between two nodes exists, and if the $i^{th} - 1$ link previously traversed has the same link label of the $i^{th}$ link (i.e., the current link). The priority among all the applicable visits (as merge of multiple policies of multiple data owners) is determined by a specified policy combining algorithm, that can be a configuration of the

```
1: generate single requests {s_i, a_j, r_k} from RM
2: MR := ∅                                                    # matching resources
3: for each single request {s_i, a_j, r_k} do
4:     for each viable route in PG for {s_i, a_j, r_k} do
5:         save the decision paths DPs of this route
6:     end for
7:     if there is at least a viable route for {s_i, a_j, r_k} then
8:         add r_k to MR                                      # r_k is a matching resource
9:     else
10:        the result of the individual evaluation of {s_i, a_j, r_k} is Not Applicable
11:    end if
12: end for
```

Figure 1.6: Target matching of a request-Multi $RM$ in a policy graph $PG$

| Link | Route | Decision Path |
|---|---|---|
| $p_1$ | $\{s_i, a_j, r_k\}$ | $DP_1 \vee DP_2$ |
| $p_2$ | $\{ANY, a_j, ANY\}$ | $DP_4$ |
| $p_3$ | $\{ANY, ANY, ANY\}$ | $DP_5$ |

Figure 1.7: Target matching for an individual request

system or defined in a *PolicySet* element that contains all policies, according to the considered use case. In any case, according to the assumption expressed in Section 1.3, data owners determine unequivocally and consistently the policies for their resources. In addition to this, ordering among policies of different data owners is irrelevant with respect to policy evaluation correctness, as no one is allowed to express policies for resources they don't own, so they cannot alter the correct graph visit order.

Given a request-Multi $\{s_i, a_j, [r_1, \ldots, r_n]\}$, we split it into $n$ individual requests $\{s_i, a_j, r_1\}, \ldots, \{s_i, a_j, r_n\}$, where $n = |R|$. Then, for each individual request, we follow all the possible routes in the graph. Decision Paths are not considered at this stage (their evaluation is delegated to the next phase). If the individual request does not have any applicable route for the resource (i.e., no policy exists to regulate the request), that resource is *filtered out*, i.e., the result of this individual request evaluation is *Not Applicable*. Figure 1.6 shows our target matching procedure.

The aim of the target matching phase is to find all the applicable Decision Paths, filtering out resources that are not matched by any policy.

**Definition 1.4.2.** *Given an individual request $\{s_i, a_j, r_k\}$, $r_k$ is a matching resource if at least one viable route for $\{s_i, a_j, r_k\}$ exists in the graph.*

**Theorem 1.4.1.** *For each matching resource, at least one Decision Path exists.*

*Proof.* Assume that $r_k$ is a matching resource but it has no Decision Paths associated to. Being $r_k$ a matching resource, it follows that at least one policy exists, having at least one rule, that matches $r_k$. Hence, the following elements exist: a RuleEffect (that is a mandatory attribute for every Rule), a Traditional condition (possibly null), and an OtherData condition (possibly null) related to $r_k$. Yet, Condition, OtherData and RuleEffect are the elements composing a DecisionPath. □

Given a set of matching resources, the order in which we evaluate the correspondent Decision Paths is given by the link label. In Figure 1.7 we see the result of the target matching phase of an individual request $\{s_i, a_j, r_k\}$ over the policy graph illustrated in Figure 1.4. The possible routes are ordered according to their link label, that is also the order they are evaluated during the second phase (*conditions evaluation*).

### 1.4.2 Conditions Evaluation

In this phase, we evaluate the expressed conditions for all the matching resources. Traditional XACML conditions express whether a resource can be authorized or not according to the traditional XACML evaluation. Our new condition type expresses whether constraints exist among the requested resources. The condition evaluation phase takes place for all matching resources, one by one. We define the matching resource currently under evaluation, as the *current-resource*, and all the other matching resources as *other-resources*.

**Definition 1.4.3.** *Given a request-Multi* $\{s_i, a_j, [r_1, \ldots, r_n]\}$*, its set of matching resources* $MR \subseteq R$*, and given an individual request* $\{s_i, a_j, r_k\}$ *with* $r_k \in MR$*, we define* $r_k$ *as the current-resource, and* $\forall_{x \in 1,\ldots,|MR|} \wedge x \neq k : r_x \in MR$*, as the other-resources.*

At this stage, all matching resources have a Decision Path, as previously discussed. Each Decision Path $DP$ can have a Condition and an OtherData condition. The former is evaluated on the current-resource request attributes, while the latter is evaluated for each of the other-resources, considering attributes of each of the other-resources by means of the Extended Context. In accordance with the actual XACML definition, if a condition (either Traditional or OtherData) is null, then it is evaluated (i.e., it is true) by default. In order to avoid ambiguities, the Decision Paths are evaluated in order, since their order was assigned at the graph construction time, considering combining algorithms for rules and policies. The conditions evaluation phase produces an evaluation matrix per matching resource. An evaluation matrix

```
1: for each matching resource r_i in MR do
2:     OR ← R − r_i                                   # other resources
3:     get the Decision Paths DPs applicable to r_i   # identified in the target matching phase
4:     initialize the condition evaluation matrix CEM of r_i
5:     for each Decision Path DP do
6:         Condition ← get Condition from DP
7:         OtherData ← get OtherData from DP
8:         if Condition is true for r_i then
9:             for each other-resource or_j in OR do
10:                if OtherData is true for or_j then
11:                    CEM[OtherData][or_j] ←true
12:                    remove or_j from OR            # no need to evaluate another OtherData condition for or_j
13:                else
14:                    CEM[OtherData][or_j] ←false
15:                end if
16:            end for
17:        end if
18:    end for
19: end for
```

Figure 1.8: Evaluate conditions of the matching resources $MR$

| $\mathbf{r_1}$ | $r_2$ | $r_3$ | $r_4$ | $r_5$ | $r_6$ |
|---|---|---|---|---|---|
| $OtherData_{id_1}$ | f | t | f | f | t |
| $OtherData_{id_2}$ | t | | f | f | |
| $OtherData_{id_3}$ | | | f | t | |

Figure 1.9: Conditions evaluation matrix for the current-resource $r_1$ and five other-resources

indicates if an OtherData condition is true or false for a certain resource. Figure 1.8 shows the process of building the evaluation matrices of a set of matching resources $MR$, and Figure 1.9 presents an example of an evaluation matrix with six matching resources, being $r_1$ the current-resource, and three Decision Paths applicable to the individual request involving $r_1$ (thus, three OtherData for its other-resources). The use of boolean values permits to be independent from the RuleEffect of a specific Decision Path. This approach brings a benefit, as shown Section 1.7.

Finally, once all the matching resources have been taken into consideration and evaluated, all the evaluation matrices produced are merged into one global decision matrix that, for each couple of matching resources, declares the RuleEffect to be applied on their relationship. The algorithm for building the global decision matrix is shown in Figure 1.10. An example of a final decision matrix with six matching resources is shown in Figure 1.11. In this figure, each row represents a current-resource (i.e., the resource that evaluates

```
1: initialize the Decision Matrix DM
2: for each matching resource r_i in MR do
3:     CEM ← get the condition evaluation matrix of r_i
4:     OR ← R − r_i                                  # the set of other-resources
5:     for each other-resource or_j in OR do
6:         if there is an OtherData true in CEM for or_j then
7:             DM[r_i][or_j] ← get the RuleEffect associated to that OtherData
8:         else
9:             DM[r_i][or_j] ← NotApplicable
10:        end if
11:    end for
12: end for
```

Figure 1.10: Merge all conditions evaluation matrices in a global decision matrix

| **Matching resources** | $r_1$ | $r_2$ | $r_3$ | $r_4$ | $r_5$ | $r_6$ |
|---|---|---|---|---|---|---|
| $r_1$ | - | D | P | NA | PO | P |
| $r_2$ | D | - | P | D | D | P |
| $r_3$ | PO | P | - | P | P | PO |
| $r_4$ | NA | DO | DO | - | D | NA |
| $r_5$ | P | P | P | P | - | P |
| $r_6$ | D | P | PO | P | P | - |

Figure 1.11: Decision matrix with six matching resources

the Condition) and columns represent the other-resources (i.e., the resources that evaluate the OtherData). Pairs $(r_i; r_i)$ are not evaluated (because a current-resource cannot be an other-resource for itself), then the admitted decisions are: P (Permit), PO (Permit with an Obligation), D (Deny), DO (Deny with an Obligation), NA (Not Applicable) or I (Indeterminate). If, considering Figure 1.9, we envisage $OtherData_{id_1}$ belonging to a Decision Path which declares a Permit effect, $OtherData_{id_2}$ associated to a Deny effect, and $OtherData_{id_3}$ associated to a Permit effect with an Obligation, these effects are reported accordingly in the first row of the decision matrix shown in Figure 1.11, as effects of the relationship of these resources with $r_1$. It is worth noticing that there is a Not Applicable effect associated to the relationship $(r_1; r_4)$ since $r_4$ does not verify any OtherData associated to the current-resource $r_1$ (as a matter of fact, in Figure 1.9 all values of column $r_4$ are false).

### 1.4.3 Fine-grained Decision

After the target matching and conditions evaluation phases, the fine-grained decision phase gives a response on the aggregation of the inputs. We distinguish among four types of fine-grained decisions, listed below.

• **Individual Evaluation**. Each resource request is evaluated individually, and the same number of different responses are returned to the user (as for XACML Multi Profile).

• **Combined Decision**. All resource requests are evaluated to produce a single response, to allow access to either all or none of the resources (as for XACML Multi Profile).

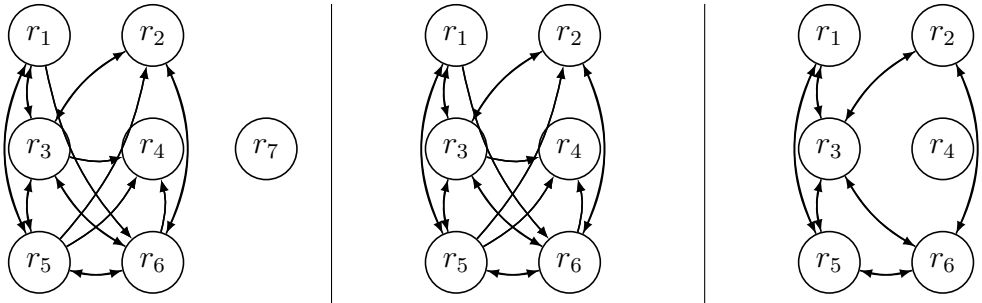
• **Max-Subset (new contribution)**. This decision type analyzes the decision matrix, and returns the subset(s) with highest cardinality of accessible resources (i.e., those with P and PO entries).

• **All-Subsets (new contribution)**. Similarly to the previous profile, here the decision matrix is analyzed to return all subsets of accessible resources.

The computation of XACML Multi decisions is straightforward, and derives from the standard XACML evaluation. For *Max-Subset* or *All-Subsets* decisions, our approach consists of building a graph derived from the decision matrix, where nodes represent resources and directed edges represent decisions. For instance, a request-Multi for 7 resources (but only 6 matching resources) that produces the decision matrix depicted in Figure 1.11, is translated into the graph in Figure 1.12(a). Figure 1.12 illustrates the fine-grained decision process: (a) non-matching resources and resources that do not have an explicit Permit (i.e., they don't have incoming/outgoing edges) are discarded from the graph as they cannot be accessed, (b) non-bidirectional Permit (or Permit-Obligation) edges are erased, since this means that one of the two resources is incompatible with the other one (i.e., it cannot be accessed with the other one), (c) a clustering algorithm is applied (in our implementation, we adopted *cliques* [64, 19]). In case of an *All-subsets* decision, the PDP communicates to the PEP all the possible clusters of resources, whereas in case of a *Max-subset* decision it communicates only the largest cluster(s).

## 1.5 Use Case

In this section we present a use case scenario derived from the actual practice in cyber security, to be used in Section 1.7. A national Computer Emergency Response Team (CERT) acts as primary information point for monitoring, awareness and reaction to cyber attacks. It cooperates with public bodies




(a) Node filtering (b) Edge filtering (c) Clustering

Figure 1.12: Fine-grained decision phase

| **timestamp** | **host** | **src** | **IP** | **latitude** | **longitude** | **ISP** | **...** |
|---|---|---|---|---|---|---|---|
| timestamp1 | host-tokio | s1 | 111.111.111.111 | y1 | x1 | Alice | ... |
| timestamp2 | host-moscow | s2 | 222.222.222.222 | y2 | x2 | Bob | ... |
| timestamp3 | host-boston | s3 | 132.123.123.123 | y3 | x3 | Carol | ... |
| ... | ... | ... | ... | ... | ... | ... | ... |

Figure 1.13: Sample entries in our database corresponding to attacks to ISPs' honeypots

and companies that must trust each other, but especially the CERT, as it receives very sensitive information about attacked entities. Indeed, if not properly protected, those information may cause reputation damages, e.g., allowing benchmarking of companies, or even attracting additional attacks using the same patterns. On the other hand, the CERT needs to produce timely alerts about ongoing threats and coordinate the attack response with multiple actors: the attacked company, cloud and Internet service providers, law enforcement agencies, etc. In this the example, we will use a public dataset of attacks to honeypots provided by a number of ISPs, and we will consider the role of a CERT that analyzes data from the different honeypots owners, showing an example of regulated analysis. The requirement is that any benchmarking among ISPs must be prevented. We assume that the CERT runs an architecture as for Figure 1.2 to control access to the different pieces of information, using policies provided by the ISPs. The mechanism is integrated with a ML library to be used for the analysis. Our example considers conflicting resource policies, that we will discuss in Section 1.7.

**Input dataset.** The data we used for our example is derived from a publicly available dataset [48] of about 450,000 samples of attacks to multiple honeypots. Figure 1.13 depicts the structure of our dataset: it represents malicious connections to the honeypots, with the recorded attack IP

address and its approximate geographic location. We supplemented the original dataset by adding the column "ISP", obtained by querying the Internet service WHOIS and assigning a random identifier for each discovered ISP. We did not work with actual ISP names as deemed unnecessary to exemplify our approach.

## 1.6 Implementation

We implemented our proposal on top of an existing open source XACML framework, developed in Java and provided by AT&T [1]. We extended both the architecture, in order to reflect the components illustrated in Figure 1.2, and the components themselves, to support the extension of the XACML language introduced in Section 1.3. Finally, we implemented the evaluation process introduced in Section 1.4.

With regard to the XACML extension, in our implementation we have to support up to two Condition elements, i.e., the standard ("Traditional") and OtherData conditions, together with the introduction of two new attributes, i.e., ConditionID (string) and OtherData (boolean). To this extent, we worked on the PEP and PDP.

The PEP was hooked to a Python library, Scikit[2] so that it could intercept calls to machine learning operations (e.g., k-means) and produce an XACML request. A simple logic allows the PEP to identify the input parameters and to produce an XACML authorization request. If multiple parameters are involved, the PEP produces a request-Multi as previously described, that is sent to the Context Handler and thus to the PDP.

The PDP communicates with the PAP in order to load the policies associated with the requested resources. In our implementation, the PDP builds a policy graph as described in Section 1.3, and stores it in the PAP. Indeed, this policy graph can also be cached in a file in the PAP, and loaded at request time instead of triggering repeated builds, provided that the cached graph is updated at each policy insertion or modification. For simplicity, we considered stable policies and built our policy graph as an adjacency matrix (i.e., a matrix whose elements express if pairs of vertices in a graph are adjacent or not).

In addition to this, the original PDP was extended to also support our new evaluation logic, that is, to run the three-step process illustrated in Section 1.4. The standard behavior of the AT&T standard engine (and of many other XACML engines) is to instantiate an Evaluation Context object (in

[1] https://github.com/att/XACML/

[2] `https://scikit-learn.org/stable/index.html`

case the request is a request-Multi, a context for each individual request part of the request-Multi). Each Evaluation Context contains the attributes provided in the (individual) request, complemented by other attributes from the PIP in case of need. Henceforward, traditional engines compare the Evaluation Context with the Target section of the available policies, to determine the applicable policy and rule(s) for the request. Subsequently, and according to the combining algorithm, the first applicable rule is evaluated, together with its condition (if present), and, if the result is positive, the rule effect (Deny or Permit) is then returned.

In our implementation, at this stage we build an *Extended Context* filled with all the attributes of the request-Multi, as shown in Section 1.3. Subsequently, the Extended Context is used to visit the corresponding applicable rows of the graph adjacency matrix (i.e., the policy graph), so that non-reachable resources are filtered out, as explained in Section 1.4.1. The remaining resources (i.e., the matching resources) go through the Decision Path evaluation, that is, the evaluation of Traditional and OtherData conditions. In order not to leak any information, we slice the Extended Context such that, for the evaluation of a Condition, least information (on the involved request) is exposed. This is equivalent to generating individual requests, using the attributes contained in an individual request to instantiate an Evaluation Context, and using then this Evaluation Context for evaluating a Traditional or OtherData condition. We observed that, at this stage, conditions might be evaluated multiple times, for example in cases where the same policy applies to multiple resources, or when similar conditions are expressed in multiple policies. For this reason, we introduced a caching mechanism to improve overall performance. The result of the conditions evaluation is a decision matrix that is then used as an adjacency matrix to build the initial graph of the clustering phase. In our implementation, we used *graphstream*, a Java library for dynamic graphs[3]. The operations of this phase are executed as described in Section 1.4. The result of the request-Multi is then returned to the PEP, indicating one or more authorized clusters of resources, if any (i.e., the Max-subset or All-subsets decision profile).

## 1.7 Performance Evaluation

In this section we present the results of four different experiments we ran to assess the correctness of our approach and its practical applicability. From the dataset described in Section 1.5, we extracted 10 evaluation datasets of about 50k entries representing attacks to honeypots, as if they were provided

[3]http://graphstream-project.org

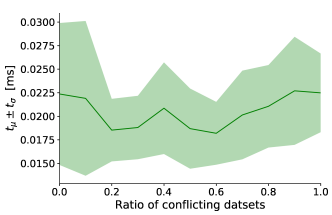



Figure 1.14: Target matching (simple policy)

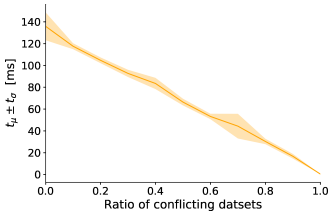



Figure 1.15: Conditions evaluation (simple policy)

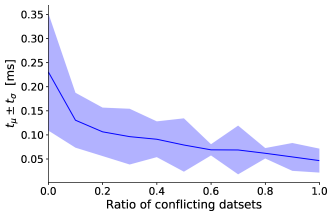



Figure 1.16: Clustering (simple policy)

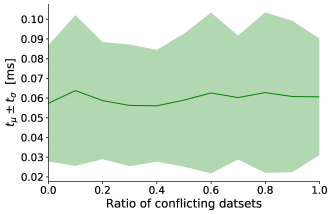



Figure 1.17: Target matching (complex policy)

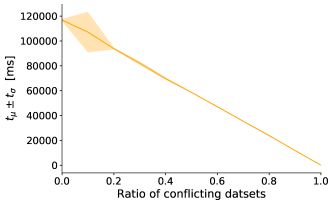



Figure 1.18: Conditions evaluation (complex policy)

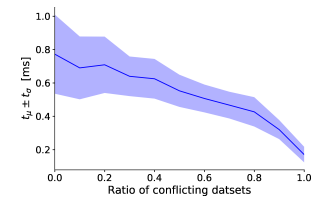



Figure 1.19: Clustering (complex policy)

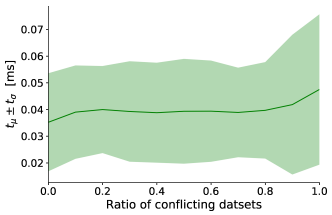



Figure 1.20: Target matching (complex policy with caching)

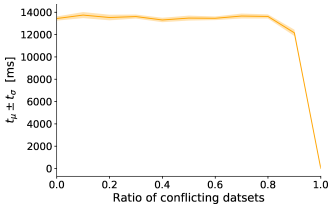



Figure 1.21: Conditions evaluation (complex policy with caching)

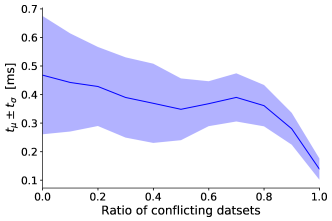



Figure 1.22: Clustering (complex policy with caching)

by different entities. We also associated to each dataset a set of attributes like owner (CompanyA, CompanyB, ...), an identifier, and a different access control policy, to allow the evaluation of access requests using our implementation. We assume that our implementation receives a request to compute an aggregation on a certain number of evaluation datasets (in particular, a k-means clustering [5]). All policies use the First-Applicable rule-combining algorithm and declare two rules: the former results in a Permit decision and specifies a Target, a Traditional and an OtherData conditions, whereas the latter is a default Deny rule matching any request.

The experiments took into account policies of different complexity, in order to assess our approach and implementation in different usage scenarios.

We refer to them as follows: *"Simple Condition"*, *"Complex OtherData"* and a derivation of the latter, *"Complex OtherData Modified"* where we introduced a rudimentary optimization for one of the most onerous tasks. We also compared the performance of our approach with those of the existing request-Multi profiles in order to understand how the difference in their specific use cases would materialize on their respective performance. All such cases are later described in this section. We decided not to benchmark the performance of our engine against other existing engines because the scope of our experiments is to give an indication of the overhead introduced by our novel approach.

For the experiment discussion, we need one more definition: two datasets (resources) are *conflicting* if they cannot be clustered, i.e., when their decision matrix value (evaluated against the OtherData condition as previously mentioned) is not Permit. To assess the scalability of our solution, we identified as a significant parameter the *ratio of conflicting resources* in a request. When this parameter is high, many resources are incompatible with each other, meaning that either the Traditional condition returns false (and the resource in evaluation is filtered out in the rest of the evaluation) or the OtherData condition results in a sparse decision matrix: therefore, a part of the overall evaluation process terminates at that stage. On the other hand, parameter values close to 0 trigger a complete process (evaluation of 2 conditions on all resources including clustering), resulting in higher execution time. For this reason, we defined 5 reference parameter values, from 0.0 to 1.0 at 0.2 step increases, and we created a combination of policies and requests for each experiment so that the input space would represent equally each of the values.

All the experimental results have been obtained using a machine with Windows 10, Intel i7-4790 CPU, 3.60 GHz 4 cores, and 16 GB memory.

### 1.7.1 Simple condition evaluation

In the first experiment, we created 1000 access requests for 10 datasets (also referenced as resources later on), balancing the ratio of conflicting requests as explained. The policies we used included two simple attribute matching operations as (Traditional and OtherData) conditions, ending up with a standard ABAC evaluation. Traditional condition was in the form: *resourceX.owner=Company1* and *requestor.role=analyst*, while for OtherData it was: *otherResource.owner=Company1*. We show the XACML policy used for this experiment in Appendix A.1.

Let us analyze the results for the main phases of our solution. Figure 1.14 confirms how target matching is practically unaffected by any conflicting re-

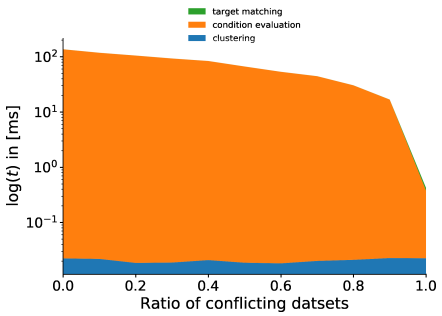



Figure 1.23: Total decision time for a simple policy

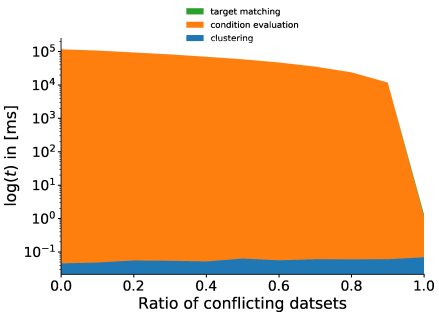



Figure 1.24: Total decision time for a complex policy

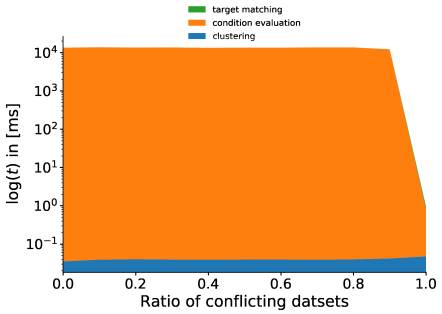



Figure 1.25: Decision time for a complex policy with caching

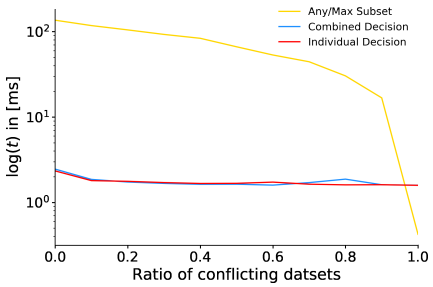



Figure 1.26: Comparison of decision profiles

source ratio, while in Figure 1.15 we can see that resource conflicts impact linearly on conditions evaluation, as in this experiment the OtherData evaluation is made useless by the Traditional condition verification. The conflicting resource ratio also affects clustering (Figure 1.16), that shows shorter execution times and therefore easier clustering computation when less resources are available.

Lastly, Figure 1.23 shows the total time required for the evaluation. From this figure, it is visible that conditions evaluation absorbs most of the evaluation time for all considered ratios.

### 1.7.2 Complex OtherData evaluation

In this experiment, we considered a complex OtherData condition, to assess the performance of our implementation with computationally heavy constraints. To do so, we requested in the OtherData condition to compute the expected value **E** of a feature in the other dataset, and to compare it with a reference range. We show this condition in Appendix A.2. We performed 100 access requests, representing fairly the conflict resource ratio. The tar-

get matching phase in Figure 1.17 exposes a behavior similar to the previous case. Intuitively, target matching depends on the policy graph and the number of requested resources, irrespective of any conflict ratio. It is interesting to note that the conditions evaluation phase (Figure 1.18) exposes the same trend of the previous experiment, i.e., linearly decreasing at higher ratio of conflicting resources. There is a deterioration in terms of performance due to the higher complexity of the conditions to be evaluated, and therefore fully expected. The clustering phase, shown in Figure 1.19, depends on the number of inputs, i.e., resources evaluated, and affects the final result with a negligible overhead, if compared to that of conditions evaluation.

Finally, Figure 1.24 shows the total time required for the evaluation of this policy. It is again well visible that the bottleneck is the conditions evaluation phase, and that it totally dominates the target matching and clustering phases.

### 1.7.3 Complex OtherData evaluation with caching

In this experiment, we used the exactly same setting of the previous case, but we introduced a naïve caching system to reduce the overhead introduced by the conditions evaluation phase, processing it as a dynamic programming problem. As expected, the target matching (Figure 1.20) and clustering (Figure 1.22) phases show the same trend (and comparable results) of the previous experiment. What it is interesting to note is the conditions evaluation phase. As we noticed in the previous experiment, our implementation processed the same OtherData section for the same dataset many times, causing a consequent impact in terms of performance. So the naïve caching strategy allows our implementation to evaluate the same OtherData condition for each dataset only once (if ever required). Results are shown in Figure 1.21. Thanks to this strategy we were able to lower the load of this phase of one order of magnitude. Indeed, the OtherData is evaluated only once per dataset, i.e., for each dataset the expectation value $\mathbf{E}$ is computed only once, and this causes the flatten trend visible in Figure 1.21. As usual, Figure 1.25 shows the total execution time, where the conditions evaluation is still the bottleneck, even if the total required time is constant, and one order of magnitude lower than in the previous experiment.

### 1.7.4 Comparison with the Individual and Combined profiles

In this experiment, we wanted to assess the difference in performance of traditional request-Multi evaluation against our new profiles, i.e., Max-subset

and All-subsets. Indeed, we expected to observe a certain difference due to the additional computation demanded by our approach and we wanted to measure it. To this aim, we used the Simple Condition evaluation setting and we issued all requests, asking for Individual and Combined decisions to the unmodified AT&T XACML engine, and All-subsets and Max-subset decisions to the same engine extended in order to support our approach. The results of this experiment are shown in Figure 1.26.

The standard Combined and Individual decisions show an overlapping trend, because in both cases each individual request is evaluated in isolation (with the traditional context) and, at the end of the process, either combined with the others or returned as-is. On the contrary, our approach requires a more extended evaluation. Indeed, after the graph visits to identify the matching resources, we have to iterate over those resources for evaluating both the Traditional and OtherData conditions, and cluster the results.

# Chapter 2

# Related Work

With respect to the graph representation of policies, our paper is not the first contribution in this direction [62, 66, 43, 79, 65]. Among these, in [66] the authors propose an approach based on the construction of a graph composed itself of two trees, evaluated in two distinct phases. The first step consists in building a *Matching Tree*. Every node of a Matching Tree corresponds to a different attribute extracted from the Target node of the policy. Leaves are themselves roots of another tree, the *Combining Tree*, that is evaluated in the second step, and that is composed by the rules applicable to some path. The leaves of a Combining Tree are Rules. According to the selected rule-combining algorithm, an integer value corresponding to the "priority" is associated to each rule. Finally, the rules are executed in descending order starting from the one with highest priority. The approach proposed by the authors is similar to ours, but they differ by the way graphs are built. Omitting the difference between the chosen data structures, that is trivial, the main difference from our approach is that bottom nodes of our policy graph are not another tree themselves, but nodes containing only the Traditional and OtherData conditions of a rule. Moreover, in [66] the authors do not consider OtherData, as they are introduced in our work.

In [18] the authors propose a novel commercial model for data security preventing conflicts of interest. This model has been widely studied, and several solutions have been proposed [61, 8, 74]. From some perspective, our work could be inserted among these ones. In [41] and [24] the authors state that XACML does not support separation of duties (SoD). In order to overcome this limitation, in [41] the authors propose the use of a novel policy language, called Next Generation Access Control (NGAC). In NGAC, policies specify relations among attributes, and SoD can be enforced dynamically creating deny relations based on the access history. A similar solution has been proposed in [23], and later conformed to XACML in [24], even though

the authors analyze SoD only from a Role-Based Access Control (RBAC) perspective. In particular, they enforce dynamic policies: they implement a blacklist at policy-level, which is updated at every request. The authors propose a solution in which an Obligation, enforced by the PEP, writes a new Rule node directly in the policy stored in the PAP. This model is different from ours, because our basic assumption is that the owner of a resource specifies the policy for that resource, and she is also the only actor allowed to update her policies. In [49] the authors enforce mutually exclusive authorization policies in order to solve the problem of separation of duties. Yet, even if the authors study the problem from an ABAC perspective, they only propose a formal approach to the problem, and do not consider a specific language, such as XACML.

Airvat [72] is a solution for providing security and privacy guarantees over MapReduce computations on sensitive data. Airvat decouples an untrusted map operation followed by a secure reduce, that enforces differential privacy. Nevertheless, this solution relies on SELinux [76] for Mandatory Access Control over the input resources, and does not take into consideration possible conflicts in aggregating those data. GuardMR [81] is another solution that aims at enforcing security guarantees over MapReduce computations. Yet, the threat model proposed by the authors is different from ours. Indeed, it aims at targeting attacks to the infrastructure, such as vulnerabilities of the execution environment. Besides, both Airvat and GuardMR do not take into consideration the underlying access control system, and how access control decisions are taken.

# Chapter 3

# Conclusions

The adoption of AI/ML techniques has increased in the last years and will grow further as new information sources and new analysis techniques become available. However, in communities of practice (e.g. collaboration in cyber security analysis) or from legal requirements (e.g. specific terms for analysis of personal data), a need emerges for controlling the scope, the methods and the objectives of such analysis. Our proposal goes in this direction, describing an access control mechanism that extends the state of the art in controlling requests for multiple resources regulated by different policies. To this aim, we take into account constraints among resources, not forgetting the performance requirements necessary for modern applications, especially if running on the cloud. To implement our method, we extended a standard XACML open source implementation in order to support new policy language constructs, and to adopt an high-speed in-memory target matching algorithm based on policy graphs. Besides, we integrated our access control mechanism with a ML library for the evaluation of mathematical conditions, and we evaluated the performance of our solution in different use cases. The preliminary results are promising as the performance remain at an acceptable level, especially when compared with those of the complex mathematical operations to be regulated. As a future work, we intend to further extend the scope of this proposal, looking at regulating closely ML operations in Big Data infrastructures.

# Appendix A

# Policies

## A.1 XAMCL Policy with simple Conditions

In this section, we show an example of simple XACML conditions. Both the Traditional and the OtherData require to fulfill simple constraints based on the attributes of the request.

Listing A.1: Simple Conditions

```
<!-- excerpt of policy regulating access to a dataset R1: XACML Policy and Target do not
     deviate from standard XACML -->

<!-- this is an example of Traditional Condition. It requires the user role to be a
    CyberSecurityAnalyst, and the owner of the current-resource to be Company1 -->

<xacml:Condition ConditionId="cond_a">

  <xacml:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">

    <xacml:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">

      <xacml:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only
          ">

        <xacml:AttributeDesignator AttributeId="urn:oasis:names:tc:xacml:2.0
            :subject:role" Category="urn:oasis:names:tc:xacml:1.0:subject-
            category:access-subject" DataType="http://www.w3.org/2001/XMLSchema#string"
            MustBePresent="true"></xacml:AttributeDesignator>

      </xacml:Apply>

      <xacml:AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
          CyberSecurityAnalyst</xacml:AttributeValue>

    </xacml:Apply>

    <xacml:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">

      <xacml:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only
          ">

        <xacml:AttributeDesignator AttributeId="example:subject:subject-employer"
            Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
            DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true"></
            xacml:AttributeDesignator>

      </xacml:Apply>

      <xacml:AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">Company1<
          /xacml:AttributeValue>

    </xacml:Apply>
```

```
  </xacml:Apply>

</xacml:Condition>

<!-- this is an example of OtherData Condition. It requires the owner of the other-
     resource to be Company1 -->

<xacml:Condition OtherData="true" ConditionId="cond_b">

  <xacml:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">

    <xacml:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">

      <xacml:AttributeDesignator AttributeId="example:resource:resource-owner" Category=
           "urn:oasis:names:tc:xacml:3.0:attribute-category:resource" DataType="http://
           www.w3.org/2001/XMLSchema#string" MustBePresent="true"></
           xacml:AttributeDesignator>

    </xacml:Apply>

    <xacml:AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">Company1</
         xacml:AttributeValue>

  </xacml:Apply>

</xacml:Condition>
```

## A.2 XACML Policy with complex OtherData

In this section, we show an example of a complex OtherData Condition node. Differently from the case presented in the previous section, which only required to compare attributes, this condition requires to perform a computation over one of the features of the dataset under evaluation.

Listing A.2: Complex OtherData

```
<!-- excerpt of policy regulating access to a dataset R1: XACML Policy, Target and
     Condition do not deviate from standard XACML -->

<!-- this is an example of OtherData Condition. It prescribes to compute the expectation
     of the feature(s) under analysis and verify that they are bounded in a certain
     interval, in order to avoid the analysis of the resource dataset R1 with non
     homogeneous data thus revealing some R1 characteristics -->

<xacml:Condition OtherData="true" ConditionId="cond_b">

  <xacml:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">

    <xacml:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:double-greater-than">

      <xacml:AttributeValue DataType="http://www.w3.org/2001/XMLSchema#double">100</
           xacml:AttributeValue>

      <xacml:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:double-one-and-only
           ">

        <xacml:Apply FunctionId="example:compute_expectation">

          <xacml:AttributeDesignator AttributeId="example:features-under-analysis"
               Category="urn:oasis:names:tc:xacml:3.0:attribute-category:action" DataType
               ="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true"></
               xacml:AttributeDesignator>

        </xacml:Apply>

      </xacml:Apply>

    </xacml:Apply>
```

```
    <xacml:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:double-less-than">

      <xacml:AttributeValue DataType="http://www.w3.org/2001/XMLSchema#double">50</
          xacml:AttributeValue>

      <xacml:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:double-one-and-only
          ">

        <xacml:Apply FunctionId="example:compute_expectation">

          <xacml:AttributeDesignator AttributeId="example:features-under-analysis"
              Category="urn:oasis:names:tc:xacml:3.0:attribute-category:action" DataType
              ="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true"></
              xacml:AttributeDesignator>

        </xacml:Apply>

      </xacml:Apply>

    </xacml:Apply>

  </xacml:Apply>

</xacml:Condition>
```

# Bibliography


[1] H. Albaroodi, S. Manickam, and M. Anbar. A proposed framework for outsourcing and secure encrypted data on OpenStack object storage (Swift). *Journal of Computer Science*, 11(3):590–597, 2015.

[2] H. Albaroodi, S. Manickam, and P. Singh. Critical review of OpenStack security: Issues and weaknesses. *Journal of Computer Science*, 10(1):23–33, 2014.

[3] E. Andreeva, A. Bogdanov, and B. Mennink. Towards understanding the known-key security of block ciphers. In *Proc. of FSE*, Hong Kong, November 2014.

[4] C. Anglano, R. Gaeta, and M. Grangetto. Exploiting rateless codes in cloud storage systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 26(5):1313–1322, 2015.

[5] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *Proc. of SODA*, pages 1027–1035, New Orleans, LA, USA, January 2007.

[6] M. Atallah, K. Frikken, and M. Blanton. Dynamic and efficient key management for access hierarchies. In *Proc. of CCS*, Alexandria, VA, USA, November 2005.

[7] G. Ateniese, K. Fu, M. Green, and S. Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. *ACM Transactions on Information System Security (TISSEC)*, 9(1):1–30, February 2006.

[8] V. Atluri, S. A. Chun, and P. Mazzoleni. A chinese wall security model for decentralized workflow systems. In *Proc. of CCS*, pages 48–57, Philadelphia, PA, USA, November 2001.

[9] N. Ayeb, F. Di Cerbo, and S. Trabelsi. Enhancing access control trees for cloud computing. In *Proc. of ICWE*, pages 29–38, Lugano, Switzerland, June 2016.

[10] E. Bacis, S. De Capitani di Vimercati, S. Foresti, D. Guttadoro, S. Paraboschi, M. Rosa, P. Samarati, and A. Saullo. Managing data sharing in openstack swift with over-encryption. In *Proc. of WISCS*, Vienna, Austria, October 2016.

[11] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati. Access control management for secure cloud storage. In *Proc. of SECURECOMM*, Guangzhou, China, October 2016.

[12] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati. Mix&Slice: Efficient access revocation in the cloud. In *Proc. of CCS*, Vienna, Austria, October 2016.

[13] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati. Securing resources in decentralized cloud storage. *IEEE Transactions on Information Forensics and Security (TIFS)*, 15(1):286–298, December 2020.

[14] M. Baldi, N. Maturo, E. Montali, and F. Chiaraluce. AONT-LT: A data protection scheme for cloud and cooperative storage systems. In *Proc. of HPCS*, Bologna, Italy, July 2014.

[15] J. Benet. IPFS - content addressed, versioned, P2P file system. Technical report, Protocol Labs, 2014.

[16] M. Björkqvist, C. Cachin, R. Haas, X.-Y. Hu, A. Kurmus, R. Pawlitzek, and M. Vukolić. Design and implementation of a key-lifecycle management system. In *Proc. of the 14th International Conference on Financial Cryptography and Data Security*, Tenerife, Spain, January 2010.

[17] M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. In *Proc. of EUROCRYPT*, Espoo, Finland, May-June 1998.

[18] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *Proc. of S&P*, pages 206–214, Oakland, CA, USA, May 1989.

[19] C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, September 1973.

[20] C. Cadwalladr. 'I made Steve Bannon's psychological warfare tool': meet the data war whistleblower. `https://www.theguardian.com/news/2018/mar/17/data-war-whistleblower-christopher-wylie-faceook-nix-bannon-trump`, 2018.

[21] P. Cataldi, M. P. Shatarski, M. Grangetto, and E. Magli. Implementation and performance evaluation of LT and Raptor codes for multimedia applications. In *Proc. of IEEE MSP*, Pasadena, CA, USA, December 2006.

[22] S. S. M. Chow. A framework of multi-authority attribute-based encryption with outsourcing and revocation. In *Proc. of SACMAT*, Shanghai, China, June 2016.

[23] J. Crampton. Specifying and enforcing constraints in role-based access control. In *Proc. of SACMAT*, pages 43–50, Como, Italy, June 2003.

[24] J. Crampton and H. Khambhammettu. Xacml and role-based access control. In *Proc. of DIMACS Workshop on Security of Web Services and e-Commerce*, page 174, Piscataway, NJ, USA, May 2005.

[25] F. Daryabar, A. Dehghantanha, and K.-K. R. Choo. Cloud storage forensics: Mega as a case study. *Australian Journal of Forensic Sciences*, 49(3):344–357, 2017.

[26] S. De Capitani and Foresti, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Over-encryption: Management of access control evolution on outsourced data. In *Proc. of VLDB*, Vienna, Austria, September 2007.

[27] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, G. Livraga, S. Paraboschi, and P. Samarati. Enforcing dynamic write privileges in data outsourcing. *Computers and Security*, 39:47–63, November 2013.

[28] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, G. Pelosi, and P. Samarati. Encryption-based policy enforcement for cloud storage. In *Proc. of SPCC*, Genova, Italy, June 2010.

[29] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Over-encryption: Management of access control evolution on outsourced data. In *Proc. of VLDB*, Vienna, Austria, September 2007.

[30] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Encryption policies for regulating access to outsourced data. *ACM Transactions on Database Systems (TODS)*, 35(2):1–46, April 2010.

[31] S. De Capitani di Vimercati, S. Foresti, G. Livraga, and P. Samarati. Selective and private access to outsourced data centers. In S. Khan and A. Zomaya, editors, *Handbook on Data Centers*. Springer, 2015.

[32] S. De Capitani di Vimercati, S. Foresti, G. Livraga, and P. Samarati. Practical techniques building on encryption for protecting and managing data in the cloud. In P. Ryan, D. Naccache, and J.-J. Quisquater, editors, *Festschrift for David Kahn*. Springer, 2016.

[33] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Efficient and private access to outsourced data. In *Proc. of ICDCS*, Minneapolis, MN, USA, June 2011.

[34] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Shuffle index: Efficient and private access to outsourced data. *ACM Transactions on Storage (TOS)*, 11(4):19:1–19:55, October 2015.

[35] Dropbox business security. a dropbox whitepaper. `https://cfl.dropboxstatic.com/static/business/resources/dfb_security_whitepaper-vfllunodj.pdf`.

[36] Z. Y. Duan and Y. Z. Cao. The implementation of cloud storage system based on OpenStack Swift. *Applied Mechanics and Materials*, 644, September 2014.

[37] G. Ducatel, J. Daniel, T. Dimitrakos, F. A. El-Moussa, R. Rowlingson, and A. Sajjad. Managed security service distribution model. In *Proc. of CCIS*, Beijing, China, August 2016.

[38] M. Dworkin. Recommendation for block cipher modes of operation, methods and techniques. Technical Report NIST Special Publication 800-38A, National Institute of Standards and Technology, 2001.

[39] D. Easley and J. Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

[40] European Parliament. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural

persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*, L119:1–88, May 2016.

[41] D. Ferraiolo, R. Chandramouli, R. Kuhn, and V. Hu. Extensible access control markup language (xacml) and next generation access control (ngac). In *Proc. of ABAC*, pages 13–24, New Orleans, LA, USA, March 2016.

[42] K. Fu, S. Kamara, and Y. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *Proc. of NDSS*, San Diego, CA, USA, February 2006.

[43] A. Gabillon, M. Munier, J.-J. Bascou, L. Gallon, and E. Bruno. An access control model for tree data structures. In *Proc. of ISC*, pages 117–135, Sao Paulo, Brazil, September-October 2002.

[44] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proc. of CCS*, Alexandria, VA, USA, October–November 2006.

[45] Ö. M. Ilhan, D. Thatmann, and A. Küpper. A performance analysis of the xacml decision process and the impact of caching. In *Proc. of SITIS*, pages 216–223, Bangkok, Thailand, November 2015.

[46] Information regarding security and privacy by design at MEGA. `https://mega.nz/help/client/webclient/security-and-privacy`.

[47] D. Irvine. Maidsafe distributed file system. Technical report, MaidSafe, 2010.

[48] J. Jacobs. The Marx-Geo Dataset. `https://datadrivensecurity.info/blog/pages/dds-dataset-collection.html`, 2014.

[49] S. Jha, S. Sural, V. Atluri, and J. Vaidya. Enforcing separation of duty in attribute based access control systems. In *Proc. of ICISS*, pages 61–78, Kolkata, India, December 2015.

[50] R. Jhawar, V. Piuri, and P. Samarati. Supporting security requirements for resource management in cloud computing. In *Proc. of CSE*, Paphos, Cyprus, December 2012.

[51] X. Jin, R. Krishnan, and R. Sandhu. A unified attribute-based access control model covering dac, mac and rbac. In *Proc. of DBSEC*, pages 41–55, Paris, France, July 2012.

[52] A. Jivanyan, R. Yeghiazaryan, A. Darbinyan, and A. Manukyan. Secure collaboration in public cloud storages. In *Proc. of the CRIWG*, Yerevan, Armenia, September 2015.

[53] Javaswift JOSS. `http://joss.javaswift.org`.

[54] N. Kaaniche, M. Laurent, and M. El Barbori. Cloudasec: A novel public-key based framework to handle data sharing security in clouds. In *Proc. of SECRYPT*, Vienna, Austria, August 2014.

[55] S. Kang, B. Veeravalli, and K. M. M. Aung. ESPRESSO: An encryption as a service for cloud storage systems. In *Proc. of AIMS*. Brno, Czech Republic, June–July 2014.

[56] K. Kapusta and G. Memmi. Circular AON: A very fast scheme to protect encrypted data against key exposure. In *Proc. of CCS*, Toronto, Canada, October 2018.

[57] G. O. Karame, C. Soriente, K. Lichota, and S. Capkun. Securing cloud data under key exposure. *IEEE TCC*, pages 1–13, 2017, pre-print.

[58] F. Keller and S. Wendt. Fmc: An approach towards architecture-centric system development. In *Proc. of ECBS*, pages 173–182, Huntsville, AL, USA, 2003.

[59] P. Labs. Filecoin: A decentralized storage network. Technical report, Protocol Labs, 2017. `http://filecoin.io/filecoin.pdf`.

[60] N. Lambert and B. Bollen. The SAFE network - a new, decentralised internet. Technical report, MaidSafe, 2014.

[61] T.-Y. Lin. Chinese wall security policy-an aggressive model. In *Proc. of ACSAC*, pages 282–289, Tucson, AZ, USA, December 1989.

[62] A. X. Liu, F. Chen, J. Hwang, and T. Xie. Xengine: A fast and scalable xacml policy evaluation engine. In *Proc. of SIGMETRICS*, pages 265–276, June 2008.

[63] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder. RaptorQ forward error correction scheme for object delivery – RFC 6330. *IETF Request For Comments*, 2011.

[64] R. D. Luce and A. D. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, June 1949.

[65] R. Nath, S. Das, S. Sural, J. Vaidya, and V. Atluri. Poltree: A data structure for making efficient access decisions in abac. In *Proc. of SACMAT*, pages 25–35, Toronto, Canada, June 2019.

[66] S. Pina Ros, M. Lischka, and F. Gómez Mármol. Graph-based xacml evaluation. In *Proc. of SACMAT*, pages 83–92, Newark, NJ, USA, June 2012.

[67] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, June 1960.

[68] J. K. Resch and J. S. Plank. AONT-RS: Blending security and performance in dispersed storage systems. In *Proc of FAST*, San Jose, CA, USA, February 2011.

[69] E. Rissanen. Xacml v3. 0 multiple decision profile version 1.0. *Oasis committee specification, Organization for the Advancement of Structured Information Standards (OASIS)*, 2010.

[70] E. Rissanen et al. Extensible access control markup language (xacml) version 3.0. *OASIS standard*, 22, 2013.

[71] R. Rivest. All-or-nothing encryption and the package transform. In *Proc. of FSE*, Haifa, Israel, Jan. 1997.

[72] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. In *Proc. of NSDI*, pages 297–312, San Jose, CA, USA, April 2010.

[73] P. Samarati and S. De Capitani di Vimercati. Cloud security: Issues and concerns. In S. Murugesan and I. Bojanova, editors, *Encyclopedia on Cloud Computing*. Wiley, 2016.

[74] R. S. Sandhu. A lattice interpretation of the chinese wall policy. In *Proc. of NCSC*, pages 329–339, Baltimore, MD, USA, October 1992.

[75] A. Shokrollahi. Raptor codes. *IEEE/ACM Transactions on Networking (TON)*, 6(3-4):213–322, May 2011.

[76] S. Smalley. Configuring the selinux policy. *NAI Labs Rep*, pages 02–007, June 2002.

[77] E. Stefanov, van M. Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple Oblivious RAM protocol. In *Proc. of CCS*, Berlin, Germany, November 2013.

[78] The Economist. The world's most valuable resource is no longer oil, but data. `https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data`, 2017.

[79] S. Trabelsi, A. Ecuyer, P. C. y Alvarez, and F. Di Cerbo. Optimizing access control performance for the cloud. In *Proc. of CLOSER*, pages 551–558, Barcelona, Spain, April 2014.

[80] F. Turkmen and B. Crispo. Performance evaluation of xacml pdp implementations. In *Proc. of SWS*, pages 37–44, Alexandria, VA, USA, October 2008.

[81] H. Ulusoy, P. Colombo, E. Ferrari, M. Kantarcioglu, and E. Pattuk. Guardmr: fine-grained security policy enforcement for mapreduce systems. In *Proc. of ASIACCS*, pages 285–296, Singapore, Republic of Singapore, April 2015.

[82] D. Vorick and L. Champine. Sia: Simple decentralized storage. Technical report, Nebulous Inc., 2014.

[83] C. Wang, N. Cao, K. Ren, and W. Lou. Enabling secure and efficient ranked keyword search over outsourced cloud data. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 23(8):1467–1479, August 2012.

[84] S. Wilkinson, T. Boshevski, J. Brandoff, J. Prestwich, G. Hall, P. Gerbes, P. Hutchins, and C. Pollard. Storj - A peer-to-peer cloud storage network, 2014.

[85] J. Yao, S. Chen, S. Nepal, D. Levy, and J. Zic. Truststore: Making Amazon S3 trustworthy with services composition. In *Proc. of CCGrid*, Melbourne, Australia, May 2010.

[86] S. Yu, C. Wang, K. Ren, and W. Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *Proc. of INFOCOM*, San Diego, CA, USA, March 2010.