Project title: Enforceable Security in the Cloud to Uphold Data Ownership
Project acronym: ESCUDO-CLOUD
Funding scheme: H2020-ICT-2014
Topic: ICT-07-2014
Project duration: January 2015 – December 2017

D2.3
Report on Data and Access Protection

Editors: Stefano Paraboschi (UNIBG)
Christian Cachin (IBM)
Enrico Bacis (UNIBG)
Marco Rosa (UNIBG)
Reviewers: Daniel Bernau (SAP)
Sabine Delaitre (WT)

Abstract

The overall goal of the ESCUDO-CLOUD project is to advance the state of the art for the protection of data stored in the cloud. The project considers a variety of scenarios and the goal of Work Package 2 is to focus on the realization of the basic protection services, which represent the foundation for the management of the advanced data sharing and multi-provider cooperation services considered in the other Work Packages. This deliverable reports on the work that has been done in the second year of the project within Work Package 2. The analysis focuses on the main contributions by the work in the second year of the project with respect to Task T2.1 “Protection of data at rest”, Task T2.2 “Key management solutions”, and Task T2.3 “Efficient and private data access”.

Type	Identifier	Dissemination	Date
Deliverable	D2.3	Public	2016.12.30

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 644579. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract No 150087. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission or the Swiss Government.

ESCUDO-CLOUD Consortium

1.	Università degli Studi di Milano	UNIMI	Italy
2.	British Telecom	BT	United Kingdom
3.	EMC Corporation	EMC	Ireland
4.	IBM Research GmbH	IBM	Switzerland
5.	SAP SE	SAP	Germany
6.	Technische Universität Darmstadt	TUD	Germany
7.	Università degli Studi di Bergamo	UNIBG	Italy
8.	Wellness Telecom	WT	Spain

Versions

Version	Date	Description
0.1	2016.11.30	Initial Release
0.2	2016.12.14	Second Release
1.0	2016.12.30	Final Release

List of Contributors

This document contains contributions from different ESCUDO-CLOUD partners. Contributors for the chapters of this deliverable are presented in the following table.

Chapter	Author(s)
Executive Summary	Stefano Paraboschi (UNIBG), Enrico Bacis (UNIBG), Marco Rosa (UNIBG)
Chapter 1: Introduction	Stefano Paraboschi (UNIBG), Enrico Bacis (UNIBG), Marco Rosa (UNIBG)
Chapter 2: Secure Cloud Storage	Enrico Bacis (UNIBG), Sabrina De Capitani di Vimercati (UNIMI), Sara Foresti (UNIMI), Stefano Paraboschi (UNIBG), Marco Rosa (UNIBG), Pierangela Samarati (UNIMI)
Chapter 3: Protection of Access Confidentiality	Sabrina De Capitani di Vimercati (UNIMI), Sara Foresti (UNIMI), Riccardo Moretti (UNIMI), Stefano Paraboschi (UNIBG), Gerardo Pelosi, Pierangela Samarati (UNIMI)
Chapter 4: Scalable Distributed Key Management for Cloud Storage	Mathias Björkqvist (IBM), Christian Cachin (IBM), Felix Engelmann (IBM), Alessandro Sorniotti (IBM)
Chapter 5: Conclusions	Stefano Paraboschi (UNIBG), Enrico Bacis (UNIBG), Marco Rosa (UNIBG)

Contents

Executive Summary **11**

1 Introduction **13**

2 Secure Cloud Storage **15**
2.1 State of the Art . . . 15
2.2 ESCUDO-CLOUD Innovation . . . 16
2.3 Basic Concepts . . . 17
2.4 Access Control Enforcement in Swift . . . 18
2.4.1 Keys and User-Based Repositories . . . 18
2.4.2 Policy-Based Encryption . . . 20
2.5 Policy Updates . . . 21
2.5.1 Enforcement of Policy Updates . . . 21
2.5.2 Implementation of Over-Encryption . . . 23
2.6 Experimental Results . . . 25
2.6.1 Comparison Between Client Re-Encryption and Over-Encryption . . . 26
2.6.2 Analysis of Over-Encryption Approaches . . . 26
2.6.3 Streaming and Batch Encryption . . . 28
2.6.4 Application of Two Encryption Layers . . . 28

3 Protection of Access Confidentiality **31**
3.1 State of the Art . . . 31
3.2 ESCUDO-CLOUD Innovation . . . 32
3.3 Overview of the Approach . . . 33
3.4 Data Organization and Storage . . . 33
3.5 Uniform Accesses . . . 35
3.6 Target Bubbling . . . 37
3.7 Speculative Rotations . . . 38
3.8 Physical Re-allocation . . . 40
3.9 Analysis and Experimental Evaluation . . . 41

4 Scalable Distributed Key Management for Cloud Storage **44**
4.1 State of the Art . . . 44
4.2 ESCUDO-CLOUD Innovation . . . 45
4.3 Security Model . . . 46
4.4 Objectives . . . 46
4.5 Related Work . . . 47
4.6 Design . . . 48

4.6.1 File Formats 48
4.6.2 System Interactions 50
4.7 Evaluation 52
4.7.1 Scaling 53
4.7.2 Latency 54
4.7.3 Consistency 56
4.8 Outlook 56
4.8.1 Per Node Asymmetric Key-Pair 56
4.8.2 Protection of Keys in Memory 57
4.8.3 Access Control 57
4.9 Final remarks 57

5 Conclusions **58**

Bibliography **59**

List of Figures

2.1 An example of authorization policy defined by user Alice (a) and corresponding policy-based encryption (b) . . . 18
2.2 Policy-based encryption $\mathscr{E}_A$ equivalent to the authorization policy $\mathscr{A}_A$ in Figure 2.1(a) 21
2.3 An example of implementation of a revoke operation using immediate (a), on-the-fly (b), and opportunistic (c) over-encryption . . . 22
2.4 An example of insertion of an object into an over-encrypted container . . . 23
2.5 Overhead of all the solutions . . . 26
2.6 Cumulative server work with different over-encryption approaches . . . 26
2.7 Comparison of the overhead caused by Streaming and Batch on-the-fly approaches with respect to the direct `get` call . . . 28
2.8 BEL+SEL encryption performance on a 1MB file using two subsequent AES invocations and TWOAES . . . 28
2.9 AES encryption rate for the modes *ECB*, *CBC*, and *CTR* using the *pycrypto* library without and with AES-NI . . . 29
2.10 Re-encryption using AES . . . 29
2.11 Re-encryption using AES-NI . . . 30

3.1 Data structure construction and its physical representation . . . 34
3.2 Two sample accesses . . . 36
3.3 Tree rotations . . . 37
3.4 Nodes downloaded to access U and rotations performed to bubble up the target (a), tree with the target bubbled to the root and speculative rotations (b), and the resulting tree (c) . . . 38
3.5 An example of physical re-allocation (a) and of view of the server before (b) and after (c) the access in Figure 3.4 . . . 40
3.6 Average height of a tree with 256 nodes, considering $500,000$ accesses . . . 42
3.7 Average length of the maximum common prefix among the paths reaching the same target (a) and rank/frequency distributions of the block identifiers corresponding to self-similar access profiles with $\gamma \in \{0.5, 0.2, 0.1\}$ when only the physical re-allocation (b), and when all protection techniques are applied (c) . . . 43

4.1 REK Header . . . 49
4.2 MEK structure . . . 50
4.3 Key manager components . . . 51
4.4 Scale up with number of clients . . . 53
4.5 Proportional scale-out effect . . . 54

4.6 TLS handshake timings 55

List of Tables

4.1 Processing times in milliseconds of queries to CCR and its components. 56

Executive Summary

The protection of outsourced data is a problem which needs the design and deployment of advanced technical solutions. With current techniques on one hand we have a significant exposure to many significant security risks of the data that is stored on cloud infrastructures. On the other hand, the unavailability of robust and efficient techniques limits the adoption of cloud technology and reduces the benefits that the Cloud can bring to many IT scenarios. The crucial aspect associated with the use of Cloud Service Providers (CSPs) is the loss of control over the data and applications imposed on the data owners. Encryption permits to mitigate this loss of control, but an effective use of encryption in this scenario requires the definition of adequate conceptual models and technical implementations. The goal of ESCUDO-CLOUD is to support this evolution.

The goal of this deliverable is to report on the research work done within Work Package 2 in the second year of the project. The tasks considered in this deliverable are Task T2.1 "Protection of data at rest", Task T2.2 "Key management solutions", and Task 2.3 "Efficient and private data access". The work in Task T2.4 "Requirements-based Threat Analysis" will be described in Deliverable D2.4, which is currently being prepared and will be ready at M27. This deliverable reports on three separate research investigations, each presented in a dedicated chapter.

Chapter 2 reports on the realization of an extension of current cloud storage services with the introduction of a layer of encryption transparent to applications. The chapter focuses on the analysis of key management techniques that allow efficient application of an access control policy on the objects. The investigation also considers different approaches for management of the evolution of encryption, with an emphasis on the consideration of alternatives for the application of over-encryption. The tool implemented in ESCUDO-CLOUD WP2 and described in D2.2 has been used as a platform for the experimental evaluation of the performance of the alternatives, in a variety of scenarios. The experiments are associated with an evaluation that justifies the observed performance and provides indications about when a given approach should be preferred.

Chapter 3 is the result of the work on access privacy and describes an alternative to the shuffle index, which had been presented in Deliverable D2.1 and was also investigated in Work Package 3 and Work Package 4. The structure proposed here is based on an encrypted binary tree (compared to the encrypted B-tree which is the static structure of the shuffle index). Each access to the binary tree is then managed by applying a randomly chosen reorganization of the data structure, which aims at hiding the access against a server that monitors the sequence of physical access requests. The source of the protection is then the realization of a reorganization of the tree both at the physical and at the logical level.

Furthermore, Chapter 4 addresses scalable key management that aims at supporting the workload from a production-grade cloud storage system with millions of keys and thousands of clients. This work was done in the context of Task 2.2 (Key-management solutions). The report describes the realization of a key manager, which uses an untrusted distributed key value store (KVS) for consistent key distribution over the Key-Management Interoperability Protocol (KMIP). To

achieve confidentiality, it uses a key hierarchy where every key except a root key itself is encrypted by the respective parent key. The hierarchy also allows for key rotation and, ultimately, secure deletion of data.

1. Introduction

The deliverable reports on the research that was done within Work Package 2 in ESCUDO-CLOUD. The Work Package is organized into four tasks and the work reported in this deliverable considers tasks T2.1, T2.2, and T2.3. The work executed within Task 2.4 "Requirements-based Threat Analysis" will be described in Deliverable D2.4, which is currently being prepared and will be released at M27.

This deliverable contains three core chapters, each dedicated to the presentation of a specific investigation on a topic associated with the management of encrypted outsourced data. The report does not provide an exhaustive representation of the research done within tasks T2.1, T2.2 and T2.3. A full coverage will be obtained at the end of the project with the preparation of the deliverables of the third year. Some work is going to be reported in Deliverable D2.5, dedicated to the construction of the tools implementing the techniques for the protection of cloud storage. Other research lines on data protection for outsourced data that are being investigated will be described in the final deliverable D2.6 due at M34.

Chapter 2 focuses on the investigation of techniques that combine object protection with the management of the evolution of the access control policy. This topic is centered on Work Package 2, due to the emphasis on the application of an encryption layer to objects in order to protect their confidentiality even against the server hosting them. Also, key management is crucial to the realization of these services. These aspects are integrated with the management of the access control policy and the realization of the sharing of objects, which is investigated in Work Package 3. With the widespread success and adoption of cloud-based solutions, we are witnessing an ever increasing reliance on external providers for storing and managing data. This evolution is greatly facilitated by the availability of solutions - typically based on encryption - ensuring the confidentiality of externally outsourced data against the storing provider itself. Selective application of encryption (i.e., with different keys depending on the authorizations holding on data) provides a convenient approach to access control policy enforcement. Effective realization of such policy-based encryption entails addressing several problems related to key management, access control enforcement, and authorization revocation, while ensuring efficiency of access and deployment with current technology. We present the design and implementation of an approach to realize policy-based encryption for enforcing access control in OpenStack Swift. The work is related with the tool that has been described in Deliverable D2.2. The availability of the implementation described in D2.2 allowed us to report experimental results evaluating and comparing different implementation choices of our approach.

Chapter 3 presents a novel approach for guaranteeing access privacy to data stored at an external cloud provider. The solution relies on the grouping of resources into buckets then organized with a binary search tree. The tree is built on an index computed in a non-invertible non-order preserving way, and supports efficient key-based retrieval. The approach to provide access privacy builds on this data organization offering uniform observability to the server in access execution and dynamically changing not only the physical storage allocation, but also the logical structure itself.

The analysis and experimental evaluation show the effectiveness of our approach. This structure is an alternative to the shuffle index, which had been presented in Deliverable D2.1 and has been the subject of extensive investigation in the other ESCUDO-CLOUD technical Work Packages. As it will be discussed in the chapter, this tree structure has the benefit that it offers protection operating at a different level. This leads to a simplification of the structure that must be used by the client to access the data, reducing the need to have local cache in the client. The benefit is a more direct adaptation of this technique to scenarios where there are multiple clients accessing the same tree, without the need to introduce a centralized proxy service that mediates all requests, as done by many other approaches for access privacy.

Chapter 4 describes a scalable key manager, which uses an untrusted distributed key-value store (KVS) and is accessible over the Key-Management Interoperability Protocol (KMIP). It implements a key hierarchy and focuses on providing scalable performance that is suitable to serve keys at very high rate. Specifically, a prototype has been integrated with IBM GPFS, a highly scalable cluster file system, where it serves keys for file encryption. Linear scale-out is achieved even under load from key updates.

The outline of this Deliverable is the following. Chapter 2 presents the work on the realization of advanced approaches for key management in the cloud storage of client-encrypted objects. The experiments refer to the use of the ESCUDO-CLOUD implementation in Swift, but the techniques can be adapated to any domain where objects are stored in an encrypted format. Chapter 3 considers the topic of access privacy, which is investigated in Task T2.3, and presents a novel approach based on encrypted binary trees and the application of rotations to hide the access pattern. Chapter 4 illustrates the work on the realization of a scalable key management service. Finally, Chapter 5 draws a few concluding remarks.

2. Secure Cloud Storage

The ESCUDO-CLOUD project aims at protecting data stored in the cloud with the use of encryption. An important benefit provided by data encryption is that it enables effective enforcement of access control. In fact, data can be encrypted with different keys, depending on the authorizations holding on them, and keys shared with users according to authorization (*policy-based encryption* [14]). This policy-based encryption translates the access control policy into an equivalent *encryption policy* which provides self-protection and effective access control enforcement on the outsourced data.

One of the complicating aspects in the management of policy-based encryption relates to the enforcement of possible changes to the access control policy, and in particular revocation of authorizations. When resource maintenance is decoupled from access control thanks to the use of encryption, revocation cannot be simply managed by dropping access to the encryption key (as done in other scenarios). The revoked user can, in fact, have maintained local copies of the keys, and if the layer of protection is not refreshed, the user could still be able to pass the encryption wrap and access objects for which she does not have authorization anymore. On the other hand, changing the key and re-encrypting objects affected by revocation would entail download and re-upload operations by owners, which could become cumbersome and affect the performance of the system. The solution that was proposed to this problem in [14] introduces *over-encryption*, based on the application by the server of an additional layer of encryption (operating on the object already encrypted by the data owner) with a key not accessible by the revoked user, thus adapting the encryption on objects to the new state of the access control policy.

2.1 State of the Art

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., [42]). Techniques have been designed that let the data be available only to users with specific properties (e.g., ABE [10, 25]). Another important line of research focuses on protecting access privacy (e.g., [18, 18, 39]). In this deliverable, we focus the analysis of over-encryption, on the approaches for existing cloud storage frameworks, and on proposals for the sharing of large client-encrypted objects (instead of structured data).

Over-encryption has been proposed to effectively and efficiently enforce policy updates over encrypted outsourced data [14, 15]. 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 [13]. 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.

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 [31] 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 [45], 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 [30], an architecture for sharing encrypted objects outsourced to a Cloud Service 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 [46], 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 being able to access the plaintext of the resource. Proxy re-encryption supports expressive encryption schemes, which allow attribute-based selection. In contrast, 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 within a given 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.

The OpenStack Swift community is making a significant effort toward the introduction of object encryption in Swift [40]. This initiative saw the direct support by the ESCUDO-CLOUD project within UC1. The support is offered for the server side, aiming at protecting data at rest. There is a plan to adapt the solution presented in this chapter to this implementation, within the work that will be reported in Deliverable D2.5 at M27.

2.2 ESCUDO-CLOUD Innovation

Policy-based encryption for providing self-enforcement of the access control policy and over-encryption for supporting policy changes result particularly appealing and promising. However, their integration and deployment in available cloud storage systems requires addressing several additional issues, including: the support for co-existence of several data owners in a single system, the realization of key management solutions to enable users to access keys used for objects for which they have authorizations, and the implementation of policy-based encryption and over-encryption functionality with services supported by the cloud service providers.

This chapter reports how ESCUDO-CLOUD innovated over all these directions, illustrating the realization of policy-based encryption and over-encryption in the context of OpenStack Swift.

OpenStack [36] represents today the reference platform for the cloud [43], and is receiving significant attention by the industrial community, and Swift is the OpenStack's object storage system. Swift exhibits features that are shared by most object storage solutions for the cloud (e.g., Amazon S3).

In this chapter, we illustrate how the work of ESCUDO-CLOUD led to a system where policy-based encryption can be applied over the OpenStack Swift module. Policy changes are enforced implementing over-encryption in Swift. For over-encryption, in particular, we investigate different implementation alternatives, which can be suitable for different scenarios, depending on the frequency of access requests and policy changes.

Furthermore, there are two main concrete contributions to innovation of ESCUDO-CLOUD in this work. First, it provides an effective design and implementation of policy-based encryption and over-encryption, which can be adopted by others for immediate deployment in current cloud storage solutions. Second, our extensive experimental evaluation of different design choices can provide precious observations for such adoption, enabling the tuning of the implementation depending on the characteristics of the considered scenario.

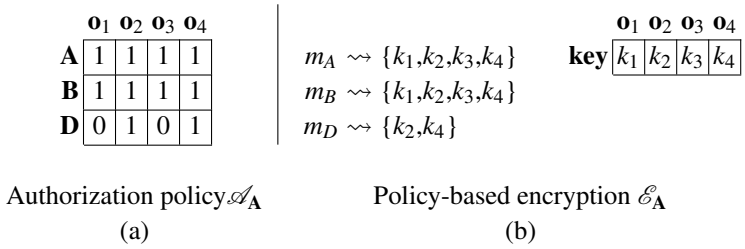
2.3 Basic Concepts

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.

We assume that the CSP is based on the OpenStack framework, which includes the Swift module, an object storage service allowing users to store and access data in the form of *objects* (i.e., each resource, such as a file, uploaded on Swift is an object). Swift organizes objects in *containers*, which are user-defined storage areas containing sets of objects. Containers are organized in *tenants*, which are sets of containers. Each tenant is usually assigned to an organization. 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. Keystone is an OpenStack component acting as identity server, which provides a central directory of users mapped to the OpenStack services they can access.

We assume the CSP 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 [12]. In the following, $acl(o)$ denotes the read access control list of object o and $\mathscr{A}_u$ 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., $u \notin acl(o_i)$).

Our work is based on the policy-based encryption and over-encryption approach proposed in [14, 15], and focuses on their representation and enforcement with Swift, which also requires some re-definition and adjustment of these concepts. Essentially, each user is associated with a



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

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

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

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., $u \in acl(o_i)$). Policy updates, which would require re-encryption of an object, are enforced by super-imposing a second layer of encryption on the encrypted object itself. 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 $\mathscr{E}_u$ to denote the policy-based encryption equivalent to the authorization policy $\mathscr{A}_u$ 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 2.4), and then illustrate how to enforce policy updates (Section 2.5).

2.4 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.

2.4.1 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 [14, 15].

We describe the different keys, 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 associated 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. Intuitively, a KEK is the encryption of a DEK that a user can extract using a secret (key) that only she knows. 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 $\mathscr{R}_u$. 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 into 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 2.5, 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.

2.4.2 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 $\mathscr{A}_u$ defined by user u is translated into an equivalent policy-based encryption $\mathscr{E}_u$ 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 $\mathscr{R}_u$. 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 it using ss_u, thus producing an asymmetric KEK usable by u_j. This KEK is stored in u_j's repository $\mathscr{R}_{u_j}$.

Example 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 $\mathscr{R}_A$. 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 $\mathscr{R}_B$ and $\mathscr{R}_D$, 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 $\mathscr{R}_{u_j}$ 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 $\mathscr{R}_B$ the corresponding KEK, decrypts it

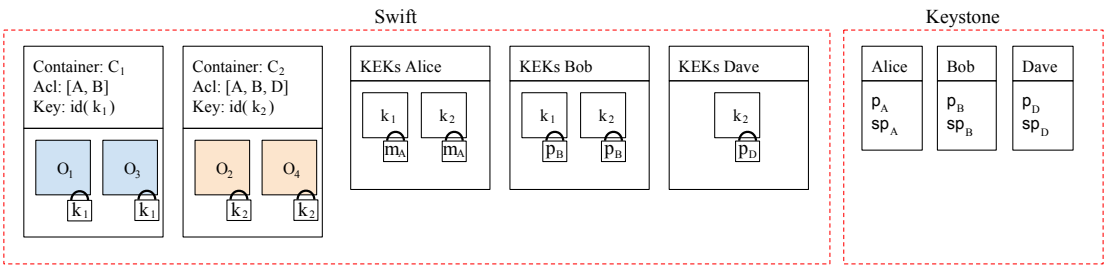


Figure 2.2: Policy-based encryption $\mathscr{E}_A$ equivalent to the authorization policy $\mathscr{A}_A$ in Figure 2.1(a)

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 $\mathscr{R}_u$ 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.

2.5 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).

In the remainder of this section, we first illustrate how policy updates can be realized, and then discuss different alternatives for the practical implementation in Swift of the over-encryption requested for their enforcement.

2.5.1 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.

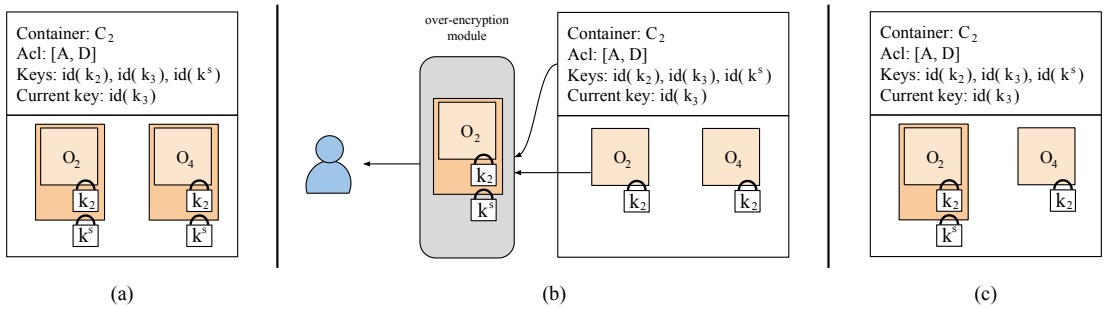


Figure 2.3: An example of implementation of a revoke operation using immediate (a), on-the-fly (b), and opportunistic (c) over-encryption

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 $\mathscr{R}_{u_i}$ 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 (Section 2.3). 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(a), 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 $\mathscr{R}_B$, while the KEKs enabling Alice and Dave to compute k^s are created and inserted into $\mathscr{R}_A$ and $\mathscr{R}_D$, 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

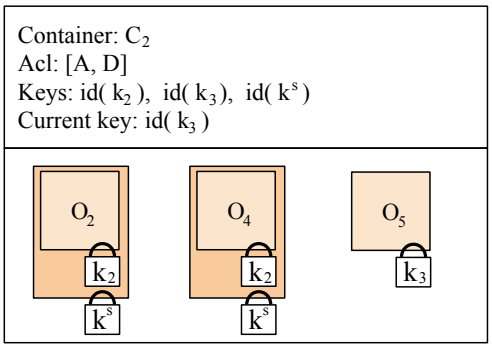


Figure 2.4: An example of insertion of an object into an over-encrypted container

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 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(a), 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.

2.5.2 Implementation of Over-Encryption

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 2.3(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 lays 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 2.3(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 2.6), 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 encryp-

tion 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 2.3(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 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.

2.6 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 2.6.1);
- the performance of the immediate, on-the-fly, and opportunistic options (Section 2.6.2);
- the performance of a batch and a streaming option for the execution of encryption by the server (Section 2.6.3);

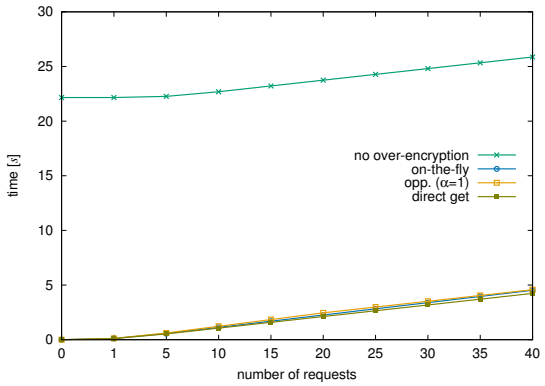


Figure 2.5: Overhead of all the solutions

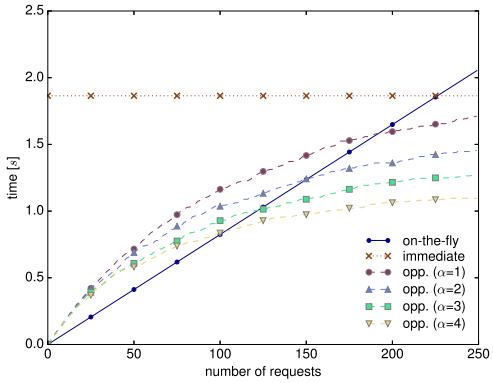


Figure 2.6: Cumulative server work with different over-encryption approaches

- the performance at the client-side for the removal of the two encryption layers for over-encrypted objects (Section 2.6.4).

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

2.6.1 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 2.5 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.

2.6.2 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 2.5). Figure 2.6 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 2.6 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 2.6 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 [24] with varying values for the α parameter, from 1 to 4. A value of α 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 α 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.

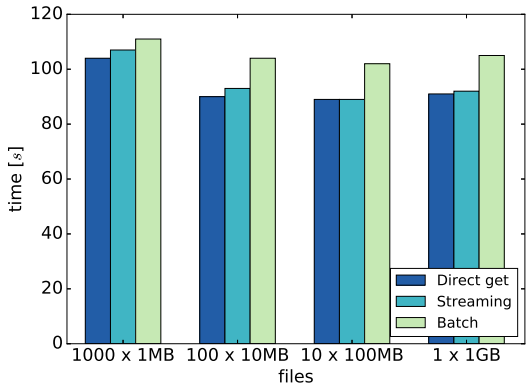


Figure 2.7: Comparison of the overhead caused by Streaming and Batch on-the-fly approaches with respect to the direct `get` call

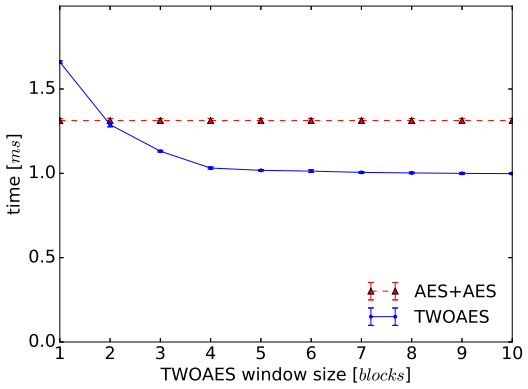


Figure 2.8: BEL+SEL encryption performance on a 1MB file using two subsequent AES invocations and TWOAES

2.6.3 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 2.7, 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.

2.6.4 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. In particular, we verified the impact of the AES-NI (Intel AES New Instruction

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

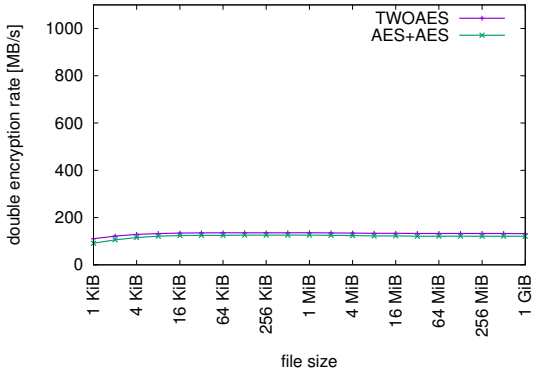


Figure 2.10: Re-encryption using AES

set) instructions available on Intel processors. A first set of experiments, reported in Figure 2.9, 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 2.8, 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 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 2.10 and 2.11 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 2.10 indeed shows that the difference between the two approaches when hardware acceleration is not used is limited. Figure 2.11 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

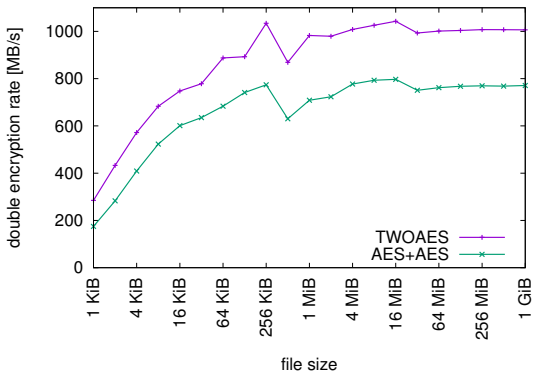


Figure 2.11: Re-encryption using AES-NI

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.

3. Protection of Access Confidentiality

Traditionally addressed within the line of work of Private Information Retrieval (PIR) [34, 9] (known to suffer from high computational complexity), access confidentiality has been recently addressed by several researchers aiming at more practical solutions, limiting computational overhead and providing effective key-based retrieval capabilities. Among them, the more recent ORAM-based solutions (e.g., [38, 39]) and the shuffle index [18]. A common aspect of these approaches is the idea of breaking the otherwise static correspondence between data and the physical locations where they are stored.

In this chapter, we propose a novel approach to provide access privacy. Our solution groups resources in buckets according to a randomly and non-invertible mapping and associates non-order preserving indexes with buckets, then organizing bucket indexes in a binary search tree. Such bucketization and indexing provide fine support for key-based retrieval while protecting confidentiality of original index values and their relationships. Like previous works, our approach dynamically changes the allocation of buckets (i.e., nodes of the tree) to physical blocks at every access, so to destroy the correspondence between data and physical locations. In addition to this, our approach protects confidentiality by making accesses all look alike from the point of view of the server, and continuously changing the logical organization of data themselves.

3.1 State of the Art

With the increasing interest in data outsourcing, many proposals have first been devoted to the protection (of the confidentiality) of data in storage. Outsourced data are protected by wrapping a layer of encryption around them, and query evaluation is supported through indexes (i.e., metadata complementing the outsourced encrypted data) or specific cryptographic techniques for keyword-based searches (e.g., [27, 42]). Recently, significant attention has been given to the problem of protecting confidentiality of accesses. Current proposals are based on Private Information Retrieval (PIR) techniques or on dynamically allocated data structures, which change the physical location where data are stored at each access (e.g., [34, 9, 38, 39, 18, 16, 17, 19, 11, 32, 44]). PIR solutions are computationally expensive and do not protect content confidentiality (e.g., [34, 9]).

Dynamic data structures rely on the Oblivious RAM (ORAM) for protecting content, access, and pattern confidentiality (e.g., [38, 39, 11, 44]), or on tree-based structures (e.g., [18, 16, 17, 19, 32]). While preliminary ORAM-based proposals suffer from high computational and communication overheads, recent attempts make ORAM more practical in real-world scenarios (e.g., ObliviStore [38] and Path ORAM [39]), as illustrated in [6] where different ORAM-based approaches are compared.

Path ORAM based solutions [39] store data both at the server side and in a local cache (stash) at the client side. The client also stores a position map (with size proportional to the number of data blocks) that keeps track of where the data are stored. To reduce to one block the storage at

the client, recent proposals move the stash from client to server and store the position map recursively on the server in smaller ORAMs. These approaches, however, cause an increase in response time and a bandwidth blowup of over two orders of magnitude in data exchange between client and server [35]. To reach a constant bandwidth blowup, an additively homomorphic encryption construction can be used to perform server computations (i.e., $O(\log^4 N)$, where N is the number of outsourced data blocks), but at the cost of an increased computational effort for the client (i.e., $O(log^2 N)$) [20].

Our approach applies only efficient symmetric encryption primitives and has limited bandwidth blowup and client storage capacity ($O(logN)$ blocks) as well as lower computational requirements ($O(\log N)$) at the client side.

Solutions that rely on tree-based data structures provide a good trade-off between privacy and performance (e.g., [18, 16, 17, 19, 32]). Among them, the shuffle index has first been proposed in [16]. A shuffle index is a dynamically allocated B+-tree offering access and pattern confidentiality, while supporting efficient key-based data organization and retrieval. To protect access and pattern confidentiality, the B+-tree is stored at the server side in encrypted form and jointly uses cover searches (fake searches indistinguishable from actual searches, executed in parallel), cache (most recent visited target paths), and shuffling for protecting the confidentiality of accesses (corresponding to physical re-allocation). The shuffle index has then been extended to support concurrent accesses by different users [17], to operate in a distributed scenario characterized by the presence of multiple (three) storage servers [19], and to support insertion and removal of tuples in the outsourced relation [18]. The main differences between the shuffle index and our solution is that they are based on different protection techniques and, in particular, the shuffle index does not change the logical tree structure but relies mainly on shuffling. Also, our proposal does not require any client-side storage.

3.2 ESCUDO-CLOUD Innovation

The main innovation of this approach compared to previous solutions is that it does not require to store data at the client. Both the shuffle index [18] and ORAM-based solutions [38, 39] require instead to maintain a local cache and a local map and stash, respectively. We note that while ORAM-based solutions allow also the storage of the local map and stash at the server side, this solution yields a bandwidth blowup of at least two orders of magnitude compared with an unprotected solution [35].

Besides not requiring the client to commit storage, being stateless for the client, our approach supports access by multiple clients. Compared with the shuffle index, in addition to dynamically changing physical location of data (as the shuffling does), we also change the logical structure adding a further level of confusion with respect to observables by the server.

Compared with ORAM [38, 39, 35], in addition to providing good reliability guarantees (being resilient to client failures, as all resources are always stored at the server in a complete and consistent way), we enjoy satisfactory performance figures. Considering that the main service provided by data outsourcing applications is the durable and reliable storage of data, our approach keeps the state of the system safely stored on the remote server. Hence, our solution fits well within the replication, backup, and migration mechanisms adopted by any storage back-end application to cope with software or hardware failures during the system lifetime.

3.3 Overview of the Approach

The goal is to protect access privacy against any possible observer. Since the most powerful observer is the storing server itself, without loss of generality, we assume the server as the observer. Our approach to provide access privacy is based on a combination of techniques that avoid causing, in access execution, observables that can be exploited by the server to learn information about the access. A first level of protection is therefore represented by our storage organization, which provides key-based retrieval functionality, while leaving content not intelligible to the server. Our storage organizes data in buckets, then defines an index key and a binary search tree for it to support efficient retrieval without exposing any information on the original values. Content is encrypted client-side before upload. Hence, the server receives from the data owner a set of encrypted blocks to store, and serves requests accessing them. The application of encryption and the fixed size of the blocks ensure confidentiality in storage of the blocks *content* with respect to the server.

Like other works in this area [18], our approach makes the data structure dynamic (re-allocating nodes in blocks at every access) to destroy the otherwise static correspondence between nodes and blocks where they are stored. In addition to this, we also re-arrange the tree structure itself, thus introducing a further level of protection, and making every access to the data structure uniform (independently of where the actual target of a search is located in the tree). The building blocks of our solution are as follows.

Uniform accesses (Section 3.5). All accesses download from the server the same (constant) number of blocks, regardless of where the target is located. Blocks not pertaining to the target path are not recognizable as such. This permits to hide the block storing the target among all the accessed blocks.

Target bubbling (Section 3.6). At every access, the target node is moved up in the data structure by means of rotations, also causing a re-arrangement of the data structure. The main motivation for bringing the target up in the tree with re-arrangement of the data structure is that a subsequent (or close) search for the same node will not follow the same path in the tree.

Speculative rotations (Section 3.7). At every access, possible re-arrangements (rotations) can be enforced at the logical level that control the depth of the (sub-)trees. The reason for this is to maintain the height of the tree to be at most twice the height of the balanced tree, that is, $2\lfloor\log(|N|)\rfloor$. While not a protection technique per se, rotations also bring protection benefits, since they cause a change in the topology of the tree structure.

Physical re-allocation (Section 3.8). At every access, all the accessed nodes are allocated to different physical blocks. Re-allocation also entails re-encrypting nodes with a nonce (i.e., an always distinct random salt) so to make the nodes indistinguishable and re-allocation not traceable.

3.4 Data Organization and Storage

The data outsourced are assumed to be generic resources identified by an index value for the search. For instance, with reference to a relational database, resources are tuples in a relation and the index is the primary key of the relation. For outsourcing, we organize data in a binary search tree. To support efficient key-based (traversal and) retrieval while protecting the ordering among index values: *i)* each node is a *bucket* containing up to Z real resources, and *ii)* index values are mapped into *bucket indexes* (with which the tree is organized) in a *non-order preserving* way. Bucketization permits to limit the height of the tree, and therefore the length of search paths. Non-

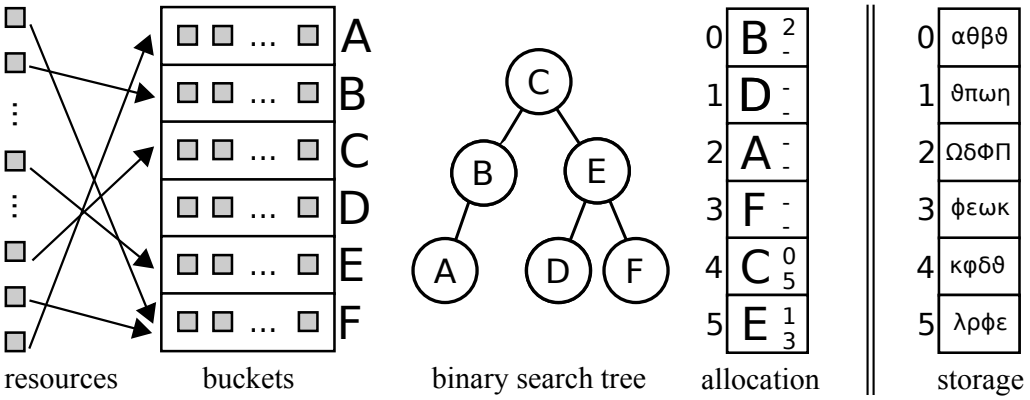


Figure 3.1: Data structure construction and its physical representation

order preserving mapping protects the order relationship among the content of the nodes involved in a tree traversal (which could otherwise be leaked). We do not make any assumption on the mapping of original indexes to bucket indexes as long as such mapping: *i)* is *non invertible* (to avoid reconstructing the original index knowing the bucket index), *ii)* does *not* require storing any *explicit map* at the client (i.e., it is simply a function that can be computed at run-time), *iii)* is *not order-preserving* (so to protect the order relationship among original indexes), and *iv)* resources are *well distributed* among the nodes (to have all nodes indistinguishably of the same size, nodes with fewer resources are padded with dummies, whose occurrences should be kept limited). A simple approach to provide such a mapping consists in applying a pseudo-random function on the original index values and mapping to the same bucket the pseudo-random values with the same value for a given number of most significant bits.

At the physical level, each node is stored in a physical block in encrypted form. The allocation function $\phi : N \rightarrow ID$ randomly maps each node to the identifier of the physical block where it is stored. Pointers between nodes are represented, at the physical level, by storing in each internal node the identifiers of the blocks storing its children. The content of a block storing a node is obtained by first encrypting the concatenation of the node's content with a random salt to destroy plaintext distinguishability, and then concatenating the result with the output of a MAC (Message Authentication Code) function applied to the encrypted node and the block identifier. Formally, the block content is computed as b=$Enc||Token$, with Enc=$E(salt||n,k_e)$ and $Token$=$MAC(id||Enc,k_m)$ where E is a symmetric encryption function with key k_e, *salt* a randomly chosen salt, and *MAC* a strongly unforgeable keyed cryptographic hash function with key k_m. In this way, the client can assess the authenticity of the node returned by the server as well as of the whole data structure, thanks to the presence of pointers to children in each internal node.

Figure 3.1 summarizes the data structure construction (bucketization, tree definition, allocation) and its physical representation. At initialization time, we assume the tree to be balanced, and hence with height $\lfloor \log(|N|) \rfloor$. In the following, we use the term node to refer to an abstract data content and block to refer to a specific memory slot in the physical structure. When either terms can be used, we will use them interchangeably. Having noted that each node in the binary search tree contains several resources and the ordering of indexes of the tree does not leak any information on the ordering among original index values, from now on we will explain our techniques with reference to the binary search tree and its index.

3.5 Uniform Accesses

Without the application of any protection technique, access execution and server's observations would be as follows. To access the node (which from now on we will call *target*) containing a sought value, the client performs an iterative process, retrieving first the block containing the root node, and iteratively determining the child to retrieve the next node until the target one is reached. The observation of the server would then be a sequence of requests of block downloads. Serving an access, the server can observe the blocks in the path to the target and the block storing the target (which is the last one downloaded). Since node indexes and their parent-child relationship do not convey any information on the original index values (or their relationship), a path observation does not cause a problem per se. However, accumulating exact knowledge on target nodes and observing multiple searches, the server could observe or infer possible access patterns, as well as - combining observations with possible knowledge of frequencies of accesses to real values - eventually breach access (and even content) privacy.

The first level of our protection aims at preventing the server to observe (and accumulate) exact information on the target of a search. To this end, we make accesses all look alike from the point of view of the server, and perform searches in the tree always accessing the same number of nodes (and hence blocks at the server), regardless of where the target is located in the tree, be it the root or the deepest leaf. Setting the (constant) number of nodes to be accessed at every search, we need to ensure it is sufficient to reach any target, that is, it can cover the longest path in the tree. In a search tree, the number of nodes in the longest path can go from a minimum of $\lfloor \log(|N|) \rfloor + 1$ (balanced tree) to a maximum of $|N|$ (tree degenerated in a list). Aiming at balancing some degree of freedom in the data structure (which we dynamically re-arrange at every access) while avoiding degeneration, we set a limit on the height of the tree to be at most $2\lfloor \log(|N|) \rfloor$ (Section 3.7 illustrates enforcement of such a limit), which is a well recognized performance trade-off between the height of a perfectly balanced tree ($\lfloor \log(|N|) \rfloor$) and the amortized height of an unbalanced tree with the target bubbling mechanism in place ($3\lfloor \log(|N|) \rfloor - 2$) [37]. The longest path in our data structure can therefore require to access at most $2\lfloor \log(|N|) \rfloor + 1$ nodes. Also, we assume the children of the root to always be read. Hence, we set the constant number of nodes to be read at every access to $2\lfloor \log(|N|) \rfloor + 2$. If, as it will typically be the case, the number of nodes in the path to the target plus the other child of the root (meaning the one not in the path to the target) do not reach $2\lfloor \log(|N|) \rfloor + 2$, we complement the access with other nodes, which we call *fillers*. (The reason for assuming both children of the root to be always read is to accommodate flexibility in the choice of indistinguishable fillers.) Every access request will always be translated into a sequence $A = \langle n_1, \ldots, n_m \rangle$, with $m = 2\lfloor \log(|N|) \rfloor + 2$, of accesses to nodes (corresponding to blocks for the server).

In choosing fillers, we need to ensure their indistinguishability from nodes in a target path. In this case, from the point of view of the server, any of the m nodes accessed could correspond to the actual target of the search, others being nodes in the path to the target or fillers, all indistinguishable one from the other. In this respect, choosing fillers just at random at any place in the data structure would not provide such a characteristic, as being completely unrelated in the structure, they could be recognizable as fillers. In fact, while (as we will see later on) we prevent the server from accumulating topological information across accesses, the server can observe a sequence of blocks accessed where a sequence of never-downloaded blocks is followed by a sequence of blocks intersecting with a previous search. This situation would expose the blocks in the intersection as fillers (a path is always connected, hence their occurrence after the never-downloaded blocks im-

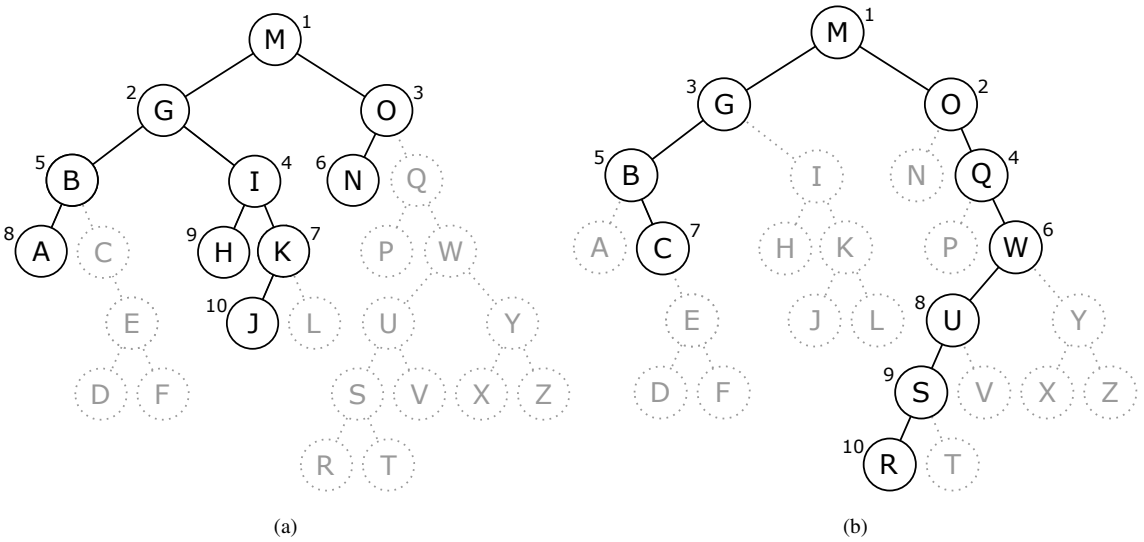


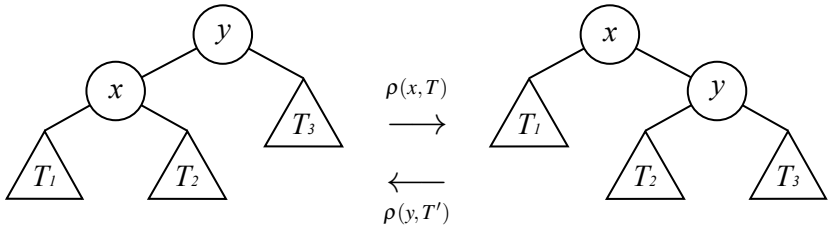
Figure 3.2: Two sample accesses

plies that they cannot belong to the path). A possible natural choice of selecting fillers at random wherever in the tree could then make them, or others following them in the sequence, recognizable as fillers. To avoid exposing accesses to such intersection attacks, we (randomly) choose fillers in such a way that they are connected to the paths (either target or fillers) being followed in the tree (i.e., a node can be accessed only if its parent has been) and always proceed forward in levels in the tree (i.e., the node visited next in the sequence cannot have a level lower than the one visited before it). Selecting fillers so that they are connected to nodes already accessed (path continuity) and with monotonically non-decreasing levels (forward visit) avoids possible intersection attacks from the server, guaranteeing fillers to be indistinguishable from a genuine path to a target.

Definition 3.5.1 (Uniform access) *Let T be the data structure. A sequence $A = \langle n_1, \ldots, n_m \rangle$ of nodes in T is said to be a* uniform access *iff:* 1) $m = 2\lfloor \log(|N|) \rfloor + 2$ *(constant number);* 2) $\forall n_i \in A : n_j \in path(n_i, T) \Longrightarrow n_j \in A$ *(path continuity);* 3) $\text{level}(n_i, T) \leq \text{level}(n_{i+1}, T)$, $i = 1, \ldots, m-1$ *(forward visit).*

Ensuring forward visit requires to 'think ahead' for the need of fillers, to avoid being blocked in a situation where there is no node that can be accessed at a level equal to or higher than the last one visited, but fewer than $m = 2\lfloor \log(|N|) \rfloor + 2$ nodes have been accessed. An easy way to avoid ending in such a situation consists in keeping track, in each node, of the number of nodes in the longest path of its children (i.e., for each of them, their height plus one), and, when performing searches, of the number of nodes to be still read to reach the fixed number $2\lfloor \log(|N|) \rfloor + 2$. Searches can then be performed level by level (forward visit). After having read the root and its children, we choose, in addition to the node to the target, one or more filler nodes, children of a node read at the previous level (path continuity), such that the sum of the number of nodes in their longest path is greater than the difference between the number of nodes to be still read and the maximum length of the path to the target. If a target is retrieved at level l, the search (at that and subsequent levels) continues with filler nodes only. The nodes to be accessed at each level in the tree are downloaded in sequence and in random order, to prevent the server from identifying how many blocks are accessed at each level and which of them is along the path to the target.

Figure 3.2 illustrates two possible accesses on a sample data structure with 26 nodes, which then requires to visit 10 nodes at each access. Nodes involved in the access are circled with solid



(a) T (b) T'

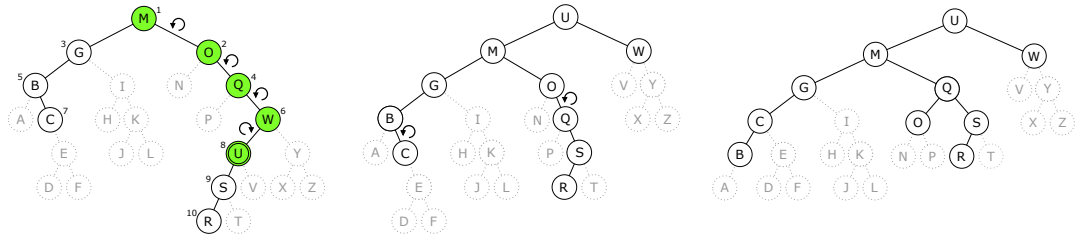
Figure 3.3: Tree rotations

lines and the numbers at their side represent the order in the sequence of requests to the server. In both accesses, any of the nodes could be the actual target or a filler. Also, the two accesses, while visiting different nodes, could actually correspond to a search for the same target (e.g., B).

3.6 Target Bubbling

Our second protection technique aims at hiding from the server subsequent (or close) searches for the same target. Even with fillers, such searches would necessarily contain the same path, and this situation could be easily observed by the server that would see access to the common sequence of corresponding blocks. For instance, any search for R in the tree in Figure 3.2 will visit nodes M, O, Q, W, U, S, R and 3 filler nodes (one of which will always be G), accessing the corresponding blocks. Observing accesses that visit 8 common blocks, the server can reasonably infer that the accesses are for the same target with high probability. The longer the common sequence, the higher the probability that the target of the two accesses is the same. To protect against such intersection attacks, our second technique simply dictates to bubble the target of a search up in the tree, so that at the end of the access, the target appears at the top of the tree (regardless of where it was before the search). A subsequent search for the same target (repeated search) would find the target high in the tree, then randomly proceeding following filler nodes. This would result in an access retrieving a set of blocks different from the previous one, hence appearing to the server as a search for a different target. A repeated search would then not be recognizable as such by the server.

In choosing where to move the target up in the tree, we note that placing the target in the root would seem the best choice for protecting repeated subsequent searches (as any search always accesses the root anyway). This would possibly expose recurrences of the same target at a fixed distance. In fact, after a sequence of all different searches (each aiming at a different target), the target of the m-last search would be at level m in the tree (having first been placed in the root and then moved down m times to accommodate the bubbling of the subsequent targets), and the server could exploit such a knowledge to make inferences on the target of the access. While noting that, thanks to the synergy with the other techniques of our approach, this situation would not be that deterministic, to the aim of avoiding any determinism in the first place, we move the target up in the tree choosing the (high) level at which to place it at random. We then assume a level *top* (which we expect to be typically 1 to 3) above which the target should be placed. At every access, the new tree level at which the target should be placed is randomly chosen between 1 and the minimum of *top* and the current level of the target (a target is never moved down).



(a) accessed nodes and target bubbling (b) speculative rotations (c) resulting tree

Figure 3.4: Nodes downloaded to access U and rotations performed to bubble up the target (a), tree with the target bubbled to the root and speculative rotations (b), and the resulting tree (c)

Moving the target up in the tree at the desired level is realized by applying classical single tree rotations of binary search trees. A rotation essentially swaps the child-parent relationship between a node and its parent, placing the node at the level of its parent and making the parent a child of the node (right if the node was the left child of its parent and vice versa). We denote the rotation of a node n in a tree T as $\rho(n,T)$. Figure 3.3 illustrates the result of such rotations, where tree T' in Figure 3.3(b) is the result of rotation $\rho(x,T)$ and tree T in Figure 3.3(a) is the result of $\rho(y,T')$. In the figure, we single-out only nodes directly involved (x and y), representing the remaining ones as sub-trees (T_1, T_2, T_3). Formally, a tree where the target has bubbled up is defined as follows.

Definition 3.6.1 (Bubbled Target) *Let T be the data structure, n be a target node, l=*level(n,T) *be its level before the access, and $l' \leq l$ be the new level at which the target should be placed. The data structure T' equivalent to T where the target has* bubbled up *is $T' = \rho(n,(\rho(n,\ldots(\rho(n,T)))))$, with* level$(n,T') = l'$*, obtained by recursively applying a sequence of $l'-l$ rotations.*

For instance, Figure 3.4(a) illustrates the nodes accessed by a search for U. The target is double circled, nodes in the path to the target are colored, and white solid nodes are fillers. Curved arrow on arcs show the rotations to bubble the target to the root level, swapping U with (in sequence) W, Q, O, and M. Figure 3.4(b) shows the resulting tree.

As already noted, since at each access the target is moved up in the tree, the targets of recent accesses will be located high in the tree (close to the root), while nodes that have not been accessed since long time will be at deeper levels in the tree (close to leaves). This is due to the fact that rotations that bubble up the target change the level of the other nodes in the top levels of at most one (up or down), and therefore it takes a few accesses for a high (or raised high) node to move down in the tree (e.g., the root node in Figure 3.4(a) becomes the left child of the root in Figure 3.4(b)). We also note that repeated accesses to a same target keep it in the top levels of the tree. This provides protection of repeated searches, since all such accesses, following random filler nodes in the tree, will look all different. We also note that bubbling the target with a recursive sequence of rotations causes changes in the topology of the tree, adding confusion to the server.

3.7 Speculative Rotations

Bubbling up the target after each access causes a natural re-organization of the tree. Because of rotations, at each access the height of the tree can increase (or decrease) of one. A long sequence of accesses can then potentially unbalance the tree structure. To ensure that any node can be reached

via a uniform access, we need to maintain the height of the tree to be at most $2\lfloor\log(|N|)\rfloor$. To this end, at every access, we consider nodes involved in the access in decreasing order of level in the tree, and, for each node, we evaluate whether its rotation can shorten some paths in the tree. Intuitively, rotation $\rho(n_i,T)$ decreases the length of the path reaching one of the children of n_i by one, while increasing by one the length of the path reaching n_i's sibling, say n_j. For instance, with reference to Figure 3.3, rotation $\rho(x,T)$ shortens of one the length of all paths ending in T_1 and increases of one the length of all paths ending in T_3 (the contrary happens for rotation $\rho(y,T')$). It is then easy to see that rotating n_i is potentially beneficial to shorten (or maintain limited) the height of the tree every time the height of the subtree rooted at n_i is greater than the height of the subtree rooted at its sibling n_j of at least two.

Definition 3.7.1 (Beneficial Rotation) *Let T be the data structure, n_i be a node in T, and n_j be its sibling. Rotation $\rho(n_i,T)$ is* beneficial *to possibly keep the height of T limited iff* height$(n_i,T)>$height$(n_j,T)+1$.

For instance, the beneficial rotations that are performed over the tree in Figure 3.4(b), resulting when bubbling up the target, are ρ(C,T) and ρ(Q,T). Enforcing them results in the tree of Figure 3.4(c). Note how rotating C decreases the length of the paths to D and F, decreasing also of one the height of the tree itself. Note also how the topology of the tree has changed with respect to the tree before the access (Figure 3.4(a)).

A beneficial rotation does not guarantee to reduce the height of the tree, but it can shorten subtrees, hence avoiding later degeneration of the structure. At every access, we evaluate whether rotating accessed nodes would be beneficial and, if so, we perform such speculative rotations. This occurs regardless of the height of the tree, to also try to avoid reaching a length close to our limit of $2\lfloor\log(|N|)\rfloor$. As only exception, we never perform rotations on direct children of the target (or of one of its ancestor) as this would decrease the level of the target (which should instead remain at the level where it was bubbled up). For instance, considering the tree in Figure 3.4(b), even if rotation ρ(M,T) is beneficial, it would move the target to a lower level. In our example, the height of the tree resulting from the application of speculative rotations (Figure 3.4(c)) is 5 while the height of the tree before access (Figure 3.4(a)) was 6.

Typically, simply performing beneficial rotations at every access allows maintaining the height of the tree within our aimed maximum of $2\lfloor\log(|N|)\rfloor$. In the unlikely case (one over 3,000 in our experiments) where after such speculative rotations the tree height is $2\lfloor\log(|N|)\rfloor+1$ (given our control it can certainly never go higher than that at any access), a further pass of rotations can be performed.

An additional advantage of our speculative rotations is related to the protection of access privacy. In fact, a rotation swaps the parent-child relationship between the rotated node and its parent, also changing the parent of one of its children (see Figure 3.3). Therefore, rotations change the topology of the tree, modifying search paths for some nodes. Since the topology changes at every access, the server cannot accumulate knowledge on it. We note that accommodating different topological structures is also the reason why we do not aim at maintaining the tree perfectly balanced (as the structure would have less degrees of freedom), and set instead - was a trade-off - our height limit to be twice the height of the perfectly balanced tree.

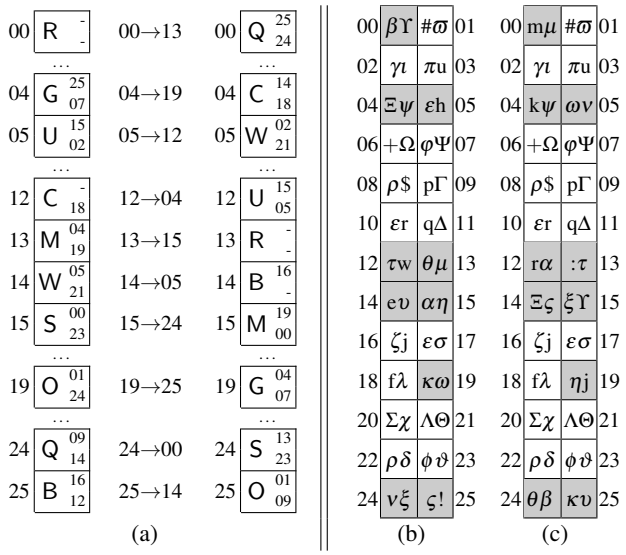


Figure 3.5: An example of physical re-allocation (a) and of view of the server before (b) and after (c) the access in Figure 3.4

3.8 Physical Re-allocation

Enforcing rotations to bring up the target and shorten paths changes the logical organization of the tree. As noted, such a change in topology provides some protection since it changes the location of nodes and therefore paths to be followed to reach them. Still, since the tree is a search tree, even if topology changes, strong commonalities can remain even after rotations. For instance, a rotation reverses a parent-child relationship between two nodes n_i and n_j, but still the two will remain connected. A sequence of rotations can bring more changes, but still common sub-paths may remain. The fact that the server can infer the path followed to reach a given node is not an issue per se since, as already noted, the index on which the tree is organized does not convey any information on the original index values and their relationships. However, if the server can maintain such a knowledge across the accesses, it can potentially reconstruct the topology of the tree and observe paths in common between different accesses, hence possibly learning information on an access.

We note that the server only observes accesses to blocks (not nodes) and that the parent-child relationship is (partially) known to the server since the access is iterative: a block will be child of one of the blocks accessed before. The uncertainty of the parent-child relationship comes from the fact that more nodes can be accessed at any level of the tree (since in addition to the target, also filler nodes will be followed). The $(i-1)$-th block accessed in an access sequence could be a parent, an uncle, a brother or even not be in a direct relationship with the i-th accessed block. For instance, in the sequence of nodes A=⟨M,O,G,Q,B,W,C,U,S,R⟩ (Figure 3.4(a)) accessed to retrieve U, M is the parent of O, O is the sibling of G, G is the uncle of Q, Q is not in relationship with B. However, such uncertainty cannot provide protection from a server observing common blocks among sequences of accesses. To prevent the server from accumulating information on the topology of the tree, we destroy such information by re-allocating all nodes involved in an access changing their physical location (i.e., changing the blocks where they are stored). At the physical level, and therefore from the point of view of the server, topological information is destroyed. A block id_i that contained the child of another block id_j before an access can now contain a node

appearing in a completely unrelated path that might even have only the root in common with the path to id_j, or be the root itself. A subsequent access visiting the same block id_i might (and most probably will) pertain to a completely different path in the tree. In other words, with re-allocation the (even uncertain) information on relationships among blocks that can be observed in an access will not hold anymore after the access is completed, preventing knowledge accumulation by the server. The physical re-allocation of nodes is formally defined as follows.

Definition 3.8.1 (Re-allocation) *Let T be the data structure and $\forall n_i \in T$, $id_i = \phi(n_i)$ be the identifier of the physical block storing n_i before the access. Let A be the nodes involved in an access execution and $\pi : ID_A \rightarrow ID_A$ be a random permutation of $ID_A = \{\phi(n) : n \in A\}$.* Re-allocation *changes the allocation function ϕ for all $n_i \in A$ to be $\phi(n_i) = \pi(id_i)$.*

Re-allocation entails moving a node to a different physical block (or leaving it at the same position if so dictated by the permutation). Re-allocation requires to re-encrypt the node with a different random salt. All blocks accessed will be rewritten and will all look different from any read block. The server will then not be able to learn any information on the re-allocation process and, in particular, will not be able to trace where the former content of a block might have been re-allocated. We note that, at the physical level, re-allocation also requires to update the parents of the re-allocated nodes, to guarantee the correct representation of pointers to children (and then the correctness of the tree structure). This is not an issue since the path continuity guaranteed by the access (Definition 3.5.1) ensures that the parent of every node involved in the re-allocation is also involved in the same re-allocation (and therefore it is available to the client for content update and re-writing). Figure 3.5(a) illustrates the original content of the blocks accessed by the search in Figure 3.4, an example of their physical re-allocation, and their content after its application. In the figure, we report in each node its index value and the identifier of the blocks storing its children (symbol $-$ denotes the absence of the child). The block identifier is reported on the left of each block. Figures 3.5(b-c) illustrate the server view before (b) and after (c) the access, where blocks downloaded/uploaded are colored.

Thanks to the fact that every node is moved to a different untraceable physical block every time it is accessed, re-allocation prevents the server from determining whether two accesses visited a same node (or sub-path). Hence, the server will not be able to reconstruct the frequency of accesses to nodes by observing accesses to physical blocks. Indeed, accesses that aim at the same target (or visit the same path in T) will access a different set of physical blocks. Furthermore, since re-allocated nodes belong to different paths and are located at different levels in the tree, re-allocation also destroys information the server could have gained on the topology of the tree by observing the sequence of accessed blocks/nodes. In fact, a block storing a node at level i in the tree might contain after the access a node in a completely different path and at a completely different level.

3.9 Analysis and Experimental Evaluation

To assess the access privacy provided by our approach we need to evaluate the indistinguishableness of accesses or - put another way - the degree of confusion on the accesses to the server. To this end, we start noting that the physical re-allocation employed by our approach can be compared with the physical re-allocation of the shuffle index [18] (it is actually stronger). In particular, the shuffle index re-allocates the logical nodes accessed on disjoint physical paths of its tree structure on a per-level basis. The entropy-based analysis used to show the soundness of such a mechanism in obfuscating the mapping between nodes and blocks applies also to our approach. In addition,

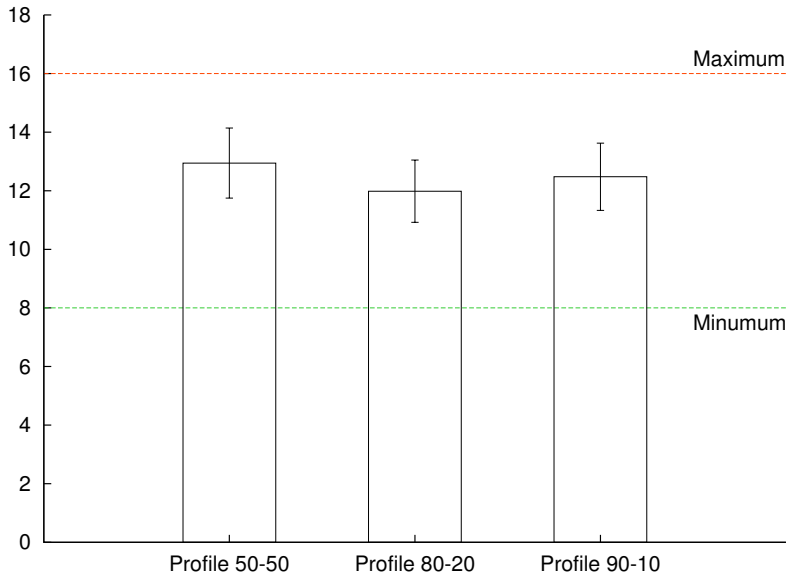


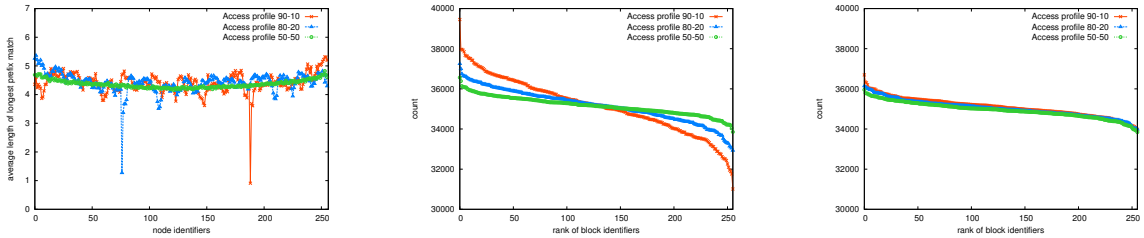
Figure 3.6: Average height of a tree with 256 nodes, considering 500,000 accesses

our approach enjoys even stronger guarantees than the ones proved in [18]. Indeed, the shuffle index changes physical location of only a limited set of nodes and operates only within level of the logical structure, while our approach changes the allocation of all nodes involved in an access, operating also across levels, hence producing a complete re-allocation of the whole set of accessed nodes. With respect to short-term observations (protected by the cache in the shuffle index) our approach, bubbling the target at the top, is clearly protected since repeated accesses are indistinguishable, as already noted. Enjoying such theoretical analysis and observations, which apply also to our solution, we then performed an experimental analysis on our approach. We evaluated how and to what extent our proposal hides to the server the correspondence between nodes and blocks where they are stored. We implemented our approach in Java and evaluated: *i)* how the *height of the data structure* can vary; *ii)* the *effectiveness of rotations* in protecting access privacy; *iii)* the degree of *obfuscation of the actual paths* observed by the server at every access request due to the physical re-allocation.

In our analysis, we used a data structure with 256 nodes and a height ranging from 8 to 16. We simulated different access profiles by synthetically generating a sequence of target index values that follow a self-similar probability distribution with skewness γ in the range $[0, 0.5]$[1]. A value of γ=0.5 generates a sequence of values that follows a uniform probability density function. In particular, the results of our experimental evaluation have been obtained executing 500,000 accesses for target values drawn from three self-similar distributions [26] with γ=0.5 (50-50 rule), γ=0.20 (80-20 rule), and γ=0.10 (90-10 rule), respectively.

Data structure. Figure 3.6 shows the average height of the tree for the different access profiles. As visible from the figure, the average is around $1.5h$ (with $h=\lfloor \log(|N|) \rfloor$ as a baseline), with a sample standard deviation of 1. Hence, the data structure maintains itself within the set limit, also nicely providing rooms for fillers in the search. Since the height of the tree dictates the number of

[1] Given an index domain of cardinality d, a self-similar distribution with skewness γ provides a probability of $1-\gamma$ of choosing one of the first γd domain values; the same proportion holds when considering any sub-range of the domain values.



(a) (b) (c)

Figure 3.7: Average length of the maximum common prefix among the paths reaching the same target (a) and rank/frequency distributions of the block identifiers corresponding to self-similar access profiles with $\gamma \in \{0.5, 0.2, 0.1\}$ when only the physical re-allocation (b), and when all protection techniques are applied (c)

client-side interactions that would be needed to access the data structure (following a path in the tree), we note that, requesting a constant number of accesses, our solution exhibits a $\times 2$ overhead with respect to an encrypted binary tree (i.e., an encrypted binary tree that still requires the client to visit the tree level-by-level), which however would provide no access privacy protection at all.

Effectiveness of rotations. To evaluate the effectiveness of rotations (target and speculative bubbling) for protecting access privacy, we analyzed the average length of the common prefix paths to a common target in sequences of subsequent accesses. Figure 3.7(a) shows the results of such an analysis, where the x-axis reports the node identifiers in ascending order, and the y-axis reports the average length of the common prefix. It is interesting to note that the reported average value for the maximum common prefix of the logical paths aimed at the same target is around one third the average height of the tree (Figure 3.6), implying that rotations largely change the topology of the data structure. Also, if the statistical distribution of the target values is highly skewed, only a few values will be accessed for serving most of the access requests. The two spikes in Figure 3.7(a) confirm that our approach keeps as near as possible to the root the most recently (and frequently) accessed nodes, thus being effective in making subsequent accesses to the same target indistinguishable from random ones.

Path obfuscation. The experimental evaluation validates the ability of physical re-allocation involving all accessed nodes to provide indistinguishability of the profiles of the accesses to the data structure. Figure 3.7(b) shows the rank/frequency distribution of block identifiers observed by the server when only physical re-allocation is applied. The figure shows that the physical re-allocation alone is already able to make skewed frequency distributions of the accesses to the blocks quite close to the one corresponding to a flat access profile.

Combined protection. The small differences among the curves in Figure 3.7(b) are a consequence of the information leakage coming from the observations of the blocks shared by different access requests. Such differences disappear thanks to the contribution of rotations. This is visible in Figure 3.7(c), showing the rank/frequency distribution of block identifiers observed by the server during the execution of access requests when all our protection techniques are applied. The figure validates our approach to preserve access privacy as it shows how the proposed techniques make skewed frequency distributions of accesses to the blocks statistically indistinguishable from the one produced by a uniform access profile.

4. Scalable Distributed Key Management for Cloud Storage

As use of cryptography is increasing in all areas of computing, managing keys in distributed systems has to be solved efficiently. Large deployments in the cloud can require millions of keys for thousands of clients. All current approaches to cope with this problem are centralized key managers which do not scale out as necessary.

This section reports on the realization of a key manager which uses an untrusted distributed key value store (KVS) for consistent key distribution over the Key-Management Interoperability Protocol (KMIP). To achieve confidentiality, we use a key hierarchy where every key except a root key itself is encrypted by the respective parent key. The hierarchy also allows for key rotation and, ultimately, secure deletion of data.

The prototype was integrated with IBM GPFS, a highly scalable cluster file system, where it serves keys for file encryption. Linear scale-out was achieved even under load from key updates. Concerning performance, the throughput and latency is bounded by the TLS-handshake procedure and the performance of the distributed KVS.

4.1 State of the Art

Encryption plays a fundamental role in realizing secure computing environments. Unlike classical systems, cloud-scale distributed installations pose additional challenges. Especially key management has to cope with their increased complexity and ensure reliable and secure distribution of keys to many legitimate clients, which are then able to encrypt files or establish secure network communication. Most encryption schemes take a secret key and the data as input and transform this into a ciphertext. The ability to decode the ciphertext back to its original plaintext is dependent on the encryption algorithm and on the used key. Algorithms used for encryption are standardized results of a public revision process. The aim is to make the security of the system depend only on the knowledge of the secret key and not of the secrecy of the method. Therefore it is extremely important to maintain the confidentiality of the keys.

Because key management is an issue in many environments, standards were introduced to allow key management systems to be built independently from the components using the keys. Current solutions to the problem include the OASIS Key Management Interoperability Protocol (KMIP) [33], which specifies operations for managing keys at a remote server. Other key managers provide a PKCS #11 interface, or a REST API. In the context of OpenStack, for example, where every service interaction is a REST call, the Barbican project seamlessly provides all necessary key management functionality for other services.

Key managers differ in the operations they support and in their performance, resilience, and security capabilities. More complex and evolved solutions such as the Vormetric Data Security

Management (DSM) or the IBM Security Key Lifecycle Manager (ISKLM) key management servers (KMSs) put emphasis on the needs of a business environment. For example, governmental standards for handling health data dictate a reliable audit trail to reconstruct all operations in case of a problem. This function unavoidably makes the key manager more complex. Such enterprise key managers were also designed for high availability, to allow uninterrupted service.

While it already is difficult to secure keys stored on one system, this becomes even harder in a distributed system with multiple entities performing cryptographic operations. First keys have to be generated from a cryptographically strong source of randomness. These keys then have to be distributed in a secure way to all nodes of the system which need them. And finally it should be possible to revoke keys and destroy the key material permanently. This life cycle is valid for any key in the system. Instead of performing this manually which is prone to human error, special key-management software can automate these tasks.

Unfortunately this automation can not be trivially scaled to a distributed environment. A distributed key manager faces a lot of challenges common among distributed systems, which it has to solve without compromising security. Also its concepts of processes and the user interface have to be designed in a way that the most intuitive and easiest way is the secure one. Accidental configuration errors should be prevented by useful defaults and extensive checks for plausibility of user input. Key managers are an integral component of a cryptographic environment and should be developed with the same diligence as every other component.

4.2 ESCUDO-CLOUD Innovation

In the remainder of this chapter we present a solution for scaling a key management service to cloud applications with thousands of clients and possibly millions of keys. This includes the capability to dynamically scale out based on demand while maintaining security. Our distributed key management solution should handle all core tasks and scale in a linear way. No existing key manager provide such scalability.

In particular, our solution addresses an enterprise environment where a key management system is deployed. Clients accessing the key manager can be any application that performs cryptographic operations and therefore needs access to keys. They can retrieve these keys after successful authentication and authorization from a key management server. The key management server has to provide a consistent view of the available keys even if it serves keys from multiple servers. Eventually keys have to be stored somewhere. There are three different regions with different properties.

To store keys persistently across power cycles, there has to be a storage medium that is local to the key manager. To destroy keys it has to provide a way to be securely erased. A possible implementation is a battery backed hardware security module (HSM) with the key in volatile memory, so that the key can be deleted by power cycling. It is important that there must be no way to recover the key afterwards even for former legitimate parties. Another, cheaper solution is to use removable USB drives which can be physically destroyed. Despite their possibility for erasure, these solutions are not scalable and have low capacity in common.

Keeping keys up to date across multiple nodes is possible by managing them in a distributed key value store (KVS). There are different distributed KVS, most of them only providing eventual consistency. Their key-value pairs are versioned and they support conditional put operations which allows to perform atomic get, update and put transactions. The problem is that confidentiality can not be guaranteed in this distributed system and it should only process encrypted content.

Eventually the key manager has to serve keys in plaintext. This is only possible if they are available in decrypted form in the memory of the machine running the key manager. They are protected from other applications running on the node by the separation of virtual memory. Keys that are no longer needed can be overwritten and thereby effectively erased.

4.3 Security Model

In a typical scenario where a key manager is deployed, there are three types of actors. The key manager, the client, and a possible third party which has access to the ciphertext, but is not supposed to decrypt it. This can be a storage provider of the client or a network operator between the client application and one of its users. Trusted connections between actors can be established, if they share a secret.

The client has to have means to authenticate against the key manager and establish a confidential connection, which can be done using TLS and mutual authentication using client and server certificates. The key manager then authorizes access to keys and delivers them to the client over the secure channel. After the clients encrypt some data, they can store it or send it to some user. Either way, it is important that the key management operator does not collude with either of the other parties. A storage provider colluding with the key manager could circumvent any client based restrictions and decrypt the storage without its interaction.

4.4 Objectives

Our new key manager aims at satisfying the following objectives.

$\mathcal{O}1$ **Scalability:** To provide keys for several thousand clients, the service must be able to scale out without compromising the atomicity of operations. So all key servers have to operate in a consistent way, to serve the current version of a key as well as to not serve deleted keys. It should also be possible to dynamically adapt the number of servers to the load of the system. This requires an online way to add new key servers to the cluster.

$\mathcal{O}2$ **Availability:** Associated with scalability, the service must also be resilient to failures of individual nodes, which can, after being repaired, rejoin the cluster. Concerning fail-over and load balancing, we can rely on the clients, which randomly choose between the announced endpoints and change the server for retries. A failure of availability results in a complete breakdown of the provided service, as the clients no longer have access to the encrypted data. Problems with the consistency of different key management servers can result in loss of data, when different clients encrypt with different keys.

$\mathcal{O}3$ **Security Model:** As it is necessary to handle plaintext keys in the key manager, we have to trust the operator of the machine it is running on. It should however be possible to achieve confidentiality of keys at rest. This includes that a powered off system can not leak sensitive key material.

$\mathcal{O}4$ **Secure Deletion:** Clients can be requested to delete information permanently. As it is hard to assure definitive deletion of data from hard drives it is even more challenging on solid state media.

The approach introduced by Di Crescenzo et al. [21], and more recently extended by Cachin et al. [8], reduces the amount of data, namely to a single key, which has to be deleted permanently by encrypting the user data. To securely erase all stored data, it is now sufficient to securely erase the key, after which the user data, given strong cryptography, resembles only random noise. This also works for partial deletion, where some files can be retained by re-encryption under a new key before the old key gets deleted. From a key manager perspective, all these operations can be supported by key rotation. We need a possibility to change every key and permanently destroy the sensitive key material that is no longer used.

$\mathscr{O}5$ **Usability:** The encryption system is only as secure as the keys used. For many systems, the weakest point is erroneous handling of sensitive material by the operator. If the system fails to provide the most convenient way of operating it and there is a less secure and lazier alternative, sooner or later this alternative will be taken. By streamlining the processes beforehand, it is possible to check for security breaches in these and if any breaches are detected, rigorous checks can be implemented.

4.5 Related Work

In every cryptographic system there is also a key manager. In an enterprise context support for key-lifecycle management is important [7]. Several standalone key managers have been developed to provide a generic service and support the standard protocols, such as KMIP [33].

The IBM Secure Key Life-cycle Manager is an enterprise key manager with all necessary functions. It runs on an IBM software stack with WebSphere and DB2. With high demand, ISKLM can run distributed on a high availability cluster in a robust way. ISKLM provides two interfaces to manage and retrieve secrets. First, a web interface allows for comprehensive management, control and auditing; second a KMIP server interface gives the clients access to keys. Clients are authenticated through the secure connection, using a TLS client-certificate, and then by an in-line in KMIP using a proprietary authentication scheme. We will provide a subset of ISKLMs features and within this be fully compatible. Additional security features such as PKCS #11 attachments to Hardware Security Modules are not within our target solution and ISLKM is better suited for these scenarios. The main drawbacks of this solution are its high complexity and initial cost as well as the operational expenses. Our solution targets architectures that need a scalable lightweight key manager without the rich features of ISKLM.

A related solution is the Vormetric Data Security Management (DSM) server [41]. Besides the interoperability with KMIP it facilitates integration with database systems such as Microsoft SQL Server using a proprietary key agent tool. Like ISKLM it is FIPS 140-2 certified and can be used in regulated businesses that handle sensitive customer data. It is also based on a centralized architecture and inherently not scalable.

Another lightweight key manager is the python based Barbican inside OpenStack [5]. As all the services in OpenStack, it provides a simple REST API for all its operations. Its back-end is designed as a plug-in system, so that at the time of writing (2016), it supports a SQL database or a KMIP server to hook into business environments, which run, e.g., a central ISKLM. Another plug-in can communicate with an HSM over PKCS #11 to store the root of a key hierarchy there securely. The scalability of Barbican is delegated to the scalability of the database used in the back-end. To process requests, multiple workers can run in a distributed fashion, but they all interact with a central atomic view of a database via a message queue.

Cloud key management services such as IBM Key Protect, Amazon KMS and CloudHSM [29, 3, 4] allow users to trade off many of the complexities of managing an in-house key management server against security, by expanding the trust boundaries to include the key management service hosted in the cloud. Some cloud KMSs store the encryption keys in secure hardware security modules (HSMs), and they are typically accessed using Barbican or proprietary REST APIs.

4.6 Design

Given the requirements of scalable key management, we designed a software architecture from scratch.

To achieve $\mathcal{O}1$, scaling out, we need a way to operate all the nodes of the key manager coherently. Because there are already good solutions that provide a consistent access to data, we rely on a distributed KVS for maintaining our keys. Our key manager is independent of the underlying KVS and only requires a small interface with $get : (key) \rightarrow (value, version)$ and $put\_conditional : (key, value, version) \rightarrow (success)$. Multiple platforms can provide such an interface[1].

Keys stored in the KVS need to be protected regarding confidentiality and integrity. Our approach solves this by encrypting the keys before putting them into the KVS. This requires the nodes to have a key to access the object. The result is a two level hierarchy with root encryption keys (REKs) at the root and the client keys below them. Each child key is encrypted by its parent. As the REKs can not be itself distributed over the KVS, they have to reside in node-local, erasable storage, such as a HSM or USB drive.

Using the distributed KVS as the only means to communicate between otherwise completely independent key management servers, $\mathcal{O}2$, availability can be guaranteed by having nodes in different failure zones. Keeping a consistent version of the key store among all nodes is assured by the KVS. The key management service itself is completely stateless and can operate if it has access to the REK and the KVS.

Our key hierarchy also allows us to fulfill the security requirements of $\mathcal{O}3$. In a powered down system, only the REKs are stored unencrypted. All other decrypted keys only existed in volatile memory and are no longer accessible.

The key rotation operation required for $\mathcal{O}4$ is a multi-step process. To permanently prohibit access to a key from the KVS it is not sufficient to remove it, as the KVS might leak information and given the REK one would still be able to recover the key. The only way is to replace the REK, which is possible because it is stored on erasable media. Retaining access to other keys is done by re-encryption under the new REK. Even with a recovered or leaked copy of the old key store, it can no longer be decrypted, as the corresponding REK is reliably erased.

As all the keys handled are symmetric AES-keys, we need to store them in plain and encrypted form as a file to store on the USB drive and in the KVS.

4.6.1 File Formats

It would be useful to store the keys in a standardized file format to increase the interoperability between tools to manipulate them. Unfortunately, there is no standard format for symmetric keys.

The PKCS #12 file format supports lists and hierarchies of keys and can encrypt the entire file, but it lacks a bag type for symmetric AES-keys. Adding a custom bag type does not help, as any

[1] CoreOS *etcd*, Apache *ZooKeeper*, *Ceph*, IBM *GPFS*

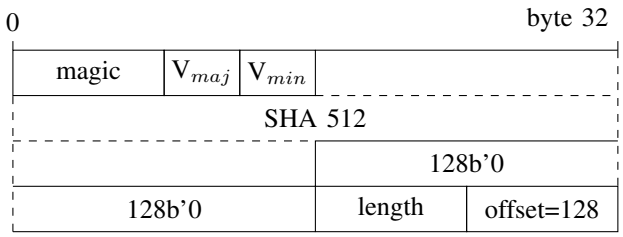


Figure 4.1: Header structure of REK files with major and minor version numbers V_{maj} and V_{maj}. The magic is `\xe3REK\x0d\x0a\x1a\x0a`. All numbers are big-endian unsigned integers.

deviation from the standard is not guaranteed to be widely supported.

The Java Cryptography Extension Key Store (JCEKS) in its latest version supports symmetric key objects and has a serialization format to store on disk. But the serialization is Java-specific and can not be reliably parsed by other languages or serialization engines.

Therefore we designed our own format which follows the published KMIP standard as closely as possible. KMIP was designed as an interoperability protocol, so it does not work for data storage as is, but provides a useful tag, type, length, value framework to save structured data. The response format of the interactions conveys the same information which is normally needed to store. So our storage format looks like a KMIP response message with all unnecessary tags stripped off. In the header we do not need a date specification and for the batch items there is no operation or status field. The response payload perfectly suits our storage needs, as it is a suitable structure to save key material together with its parameters such as length and type.

A direct advantage of this format is the reusability of the KMIP parsing and generation code. KMIP is designed to be used inside TLS with a possibility for using other secure protocols, which provide a lot more than a simple file. Integrity, confidentiality and authentication are ensured. For a KMIP response object to be suitable as a file, we designed a file header, which can provide the same environment. The file header follows the same big-endianness as KMIP itself.

REK Files

For the REK files, which only need integrity as we can not encrypt them, a hash of the whole file is sufficient. With a SHA512 hash, the probability of accidental corruption to result in a valid file is very small. To conform with tools like `file` and operating systems, we added a unique magic byte string for the first 8 bytes. We included a version field that can be incremented if the hash-specification is deprecated, or if we transition to another payload format like XML.

The size and position of the payload is not only implicitly specified, but also directly by a length and offset field in our header. With all values set, and the 64 bytes for the hash set to zero, a SHA512 hash is calculated over the whole file.

Key Store

The key store which is handled by the KVS has to hold multiple keys and must be encrypted. KMIP is a very flexible protocol and supports batch responses. So packing multiple keys inside one response structure is not a problem. It is even possible to handle wrapped keys with all the related information about the wrapping key and wrapping method. For the sake of simplicity and

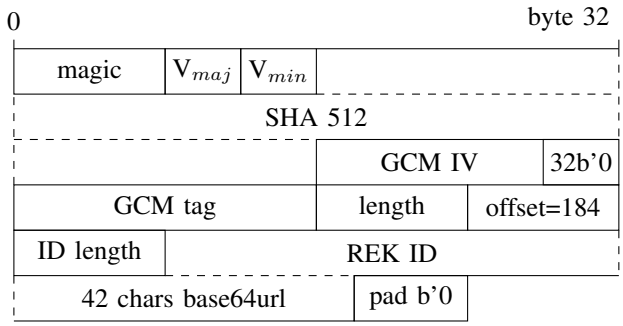


Figure 4.2: Header structure of the key store with major and minor version numbers V_{maj} and V_{maj}. The magic is `\xe3MEK\x0d\x0a\x1a\x0a`. All numbers are big-endian unsigned integers.

privacy, we decided to use the same KMIP-format for the payload as for the REK files, and to encrypt the whole KMIP body. This has the advantage of not disclosing any IDs of keys in the store.

For the encryption and authentication, we need at least an REK key ID to decrypt with the right key. As cipher mode we chose Galois Counter Mode (GCM), which provides authenticated encryption. It also allows to include additional data in plain that is authenticated together with the encrypted message, which would be perfect for the header, but was not used since the integrity of the header is already guaranteed by the SHA512 hash. Compared to the REK header, there are three new fields: The GCM initialization vector, the authenticated hash (also named tag) and an REK string which has a length field in front of it and is padded with zeros to an 8-byte boundary.

We use a 96-bit random number from our cryptographic library as initialization vector. This is announced as sufficiently secure by NIST [23].

Tampering with the unauthenticated header fields does not lead to a malfunction of the KMIP parser, as the payload first needs to be decrypted and authenticated by the correct key.

4.6.2 System Interactions

The key manager operation is split into a server and a tool. All administrative tasks are performed directly on the KVS stored data by the tool, and the KMIP server only handles get-key requests. In the following we list the most important calls in the API of the key manager.

rek_random() $\rightarrow$ (*REKfile*, ID_{REK})
To bootstrap the system, a root key for encryption is needed. The tool creates a new random key and stores it in an REK file. Along with the key material, a unique ID is generated. This does not disclose information about the key but, compared to user defined IDs, avoids having different key material with the same ID on separate machines. The downside of this approach is usability, because IDs have to be copied and pasted or entered tediously.

mek_init(*REKfile*) $\rightarrow$ (*MEKstore*)
The minimal requirements for a running system able to serve keys, are an REK file and a distributed store with at least one key in it. With the REK file, an empty store can be created. This consists of a KMIP payload with a header only and zero items. The resulting encrypted file under the REK is then put conditionally into the KVS under version zero to ensure that no one else created the same store concurrently.

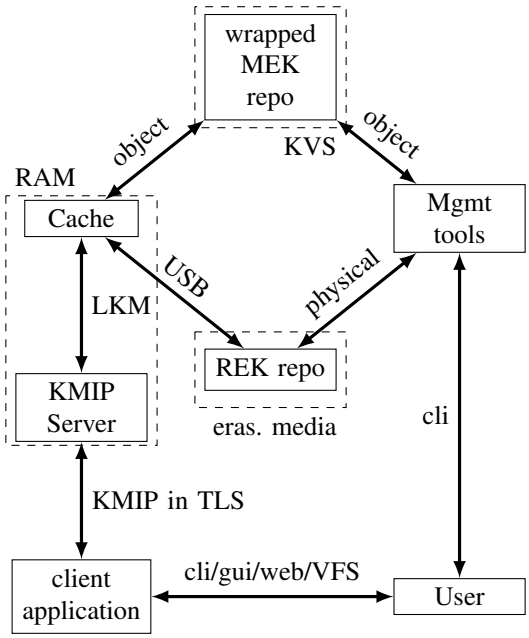


Figure 4.3: Overview of the key manager components. Sensitive key material is stored as specified by the dashed regions

$mek\_random(REKfile, MEKstore) \rightarrow (MEKstore)$

Given an REK file and access to the KVS, all operations on the store follow a common structure. First the locally available REK files are parsed and the available REK IDs are extracted. Then the version and content of the store file from the KVS are fetched. The store is decrypted with the matching REK and now it is possible to change the stored keys.

If there is a change to the content of the store, as, e.g., an added key, the store has to be flushed back to the KVS at the end. Therefore it is serialized into KMIP and encrypted again with the same REK used for decryption. With the version of the store, obtained by the get operation, a put-conditional is issued and if it fails, the procedure is retried. It may fail completely, when two nodes try to add the same key simultaneously.

Otherwise inserting a key into the list of stored keys is a small atomic operation. The new store is directly submitted to the KVS and all active nodes can respond to queries for it because they can access the store with their local REK file.

$get\_key(ID_{key}) \rightarrow (key)$

The get-key operation is the only one used by the serving component. Our key management server is listening for new requests. For every connection, a TLS handshake is performed to authenticate the client by its client certificate. This is the only authentication mechanism so far. Over the secure channel both participants exchange data in the KMIP format. The client can now request a key with a specific identifier. After extracting the ID, the server queries the cache. If there is a valid copy in the cache, it is returned, packed in a KMIP structure and sent to the client. This terminates the TLS channel as we only allow one key per session.

In two cases the cache has to query the KVS for a key. Either it has no copy of the key at all, or one of the cached keys reaches its refresh-age. This means that a key in the cache is revalidated every 15 minutes. If the key is still available in the store inside the KVS, it stays in the cache,

otherwise it is evicted. In contrast to eviction and on-demand fetching, this method reduces the query latency and is feasible as the cache size is small. Eviction only happens if the cache reaches its capacity limit.

In order to fetch a key from the KVS, the server has to perform the same steps as the management tool. First it looks for available REKs, then gets the store file from the KVS and decrypts it. The required key can be extracted by parsing the decrypted KMIP payload. Fetching a key is a read-only operation: Because the store is not modified in this operation, there is no need to write anything back to the store file, and the local copy can be discarded.

mek_delete(*MEKstore*, *key ID*) → (*MEKstore*)
To remove a key from the key store, the object from the KVS is fetched, decrypted and parsed. If available, the requested ID is deleted and the store is put back conditionally to the KVS. When any other client performs an update simultaneously, the action has to be repeated. It can fail if the interrupting operation has already deleted the key.

mek_init(*REKfile*) → (*MEKstore*)
After a key has been deleted from the KVS, there is still the possibility to decrypt an old version, because the REK is available. To perform a secure deletion of one or multiple previously deleted keys, it is necessary to rotate the REK.

A way to achieve this, is to load the store, but saving it encrypted with a new key. This new REK can be created like the previous one with *rek_random*. To avoid downtime, this key should be distributed to all key servers before the re-encryption of the store. Deletion of the old keys can be achieved by saving the REK files on cheap USB drives and physically destroying these.

4.7 Evaluation

To evaluate our key manager and verify that the prototype meets all our defined goals, we used it to serve keys for a cluster running IBM's General Parallel File System (GPFS)[2]. GPFS is a cluster file system that also offers encryption at the level of files. Therefore each node needs access to a set of shared keys. The individual nodes of a GPFS cluster are each KMIP clients and can query the required keys from our key manager. GPFS uses an integrated distributed KVS called cluster configuration repository (CCR) which we use via the command line interface (CLI) tool.

The test was performed in a realistic environment were we set up a real GPFS 4.1 cluster on two physical servers. Each is equipped with dual Intel® Xeon® E5630 processors, 40GB memory running RedHat enterprise Linux version 7. The key manager nodes shared the same physical machines with the GPFS. We created a file system in our cluster which had a policy to encrypt new files using a key stored in our key manager.

We specified the node of the cluster running the key manager as the KMIP-URL. Because the GPFS KMIP-client was built to support high availability set-ups, we specified fail-over addresses. The file system policy triggers the encryption process when a new file is created, therefore fetching the specified key. We confirmed this transaction in the network trace as well as in the log of the key manager.

To test the high availability scenario, we did not create files but directly hooked into the key fetching code and ran it from a test process. This uses the same native code as the file system. The parameters were to time out after 20 seconds and retry once before selecting the next server. We

[2]Recently rebranded as *IBM Spectrum Scale*, see `http://www.ibm.com/systems/storage/spectrum/scale/`

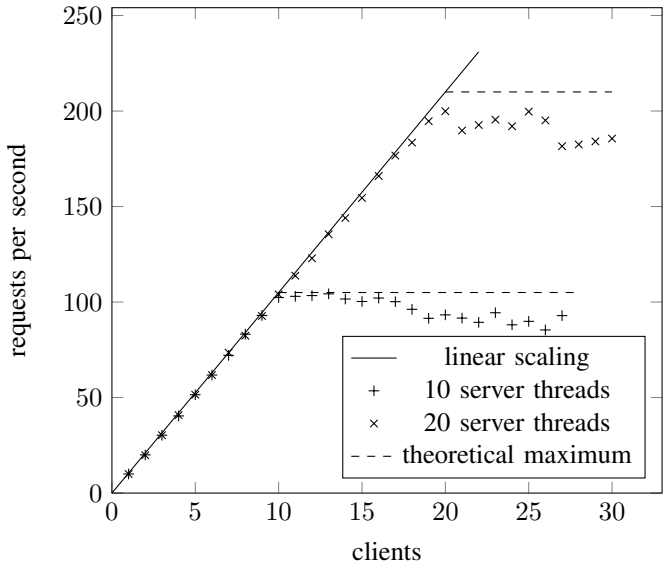


Figure 4.4: Scale up of the service with client side load balancing. It scales linearly. The samples were taken over a 10 seconds average.

queried a key repeatedly and then shut down the primary key server while monitoring the network traffic. The resulting exchanges matched the expected behavior.

4.7.1 Scaling

A reasonable measure for the performance of the key manager is the number of keys it can serve per second. Because the key manager now runs along with the cluster file system on the same nodes, it is advised to assign a small amount of resources on multiple machines to serve keys instead of dedicating one node completely to key management. This greatly improves reliability, as the key servers can then be spread over different failure zones. Restriction of resources is achieved by limiting the number of concurrent clients, namely the size of the thread-pool handling the connections.

We queried one key from a single key server with an increasing number of native clients. These were configured to query the keys back-to-back. Figure 4.4 shows the linear scaling when the system operates below its capacity limit. By adapting the thread pool size, we can show that the service scales up as expected. The scaling is independent of the threads' distribution to different machines. The number of threads can be distributed arbitrarily to nodes with at least one thread per node.

In Figure 4.5 the independence of the threads is shown. Running 6 threads on one server yields the same result as dividing them evenly among two machines. In the distributed case, the clients randomly chose a server for each key.

In the overload scenario, we see a degradation of the service, as clients fight for resources. To resolve such an overload situation, additional nodes can be added to the cluster. Clients, configured in high availability mode, will fail over to the new nodes automatically.

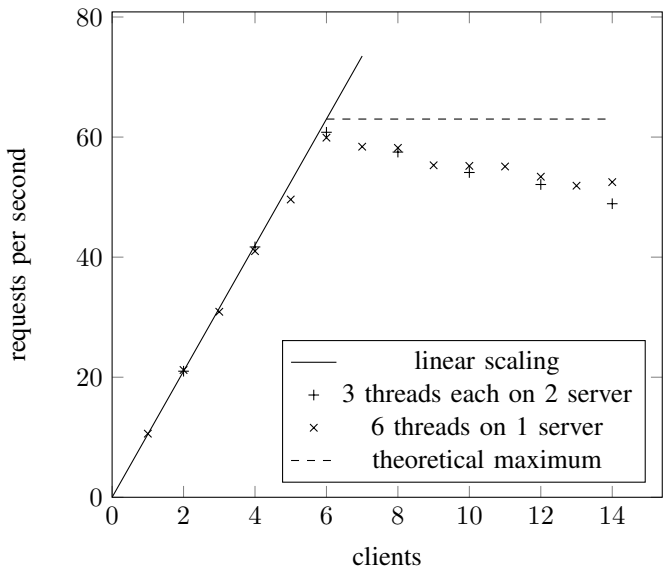


Figure 4.5: Scale out of the service with client side load balancing. It scales linearly. The samples were taken over a 10 seconds average.

4.7.2 Latency

The throughput of keys is mainly important for administrative tasks. Booting the cluster or mounting a file system queries a lot of keys. For standard operations, keys are fetched rarely. Therefore the latency until the required key is fetched, is important when the application opens a new file.

Experimental Setup To reliably measure the latency of the key retrieval, we captured packet traces on the querying machine. In these traces we can measure the total delay between the TCP SYN handshake and the KMIP response and TLS-session tear-down.

The total latency is composed of two parts. First, a considerable amount of time is spent to create the TLS channel. Then the second part represents the key manager's operation. Depending on the key queried, it might be served quickly from the cache. If the key is not in the cache, which is also the case for an unavailable key ID, the key manager has to fetch it from the back-end store.

TLS Handshake The setup of a TLS 1.2 secured channel takes several message exchanges, which might suggest that the roundtrip time (RTT) of the network is the limiting factor. The RTT in Ethernet or InfiniBand clusters where GPFS is used, is generally less than 0.5 ms. This is also true for a normal gigabit Ethernet installation. With a total of about 7 exchanged messages, the overall time should be a few milliseconds.

Our observed TLS handshake time lies at around 100 ms. To test which parts of the handshake take the longest and might be optimized, we performed tests against an ISKLM server with the GPFS client as well as querying LKMS with the default OpenSSL client. We measured the following timings shown in Figure 4.6.

Apart from the general processing times of the TLS protocol at client and server, our TLS implementation (provided by IBM Global Security Kit, GSKit) takes 40 ms to agree on a secure communication. When both client and server run on GSKit, this agreement is done serially, costing a total of 40 additional milliseconds.

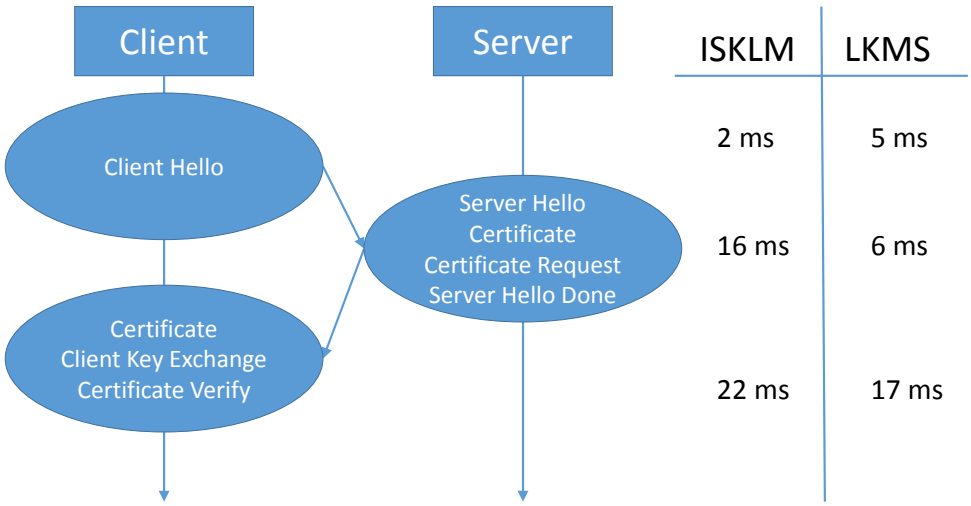


Figure 4.6: TLS handshake timings for GPFS client querying our LKMS (with `TLS_RSA_WITH_AES_256_CBC_SHA`) and an ISKLM (with `TLS_RSA_WITH_AES_128_CBC_SHA`) for a cached key. Regarding the handshake alone, ISKLM is faster in accepting the Change Cipher Spec and therefore the time to first byte (TTFB) is reduced. The times are means and each have a standard deviation of up to one millisecond. The RTT was around 0.2 ms.

Other TLS implementations, such as the ones used by ISKLM and `openssl` have optimized this delay, so that they are approximately 40% faster to the first byte transmitted. According to the TLS specification [22] the node verifies certificates and performs cryptographic operations. A plausible reason for the delay is a slow certificate validation, as the RSA calculations necessarily have to be done by all the implementations.

Processing Time After a secure channel is established, the client can request a key by its ID. The time between this request and the response packet, including the key or a failure message, is regarded as the processing time of the key manager. This time varies depending on the caching state of the requested key. To achieve representative results, we pre-warmed existing caches. For the ISKLM the processing takes about 150 ms for cached keys and 200 ms for new requests.

In our LKMS implementation, fetching keys from our distributed KVS is expensive and we expected a significant difference between cached keys and unavailable or not cached keys. When the key can be fetched from main memory, the network latency dominates the request with 0.5 ms. Parsing the KMIP query and creating the response payload can be performed in far less than 0.5 ms. This was determined by measuring the time difference of request- and response-packet on the server side.

Our interface to the KVS is a `popen` call which invokes the shell and then the CCR binary. This call to get the encrypted store takes about 86 ms. The total processing time for this class of requests takes around 90 ms. The fluctuations over time due to scheduling on the server show some small spikes, but they are about one order of magnitude smaller than the total duration. This fits with the usual scheduling of a Linux system, where scheduling decisions are done on a finer level than 100 ms.

Table 4.1: Processing times in milliseconds of queries to CCR and its components.

		minimum	mean	std. dev.	maximum
`popen` a dummy program		1.314	1.564	±0.058	1.833
copying a 14kB file with `fstream`		0.268	4.102	±1.676	22.780
get a file from KVS (CCR)		85.269	86.509	±2.163	119.400
ping server		0.124	0.179	±0.124	4.040
TLS 1.2 time to first byte		92.400	99.265	±2.145	106.400
cached key retrieval	measured on server	0.200	0.297	±0.050	0.549
	measured from client	0.400	0.504	±0.040	0.600
cold key retrieval	from request to response	88.400	90.057	±0.755	92.000
	total	176.100	187.942	±3.622	192.700

All the timings are assembled in Table 4.1. To sum up the composition of the latency to fetch a key, the TLS-handshake and the call to CCR dominate. The key management logic and network round trip times are negligible.

4.7.3 Consistency

To verify the extension of the service to multiple machines, another test is required. It is important to assess the consistency of operations in the cluster. As we rely on an external distributed key value service, we have no influence on its performance.

The experimental set up includes two servers where one serves keys to a client and the other is used to inject updates into the KVS. Our key cache holds keys for 15 minutes which by far exceeds the time of the KVS spreading the update.

By querying for non-cached keys, we can force our key manager to fetch the current key store from the KVS. In our test case, we used the GPFS internal CCR as KVS. With an update to the CCR by the key management tool every 100 ms we saw no impact on the retrieval of keys. This update rate is far above one in a practical setup, because for the rotation of many keys, the keys can be processed together in one single batch which results in a single update to the KVS.

4.8 Outlook

This key manager supports a minimalistic set of necessary operations to run as a drop-in replacement for ISKLM in a GPFS cluster. However the long term goal of this project is to provide an even better and more secure key management service. Some future features are discussed below.

4.8.1 Per Node Asymmetric Key-Pair

In our current scenario, compromising one of the key servers or stealing an REK in combination with any KVS participant immediately discloses all keys inside the store. This could be improved by having a private/public key pair for each node. Then the keys in the store can be in wrapped form for each client individually by its respective public key. The additional complexity would allow to have fine grained access control on a per client basis and still maintain the central approach where keys can be definitely deleted.

A major problem of this model is the need for a new set of key management operations to manage the asymmetric key pairs on each client. We decided against this implementation, because it would no longer be compatible with the targeted KMIP clients.

4.8.2 Protection of Keys in Memory

To handle plaintext keys as sensitive as possible, we first of all need to make sure that they never leave the volatile memory, e.g., by swapping them on disk. This is achieved by locking the pages in memory. Although there are attacks on RAM after shut down [28] we are more concerned by on-line attacks.

The most probable leakage of sensitive memory regions is by a heap based buffer overflow. To reduce the attack surface, it is possible to hold the keys in inaccessible pages. This can be achieved by setting the page's rights to `NONE` in the memory management unit. Legitimate accesses to the keys are performed via a small API, which sets the needed rights, performs the wanted action and locks the keys afterwards. Accidental direct access will result in a segmentation fault.

An overflow vulnerability can still exist and it is possible to beat the odds and read the keys while they are unlocked. Nevertheless each failure will crash the daemon and successive restarting will be noticed by a monitoring system.

4.8.3 Access Control

Our key manager has no access control yet, as anyone with a valid client certificate has access to all keys. Like the ISKLM we could use the KMIP in line authentication mechanisms to separate keys of different tenants. But it is preferred to run two independent LKMS services with different REKs and a different KVS name space on the cluster.

4.9 Final remarks

As encryption of data at rest becomes ever more prevalent, the challenge of managing the encryption keys also surfaces for more and more different kinds of systems. For large but less mission critical deployments, justifying an investment in an enterprise-grade key management solution leveraging hardware security modules may be difficult. This is a scenario where our key manager comes in: built on top of an untrusted key value store (KVS) in the IBM GPFS cluster file system, it provides a scalable and distributed key management solution, serving encryption keys to GPFS using the standard Key Management Interoperability Protocol (KMIP) for file encryption. The keys in the encryption hierarchy are stored wrapped in the untrusted key value store, and they are unwrapped by the GPFS daemon using root encryption keys stored locally in removable and securely erasable media on each storage node in the GPFS cluster. This hierarchical encryption key architecture allows for key rotation and secure, cryptographic deletion of data.

Evaluation shows that our prototype was able to linearly scale out even under load from key updates, and performance measurements conducted on the individual components of our solution indicate that the throughput and latency is bound by the TLS handshake procedure when setting up the client key retrieval connection with the server, as well as the performance of the distributed KVS.

5. Conclusions

The deliverable reported on several innovations that contribute to the advancement of the protection of cloud stored data.

The analysis of the different options for the application of over-encryption described in Chapter 2 offers a contribution that is directly implemented in Swift, one of the most used open-source cloud storage solutions, but the approaches that have been considered for Swift have a clear immediate application also to other solutions that support data storage.

In Chapter 3 we presented a dynamic tree-based data structure for storing resources at an external server and guaranteeing access privacy. Our approach does not require maintaining any storage at the client side. The advantage of being stateless, besides not requiring the client to commit resources, also permits to accommodate multiple clients and provides resilience of the structure against failures or unavailability of the client. The dynamic restructuring of the tree at both the logical and physical level provide access privacy, making the frequency distribution of accesses to the physical blocks indistinguishable from the one produced by a uniform access profile.

Chapter 4 has described a scalable key manager, which uses an untrusted distributed key-value store (KVS) and is accessible over the Key-Management Interoperability Protocol (KMIP). It implements a key hierarchy and focuses on providing scalable performance that is suitable to serve keys at very high rate.

Overall, the design of techniques able to enforce confidentiality of outsourced data has the potential to greatly accelerate the rate of adoption of cloud storage, leading it to become the standard approach for the management of any kind of data. Local storage and traditional file systems would then only play the role of a cache that speeds up access to data, but persistence would be guaranteed by cloud providers. The work presented in this deliverable aims at accelerating the realization of this vision.

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] Amazon CloudHSM. `https://aws.amazon.com/cloudhsm/`.

[4] Amazon Key Management Service. `https://aws.amazon.com/kms/`.

[5] OpenStack Barbican. `https://wiki.openstack.org/wiki/Barbican`.

[6] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang. Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward. In *Proc. of CCS*, Denver, CO, October 2015.

[7] M. Björkqvist, C. Cachin, R. Haas, X. Hu, A. Kurmus, R. Pawlitzek, and M. Vukolić. Design and implementation of a key-lifecycle management system. In S. Radu, editor, *International Conference on Financial Cryptography and Data Security*, pages 160–174. Springer, 2010.

[8] C. Cachin, K. Haralambiev, H. Hsiao, and A. Sorniotti. Policy-based secure deletion. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 259–270, New York, NY, USA, 2013. ACM.

[9] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proc. of EUROCRYPT*, Prague, Czech Republic, May 1999.

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

[11] J. Dautrich and C. Ravishankar. Tunably-oblivious memory: Generalizing ORAM to enable privacy-efficiency tradeoffs. In *Proc. of CODASPY*, San Antonio, TX, March 2015.

[12] 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.

[13] 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.

[14] 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.

[15] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Encryption policies for regulating access to outsourced data. *ACM TODS*, 35(2):12:1–12:46, April 2010.

[16] 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, June 2011.

[17] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Supporting concurrency and multiple indexes in private access to outsourced data. *JCS*, 21(3):425–461, 2013.

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

[19] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Three-server swapping for access confidentiality. *IEEE TCC*, 2016. pre-print.

[20] S. Devadas, M. van Dijk, C. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In E. Kushilevitz and T. Malkin, editors, *Proc. of TCC*, 2016.

[21] G. Di Crescenzo, N. Ferguson, R. Impagliazzo, and M. Jakobsson. How to forget a secret. In C. Meinel and S. Tison, editors, *Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1563 of *Lecture Notes in Computer Science*, pages 500–509. Springer, 1999.

[22] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[23] M. Dworkin. Sp 800-38d. Recommendation for Block Cipher Modes of Operation: Galois/Counter mode (GCM) and GMAC. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2007.

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

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

[26] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. Weinberger. Quickly generating billion-record synthetic databases. In *Proc. of SIGMOD*, Minneapolis, MN, 1994.

[27] H. Hacigümüs, B. Iyer, S. Mehrotra, and C. Li. Executing SQL over encrypted data in the database-service-provider model. In *Proc. of SIGMOD*, Madison, WI, June 2002.

[28] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, May 2009.

[29] IBM Key Protect. https://console.ng.bluemix.net/catalog/services/key-protect/.

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

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

[32] P. Lin and K. Candan. Hiding traversal of tree structured data from untrusted data stores. In *Proc. of WOSIS*, Porto, Portugal, April 2004.

[33] OASIS Key Management Interoperability Protocol Technical Committee. Key Management Interoperability Protocol Version 1.2, 2015. OASIS Standard, available from http://www.oasis-open.org/committees/documents.php?wg_abbrev=kmip.

[34] R. Ostrovsky and W. E. Skeith, III. A survey of single-database private information retrieval: Techniques and applications. In *Proc. of PKC*, Beijing, China, April 2007.

[35] L. Ren, C. Fletcher, X. Yu, A. Kwon, M. van Dijk, and S. Devadas. Unified oblivious-RAM: Improving recursive ORAM with locality and pseudorandomness. *IACR Cryptology ePrint Archive*, 205, 2014.

[36] O. Sefraoui, M. Aissaoui, and M. Eleuldj. OpenStack: Toward an open-source solution for cloud computing. *IJCA*, 55(3):38–42, 2012.

[37] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[38] E. Stefanov and E. Shi. ObliviStore: High performance oblivious cloud storage. In *Proc. of IEEE S&P*, San Francisco, CA, May 2013.

[39] E. Stefanov, M. van 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.

[40] OpenStack Swift at-rest encryption. http://specs.openstack.org/openstack/swift-specs/specs/in_progress/at_rest_encryption.html.

[41] Vormetric Data Security Management. https://www.vormetric.com/products/data-security-manager.

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

[43] X. Wen, G. Gu, Q. Li, Y. Gao, and X. Zhang. Comparison of open-source cloud management platforms: OpenStack and OpenNebula. In *Proc. of FSKD*, Sichuan, China, May 2012.

[44] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *Proc. of CCS*, Alexandria, VA, October 2008.

[45] 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.

[46] 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, USA, March 2010.