ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für
Technische Informatik und
Kommunikationsnetze

Access Control in Peer-to-Peer Storage Systems

Erol Koç

Master Thesis

October 1

Communication Systems Group
Department of Information Technology and Electrical Engineering

ETH Zurich

Supervised by:
Dipl.-Ing. Marcel Baur
Dr. Germano Caronni

Group of:
Prof. Dr. Bernhard Plattner

MA-2006-10
SS 2006

Acknowledgments

I am grateful to Marcel Baur for being an excellent supervisor, helping me whenever I needed it and spending lots of time for reviewing my ideas, drafts and the final text as well as helping with writing shell scripts. I thank Germano Caronni for motivated and regular discussions that helped to improve the design of the schemes significantly. Moreover, I thank Prof. Bernhard Plattner for his useful comments and thoughts.

Without the help of my numerous reviewers, the text would still be erroneous. I want to thank all of them[1] for their patience and constructive comments: Gabor Cselle, Markus Egli, Remo Marti, Yvonne Anne Oswald, Raphael Woy and Daniel Zeiter. Further, I am thankful to Stefano Tessaro, Vassilis Zikas and Prof. Roger Wattenhofer for their general remarks and hints regarding consensus protocols. The IT-Support-Group (ISG.EE) of the Department of Information Technology and Electrical Engineering (D-ITET) helped me with infrastructure issues and made it possible to run complex and resource intensive tests on their Linux machines. Michael Grossniklaus from the Globis group gave me the kind permission to use their LaTeX template for the thesis. Finally, I thank Sun Microsystems Laboratories for giving me the opportunity to participate in the Celeste project.

[1] In alphabetical order.

Abstract

Access control in storage systems ensures that only properly authenticated and authorized subjects can read or modify objects. In traditional systems, access control is based on a trusted infrastructure such as the local operating system or a central server. Those authorities have the competence to decide on the legality of an access attempt. In a peer-to-peer (P2P) system, access control can no longer be based on trust: Peers are unreliable and can behave arbitrarily malicious. In the context of a distributed object store on basis of a P2P system, it is inevitable to protect the stored objects. Further, the volatile characteristics of a P2P system require concepts that are able to tolerate node failures or voluntary quitting.

This thesis describes mechanisms to avoid the trusted authority by distributing the competence for access control decisions among several entities called *gatekeepers*. These gatekeepers form a distributed reference monitor that can tolerate malicious nodes and failures. They control read and write operations and guarantee freshness[2] of objects. The reference monitor must be efficient such that it can be used in practical systems. In particular, changes on the authorized entities must be feasible even for large groups of peers. The efficiency of different schemes is theoretically described and analyzed. An important contribution is the description of a tree data structure that manages keys for read access efficiently. On the average, the asymptotic runtime of the tree structure is linear or logarithmic. Measurements of the implementation of the tree underline the theoretical estimations.

Moreover, a mechanism for self-organizing gatekeepers is explained, which allows them to choose the members of their group autonomously. Since the members of the gatekeepers can change over time, readers and writers must be able to determine the latest set of gatekeepers. This thesis describes a novel and secure way to locate the latest gatekeepers without interaction with any other authority. In principle, readers and writers can efficiently compute the new set of gatekeepers on their own.

[2]Freshness of an object means that a reader can be sure to always read the latest version. This is important because an attacker could inject old versions and therefore manipulate the semantics of a read operation.

Zusammenfassung

Durch Access Control in Speichersystemen wird sichergestellt, dass nur authentisierte und autorisierte Subjekte bestimmte Objekte lesen oder schreiben können. Access Control basiert in traditionellen Systemen auf einer vertrauenswürdigen Infrastruktur wie einem Betriebssystem oder einem zentralen Server. Diese Autoritäten haben die Kompetenz um über Zugriffe zu entscheiden. Access Control kann in einem Peer-to-Peer (P2P) System nicht mehr auf Vertrauen beruhen: Die einzelnen Peers sind unzuverlässig und können sich beliebig bösartig verhalten. In Zusammenhang mit einem verteilten Object Store ist es umso wichtiger, die gespeicherten Objekte zu schützen. Zudem bedingt das unberechenbare Verhalten eines P2P Systems Konzepte, welche Ausfälle von Peers oder absichtliches Verlassen des Systems tolerieren können.

Diese Arbeit beschreibt Mechanismen zur Umgehung der vertrauenswürdigen Instanz, indem die Kompetenz für Zugriffsentscheidungen über mehrere Entitäten - sogenannte *Gatekeeper* - verteilt wird. Diese Gatekeeper bilden einen verteilten Reference Monitor, welcher in der Lage ist, bösartige Peers oder Ausfälle zu verkraften. Sie kontrollieren Zugriffe bei Schreib- und Leseoperationen und garantieren Freshness[3] der Objekte. Das Ziel ist es, den Reference Monitor effizient zu gestalten damit ein praktischer Einsatz möglich ist. Insbesondere soll es möglich sein, Gruppenzugehörigkeiten effizient abzuwickeln, auch für grosse Gruppen von Peers. Verschiedene Ansätze für die Verwaltung der Leser-Gruppe werden im weiteren beschrieben und analysiert. Ein wichtiger Beitrag ist die Beschreibung einer Baumstruktur zur effizienten Verwaltung von Schlüsseln, die für Lesezugriffe benötigt werden. Durchschnittlich verhält sich die asymptotische Laufzeit der Baumstruktur linear oder logarithmisch. Die Messwerte, welche durch die Implementation dieses Verfahrens erhalten wurden, unterstreichen diese Behauptung.

Zusätzlich wird ein Mechanismus erklärt, welcher den Gatekeeper erlaubt, die Mitglieder ihrer Gruppe selbständig zu bestimmen. Da sich die Zusammensetzung der Gatekeeper über die Zeit verändern kann, müssen Leser und Schreiber in der Lage sein, die aktuellste Menge von Gatekeeper zu bestimmen. Diese Arbeit beschreibt ein neuartiges und sicheres Verfahren zur Lokalisierung der aktuellen Gatekeeper ohne Interaktion mit einer anderen Authorität. Es ist lesenden und schreibenden Gruppenmitgliedern möglich, das neuste Set von Gatekeeper selbständig und effizient zu berechnen.

[3]Freshness bedeutet, dass ein Leser immer die neuste Version eines Objektes ermitteln kann. Dies ist wichtig da ein Angreifer, der alte Versionen einem Leser zuspielt, die Semantik von Leseoperationen verändert.

Contents

1 Introduction

Storage is an important resource. But more important than storage itself is the information that resides on the storage devices. Information differs greatly from physical resources. It can be copied, moved and deleted without cost and it can be modified very rapidly. In many cases, the physical whereabouts of information are unknown, or there might be several places at the same time where it can reside. As for all precious resources, not all subjects are allowed to access all kinds and parts of all information. The volatile nature of information requires powerful and efficient mechanisms for its protection. Information security seeks for means to protect information over its lifetime using *access control*. Access control is defined as a mechanism used to specify and restrict the rights of authorized users, application programs, systems, or processes to information system resources [1].

Definition *access control*

An illustrative example for access control are passengers boarding an airplane: They must exhibit a valid board ticket, prove their identity using a passport and have their luggage examined by different instances such as a metal detector, x-ray scanner, security staff and police dogs. Only passengers that pass all tests are allowed to board the airplane.

Access control in a computer system is comparable to the airplane example in terms of complexity. Security is established by both cryptographic means and infrastructure. Cryptography can be used for two purposes: *Authenticity* and *secrecy*. Authenticity allows to verify whether some content has been modified. Secrecy aims at protecting the content of information such that unauthorized subjects cannot infer the content. Authenticity and secrecy are distinct concepts.

Authenticity, secrecy

Information can be protected in terms of secrecy but must not be authentic and vice versa. An infrastructure is required to explicitly control which type of access is allowed[4].

Trust is mandatory

Another important aspect of security is *trust*. Trust is not a mathematical concept. Each human individual has a different association with trust. Access control relies on trusted authorities, platforms or components. In the airplane example, there is the implicit assumption that the security staff and their equipment is reliable and trustworthy. Without trust, the control mechanisms are obsolete since possibly unauthorized subjects may pass the control unharmed. However, the airplane example nicely shows that trust can also be distributed among several instances. If one instance (say the x-ray scanner) fails, the other instances still can guarantee security of the plane, a fact that the schemes of this thesis highly rely on.

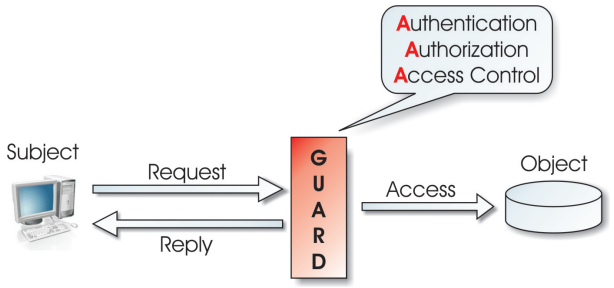

Figure 1.1: Schematic and abstract illustration of access control.

Reference monitor

Authentication, authorization, enforcement

Schematically, access control can be described as illustrated in Figure 1.1. A subject issues a request to a *reference monitor* or *guard* which verifies the clearance of the subject and grants access to an object or a resource if the subject is authorized. Even at this high level of abstraction, the complexity of access control is perceivable since the process can be decomposed into *authentication*, *authorization* and *enforcement*. Authentication allows the reference monitor to determine who is requesting access. With authorization, the reference monitor determines the resources that the authenticated subject may access and its permissions. Finally, there is a need for enforcement mechanisms since the reference monitor must be able to prohibit improper access attempts. Moreover, those steps can be carried out in cooperation with other authorities and dedicated servers.

The scheme in Figure 1.1 is general in the sense that it applies to local, centralized and decentralized distributed systems. As an example, a local reference monitor is the operating system which guards access to files in the file system according to the permissions of the files. In a centralized distributed system like it is common for banking infrastructures, a dedicated authority is contacted on an access attempt to

[4]The different types of access include amongst others read, write, append, and delete operations.

verify the credentials[5]. In a decentralized distributed system, the lack of designated authorities requires to distribute the tasks of the reference monitor among equal hosts.

1.1 Scope

The goal of this thesis is to develop a scalable concept for access control in a P2P storage system. One main focus is on an adequate group key management scheme to efficiently handle large and changing populations of users with varying data access rights. Moreover, the lack of designated authorities requires new paradigms to realize efficient access control mechanisms even in a highly dynamic environment where peers can join and leave the network arbitrarily.

Group key management scheme

Since cryptography on its own is insufficient for controlling write access, a novel scheme is presented that handles write requests efficiently and correctly. Traditional systems maintain a central trusted authority which is responsible for granting access based on password or certificate authentication schemes. Besides that such an entity does not exist in P2P systems, it is fundamental to reduce trust in a single peer to an absolute minimum since there is no control or prediction of the behavior of peers.

Minimize trust, no central authorities

Although the thesis is based on storage systems, it does not intend to explain related technologies or methods. It is assumed that a storage system with certain properties exists.

This thesis intends to be readable by a broad audience. It shortly highlights the most important concepts of cryptography before explaining how access control is carried out. The numerous references allow to deepen the topics where desired. For researchers in the field, it provides a valuable insight into the area of P2P security with all its rough edges.

1.2 Organization and Overview

The thesis is organized in chapters. A chapter may depend on the definitions or considerations of previous chapters. The core of the thesis is Chapter 5 which explains access control for a P2P storage system in-depth.

Chapter 2 highlights the most important concepts and definitions of cryptography. The chapter focuses on access control and its various techniques, but also covers secret sharing schemes as they are fundamental for one of the presented access

[5] This does not mean that there is only one dedicated authority. The authorities can be replicated and can communicate with other servers. However, they have been configured appropriately and cannot be replaced by another arbitrary host.

control schemes. Since a P2P system must cope with large numbers of peers, a separate section on reflections about groups is appended.

Chapter 3 explains the latest research proposals that have been carried out in the area of P2P storage system security.

Chapter 4 builds the basis for P2P access control. It describes the underlying object store with its requirements and functions. The system entities and their roles are introduced and the assumptions about the adversary's capabilities are specified.

Chapter 5 is the core chapter of this thesis. It contains detailed descriptions about the access control mechanism and distinguishes between the three possible schemes for read access control. The chapter is concluded with a complexity analysis.

Chapter 6 gives an overview of an implementation in Java on the basis of an existing P2P storage system.

Chapter 7 highlights and comments the efficiency of the implementation and compares it to the original performance of the system.

Chapter 8 concludes this thesis and gives an outlook of the future work to be done in this area.

1.3 Contributions

The contribution of this thesis is the description of a concrete and practical mechanism for access control in a P2P storage system. There are no dedicated authorities and trust has been reduced to an absolute minimum. The basis of the system is a distributed reference monitor that guards access to objects.

A novel type of identifier called *secure version identifier* (SVID) allows to determine the entities of the distributed reference monitor efficiently by using only local computations.

Since different storage systems have distinct policies or performance requirements, this thesis does not only focus on one mechanism, but explains read access control with three different approaches: A list-based, a tree-based and a sharing-based scheme. While the list- and tree-based schemes share some common characteristics, the sharing-based approach is completely orthogonal.

Finally, a proof-of-concept in form of a prototypical implementation gives information about the behavior and performance of the mechanisms.

2 Background

To understand access control in a peer-to-peer system, some background knowledge about security and cryptography is required. The chapter gives a broad overview and does not intend to explain all mechanisms in detail, but to sketch the most important principles in cryptography. The subsequent chapters of this thesis are based on cryptographic schemes such as encryption and digital signatures. Where necessary, a more detailed definition or references to adequate literature will be given. For a more comprehensive description of cryptography, we recommend 'Cryptology' by R. L. Rivest [6].

2.1 Information Security

Information is an important resource

Protection of information, one of the most important global resources, is a major issue in the information economy. Information differs radically from other resources. For instance, it can be copied with almost no cost, it can be deleted without leaving traces and it can be altered at a high frequency. Hence, the way how information is treated requires sophisticated and well-understood mechanisms. Some of the reasons why information security is such a hot topic include the large number of vulnerabilities and security incidents, the lack of products and standards as well as the lack of understanding of security and cryptography in general. Further, lots of companies and individuals tend to ignore security and risk to expose confidential or private information. Due to frequent headlines in the media about security breaches and Internet attacks, security has become a permanent issue.

Fundamental problems in information security

But there are two fundamental problems in information security. First of all, it is generally impossible to define *good* and *bad* precisely, especially in the context of software. Second, it is in general undecidable to distinguish between *good* and *bad* software[6]. Those criteria emphasize why information security is a necessary and difficult area of computer science.

2.2 Security Objectives

Definition *CIA*

In information security literature, there are three fundamental security goals known as *CIA*:

- **Confidentiality** or **secrecy**: access to information should only be granted to authorized individuals.
- **Integrity**: information should only be modifiable by individuals with proper authorization.
- **Availability**: information should be available when needed.

This classification is quite simplistic and has been extended with a set of different security goals including *non-repudiation*, *auditability*, *accountability* or *privacy*. There is often confusion about the difference between integrity and *authenticity*. In the context of communication, integrity means that the content of a message has not been altered. Additionally to the property of integrity, authenticity includes that metadata such as sender or receiver and all other parameters of the message have not been altered.

2.3 Cryptography

What is cryptography?

Cryptography is the way of securing information on the basis of mathematical operations and number theory. Many of the results in cryptography are highly surprising, such as public-key cryptography. All schemes are somehow based on random numbers or mathematically hard problems. In this section, the most important cryptographic principles are explained along with precise definitions where useful.

2.3.1 Perfectly-Secure Cryptography

Unbreakable cryptography

The theory of perfectly-secure or information-theoretically secure cryptography provides means to make it impossible for an adversary to compute an encrypted secret even if the adversary is provided with unbounded computing power. This is a consequence of the fact that the encrypted secret is statistically independent from the plaintext, which makes it impossible to gain any inference about the secret. The *one-time pad*, initially invented in 1917 by Gilbert Vernam [3], has

One-time pad

[6] The undecidable halting problem is a special case of the problem of deciding whether a program meets a given specification.

been proven to be perfectly secure if - as the name suggests - only used once. The information-theoretic basis for the proof has been given by Claude Shannon's information theory [4] which was later refined by Hellman [5]. A one-time pad is a random bitstring and has the same length as the data to be encrypted. En- and decryption are performed by adding the one-time pad modulo two (XOR) to the data bitstring.

Although one-time pads provide provable security, they are rarely used in practice since they are awkward for obvious reasons: First, the length of the key must be equally long as the data that needs to be encrypted. Secondly, a key can only be used once to de- and encrypt information. And finally, the pad needs to be randomly generated using a cryptographically secure pseudo random generator [15, 16, 17]. Although beyond the scope of this thesis, it is noteworthy that from a philosophical point of view, it is even questionable whether randomness really exists or not.

2.3.2 Computationally-Secure Cryptography

For the reasons mentioned, one weakens the security constraints and moves from perfectly-secure cryptography to a less rigorous model which is only computationally secure. This means that the cryptographic operations are believed to be hard to break or in other words: It is infeasible to break security within reasonable time. Two famous operations which are believed to be computationally hard are the discrete logarithm problem [13] and the prime factorization of large numbers [14]. However, the fact that no algorithm is currently known does not imply that there does not exist such an algorithm at all. Note that under the assumption that quantum computers exist, these computations have been proven to be efficiently computable in polynomial time [9]. For the remainder of this thesis it is assumed that presently used cryptographic schemes which are based on computationally hard problems are secure as long as neither the secret nor private key is leaked to the adversary.

What does *computationally secure* mean?

2.3.3 Symmetric Cryptography

A *symmetric cryptosystem*, also called a *cipher*, uses the same secret key for de- and encryption. The formal definition is given as follows:

Definition *symmetric cryptography*

Definition 2.3.1 *A* cipher or symmetric cryptosystem *consists of*

- *A message space* $\mathcal{M}$ *over some alphabet* A_1.
- *A ciphertext space* $\mathcal{C}$ *over some alphabet* A_2.
- *A key space* $\mathcal{K}$ *over some alphabet* A_3.
- *An encryption function E:* $\mathcal{M} \times \mathcal{K} \rightarrow \mathcal{C}$.
- *A decryption function D:* $\mathcal{C} \times \mathcal{K} \rightarrow \mathcal{M}$, *such that D(E(m, k), k) = m for all* $k \in \mathcal{K}$ *and* $m \in \mathcal{M}$.

An extension to this definition can be given by allowing the encryption to be nondeterministic using a randomness space $\mathcal{R}$ such that E: $\mathcal{M} \times \mathcal{K} \times \mathcal{R} \rightarrow \mathcal{C}$ while decryption remains unchanged to the definition given above. In general, every practical cryptosystem can be modeled as a finite automaton processing messages in units of certain size. This leads to three special instantiations which are used in practice:

- A *block cipher* is a stateless encryption where plaintext and ciphertext are n-bit strings. A message is divided into n-bit blocks and each is separately encrypted.
- An *additive stream cipher* is similar to a one-time pad where the random key is replaced by the output of a pseudo-random generator, which is added modulo 2 (XOR) to the plaintext. Unlike the block cipher, an additive stream cipher is stateful as it maintains the state of the pseudo-random generator.
- In a *self-synchronizing stream cipher*, encryption is performed bit-by-bit. The state consists of the m most recent ciphertext bits. Decryption is self-synchronizing meaning that it can start at an arbitrary point in time without the need for context information.

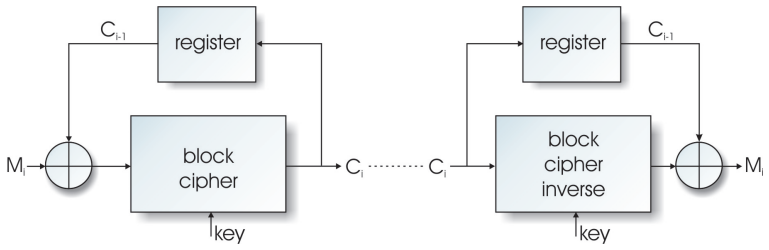

Figure 2.1: Cipher block chaining mode (CBC). Encryption (left) and decryption (right) are blockwise, where M_i denotes the i^{th} plaintext block and C_i the corresponding ciphertext block.

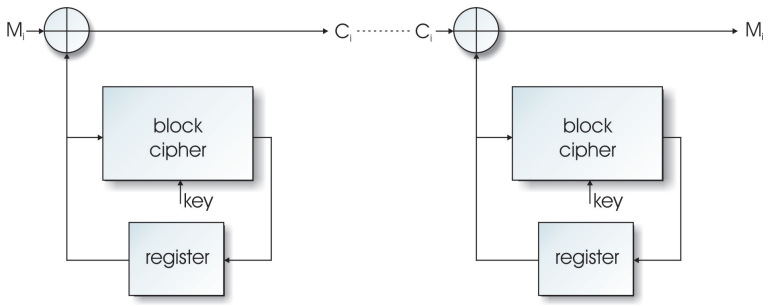

Figure 2.2: Output feedback mode (OFB). The block cipher is used as a pseudo-random generator producing blocks of pseudo-random bits.

Block ciphers can be used to construct new encryption functions, called *modes of operation*. Some well known modes are the *electronic codebook mode (ECB)*, *cipher*

block chaining mode (CBC, see Figure 2.1), *output feedback mode (OFB, see Figure 2.2)* and the *cipher feedback mode (CFB)* [26].

2.3.4 Message Authentication Codes

While symmetric encryption protects the confidentiality of information, message authentication codes[7] (MAC) protect the integrity of information and make modifications detectable.

Definition *MAC*

Definition 2.3.2 *A message authentication code (MAC) for message space $\mathcal{M}$, key space $\mathcal{K}$ and tag space $\mathcal{T}$ is a function $f : \mathcal{M} \times \mathcal{K} \to \mathcal{T}$ such that the following security condition holds: Let k be chosen uniformly from $\mathcal{K}$. There exists no efficient algorithm, with access to an oracle $\mathcal{M} \to \mathcal{T}$ computing the tag $f(m, k)$ for an input message m, which outputs with non-negligible probability a message m' different from all messages asked to the oracle as well as the corresponding tag $t = f(m', k)$ [7].*

2.3.5 Asymmetric or Public-Key Cryptography

In contrast to symmetric cryptography, public-key or asymmetric cryptography does not require to share a secret key between two entities. Instead, a private key is generated and stored only on one entity while the public-key is publicly known. The secret allows an entity to perform certain operations exclusively, hence the term asymmetric. Concretely, the public key is used to encrypt data and to verify digital signatures while the private key can be used for decryption and for generating digital signatures. Note that the term *private key* is used in the context of public-key cryptography while *secret key* applies to symmetric cryptography.

Why *asymmetric*?

Asymmetric cryptography is much more challenging than symmetric cryptography because of the paradoxical asymmetric property which requires certain mathematical structures, for example an algebraic group with special properties. As there is no general design criterion, only a small number of public-key cryptosystems have been proposed, one of them being RSA [27]. Public-key cryptographic mechanisms are used for a wide range of cryptographic protocols including identification protocols, bit commitment schemes, interactive proofs, payment systems, and secure multi-party computation.

2.3.6 Certificates

A *certificate* represents a binding of a public key to an entity which is digitally signed by a so-called certification authority. More concretely, a certificate consists of a data part followed by a signature on the data part. The data part includes at least a public key, the name or identifier of the entity for which the certificate is issued, the name of the entity that issued the certificate and possibly further

Content of certificates

[7] Sometimes also called message integrity code (MIC) or *tag*.

parameters like expiration date and so forth.

A certificate $Cert_{C,A}$ signed by an authority C allows an entity B to obtain an authenticated copy of A's public key. There are two conditions that must hold:

1. B must hold an authentic copy of C's public key in order to be able to verify the signature on the certificate.
2. B must trust C not to sign certificates for unauthenticated entities.

At first sight, it seems not to be a real advantage to use certificates as the problem of obtaining an authentic copy of some public key has been transformed to the problem where an authentic copy of an authority's public key as well as trust in the authority itself is required. However, the idea is that it might be much easier to verify the authenticity of the authority's public key.

2.3.7 One-way Functions and Hash Functions

Informally, a one-way function is a function which is easy to compute but hard to invert. More formally:

Definition *one-way function*

Definition 2.3.3 *A* one-way function *is an efficiently computable function f from a domain A to a co-domain B, $f : A \rightarrow B$, such that for every efficient (possibly probabilistic) algorithm G, taking an input from B and producing an output in A, and for $x \in A$ selected uniformly at random*

$$P\left(f\left(G\left(f\left(x\right)\right)\right) = f\left(x\right)\right)$$

is negligible [7].

It is still unproven that one-way functions exist for reasonable definitions of computational hardness and of negligible probability. Hash functions are another kind of functions that are used to map a large bitstring to a small bitstring of fixed size.

Definition *hash function*

Definition 2.3.4 *A* hash function *is an efficiently computable function $h : D \rightarrow R$ where $|D| \gg |R|$, typically $D = \{0,1\}^*$ and $R = \{0,1\}^k$ for some suitable k. A hash function can have a parameter c from some set $C = \{0,1\}^s$ for some s, selecting a function h_c from a class $\{h_c : c \in C\}$ of functions [7].*

Cryptographic hash functions aim at preventing collisions from happening even if an adversary tries to enforce them. Therefore, a hash function is often also a one-way function, which nevertheless is mostly insufficient for many applications, yielding to the notion of *collision-resistant* hash functions. Informally, they can be defined as follows: It is computationally infeasible to find two distinct bitstrings x and x' such that $h_c(x) = h_c(x')$. Or more formally:

Definition 2.3.5 *A hash function class* $\{h_c : c \in C\}$ *with domain* D *is* collision-resistant *if for every efficient algorithm* G *taking an input* $c \in C$ *and producing a pair (x, x') of values in* D*,*

Definition *collision-resistance*

$$P\left(h_c\left(x\right) = h_c\left(x'\right)\right)$$

is negligible for $c \in C$ *selected uniformly at random [7].*

Cryptographic hash functions can be used for various operations such as hashing of messages before signing them, generation of pseudo-random bits and message authentication codes. The *secure hash algorithm* SHA [50] is widely used as cryptographic hash function.

2.4 Secret Sharing Schemes

A secret sharing scheme provides means to distribute a secret value s (for example a cryptographic key) among several entities $P_1, ..., P_n$ such that only those entities can reconstruct s and all other entities have no information about the secret. The piece of information that a single entity P_i holds is known as *share* s_i. Most sharing schemes are known to be information-theoretically secure. This means that each share s_i is statistically independent of the secret s.

What is a secret sharing scheme?

Typically, a secret sharing scheme maintains a threshold where arbitrary k out of n entities knowing a share s_i can reconstruct s, but $k-1$ entities are not sufficient to retrieve any information. Such a scheme is called (k, n)-threshold scheme. The entity responsible for distributing the shares is called *dealer*.

Definition 2.4.1 *A* secret sharing scheme *consists of a pair of efficient protocols:*

Definition *secret sharing scheme*

- *A protocol for sharing a secret value s. The protocol is probabilistic and uses secret* [8] *channels to distribute the shares to all participating entities.*
- *A protocol for reconstructing the secret s from a set of shares*[9] *[8].*

In the following, the Shamir's scheme is presented which will also be used in a later chapter.

2.4.1 Shamir's Scheme

Shamir's secret sharing scheme [18] is a threshold scheme based on polynomial interpolation. It uses modular arithmetic over a finite field $GF(q)$. The dealer chooses the coefficients $a_1, ..., a_{k-1}$ in $GF(q)$ randomly from a uniform distribution over integers in $[0, q)$. He then makes use of the polynomial

Sharing through polynomials

[8] A *secret* channel is both confidential and authentic.

[9] In the simplest case, each entity broadcasts its share to all other entities and each entity then reconstructs the secret locally.

$$a(x) = s + a_1 x + ... + a_{k-1} x^{k-1}$$

with s being the secret to be shared and distributes the shares $s_i = a(\alpha_i)$ to each participating entity P_i. Note that the points α_i at which the polynomial is evaluated are publicly known. However, as the α_i have to be each distinct, it must hold that $q > n$. The secret is reconstructed by polynomial interpolation and evaluation of $a(x)$ at $x = 0$.

Theorem 2.4.1 *Arbitrary $k-1$ shares do not reveal any information about the secret s.*

Proof Assume that $k-1$ shares are revealed to some adversary. For each possible secret value s' in $[0, p)$, there is exactly one polynomial $a'(x)$ of degree $k-1$ such that $a'(0) = s'$. As all p values in $[0, p)$ are equally likely, the adversary cannot obtain any information about s [18].

2.5 Access Control

Definition *access control*

Access control is the protection of the system against unauthorized access. It is a process by which the use of system resources is regulated according to a security policy and is permitted only to authorized entities (users, programs, processes or other systems) according to that policy.

Typical access control models actually focus on authorization, for example to specify the permissions of a subject. The specification can be done using matrices, lattices or other mathematical structures, which specify which rights subjects have on objects. Additionally, access control focuses on enforcement mechanisms, identification and authentication, and accountability. Identification and authentication determine who can log on to a system, authorization determines what an authenticated user can do, and accountability identifies what a user did.

2.5.1 Authentication

Definition *authentication*

Authentication is the process of verifying the validity of something claimed by an entity. The default assumption for the claim is the identity. Authentication can be proven by:

- Knowledge of a shared secret such as a password or a personal identification number (PIN).
- Ownership of a smart card or token.
- An attribute such as fingerprint, voice, retina or iris characteristics.

2.5.2 Identification

Identification is the process of associating an identity with a subject. The identification component of an access control system is normally a relatively simple mechanism based on either user name, user ID or certificate. Proper identification requires unambiguous and unique identification of a subject.

Definition *identification*

2.5.3 Authorization

An *authorization* is a right or a permission that is granted to an entity to access a resource. It can be implemented using role based access control [29] [30], access control lists [28] or a policy language (such as XACML [31]). After an entity is authenticated, authorization determines what that entity can do on the system. Additionally to the three basic permissions read, write and execute, there can be privileges such as append, modify attributes, move, copy, and so forth. Most modern operating systems define sets of permissions that are variations or extensions of three basic types of access: read, write and execute.

Definition *authorization*

2.5.4 Access Matrix

The first to describe *access matrices* was B. Lampson in "Protection" [28]. The need for protection mechanisms arises from the fact that one should try to keep one user's malicious or erroneous behavior from harming other users. This led to the

Object / Subject	O_1	O_2	O_3
S_1	read		
S_2	write		write / execute
S_3		execute	read / write

Figure 2.3: Access matrix with three objects and three subjects having distinct permissions.

development of the so-called access control matrix model which determines the access of subjects to objects, as shown in Figure 2.3. The rows of the matrix are labeled with subject names and its columns with object names. Element $A_{i,j}$ specifies the access which subject i has to object j. Each element consists of a set of strings called access attributes which are identical to permissions like read, write, execute and so forth. Requests from subjects to access objects are guarded by a reference monitor which examines each request and decides whether to grant it. The decision is based on the subject issuing the request, the operation in the request, and an access rule that controls which subjects may perform that operation on the object. As the

Access matrix lists subjects and objects in a matrix

matrix is usually sparse, it is either implemented as access control list or capability list.

2.5.5 Access Control List

ACLs list authorized subjects for each object

Access control lists (ACLs) are columns of the access matrix described above. They provide means of determining the appropriate access rights to a given object depending on certain aspects of the subject that is making the request, principally the subject's user identity. The list is a data structure containing entries that specify individual subject or group rights to specific objects. Each accessible object contains an identifier or reference to its ACL. Figure 2.4 shows an example of an access control list. The privileges or permissions determine specific access rights, such as

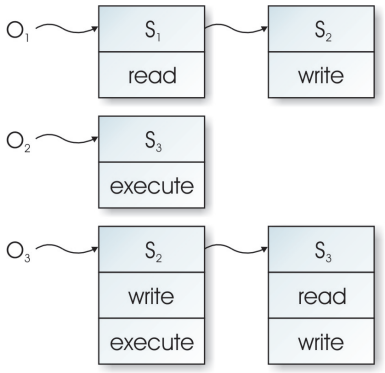

Figure 2.4: Access control list for the objects of Figure 2.3.

whether a user can read from, write to or execute an object. Some implementations also have permissions for modifying the ACL itself. ACLs are normally used for discretionary access control (DAC) described in Section 2.6.1.

2.5.6 Capability List

A *capability list* is a row of the access matrix. It is essentially a pair consisting of an object and an operation as Figure 2.5 illustrates. Capabilities must be protected from unauthorized modifications. In a centralized system, the operation system manages capabilities to protect the address space. A capability is typically implemented as a privileged data structure stored by the operating system in a list, with some mechanism in place to prevent the program from directly modifying the contents of the capability. Capabilities improve system security when used in place of plain references. In a pure capability-based system, the mere fact that a user program possesses that capability entitles it to use the referenced object in accordance with the rights that are specified by that capability. In theory, a pure capability-based system removes the need for any access control list or similar mechanism by giving all entities all and only the capabilities they will actually need. In a distributed system, the capability list has to be protected by using cryptography (e.g. signatures) or a reference monitor checking tickets on access.

Capabilities specify all accessible objects for each subject

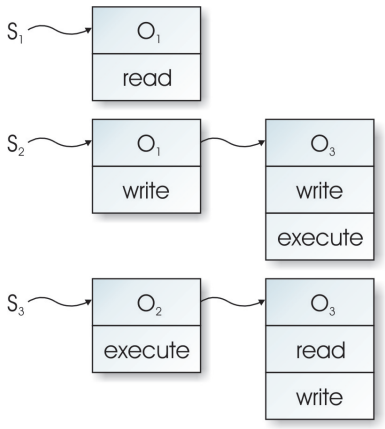

Figure 2.5: Capability lists for the subjects of Figure 2.3.

2.5.7 Comparing Access Control Lists to Capability Lists

Access control lists have some significant advantages over capability lists. On one hand, they are compact and easy to review, allowing deletion of an object in a simple way. Deletion of a subject is much more difficult since it requires the traversal of all ACLs. Delegation of permissions is possible by the owner having the (usually sole) authority to grant or revoke rights to the objects he owns or to other subject.

ACL: deletion of subjects is difficult

Capability lists are as compact as ACLs, yet they are not so compatible with an object-oriented view of the world. While delegation is easy, revocation of permissions can be difficult, especially in a distributed setting. In general, it is difficult to determine who has permissions on what objects since it requires the traversal of all capability lists.

Capabilities: revocation of permissions is difficult

2.6 Access Control Techniques

Access control techniques can generally be categorized as either discretionary or mandatory. In a discretionary system, the owner can change permissions and privileges on his own. A mandatory system enforces an access policy based on sensitivity levels. Role-based access control allows more efficient maintenance of permissions while lattice-based access control makes use of mathematical structures and security levels.

2.6.1 Discretionary Access Control

Discretionary access control (DAC) is an access policy determined by the owner of a resource. Every object in a system must have an owner. The owner decides who is allowed to access the resource and what privileges they have. The owner may

DAC: owner controls access

change the object's permissions at his[10] discretion and transfer ownership to other users. Access rights and permissions can be assigned by an owner to individual users or groups for specific resources. Discretionary access control can be applied through access control lists which name the specific rights and permissions that are assigned to a subject for a given object. Another possibility is role-based access control (RBAC, see Section 2.6.3) that assigns group membership based on organizational or functional roles. RBAC greatly simplifies the management of access rights and permissions.

2.6.2 Mandatory Access Control

MAC: system controls access

Mandatory access control (MAC)[11] is an access policy determined by the system, not the owner. MAC is used in multilevel systems that process highly sensitive data, such as classified government and military information. A multilevel system is a single computer system that handles multiple classification levels between subjects and objects. In a MAC-based system, all subjects and objects must have labels assigned to them. A subject's sensitivity label specifies a security clearance. An object's sensitivity label indicates the susceptibility or sensitivity of objects. In order to access a given object, the subject must have a sensitivity level equal to or higher than the requested object.

2.6.3 Role-Based Access Control

Indirection through roles allows better maintenance

Role-Based Access Control (RBAC) is a newer and alternative approach to Mandatory Access Control (MAC) and Discretionary Access Control (DAC). Entities can play different roles having different privileges. Since entities are not assigned permissions directly, but only acquire them through their role (or roles), management of individual entity rights becomes a matter of simply assigning the appropriate roles to the entity. This simplifies common operations such as adding an entity, or changing an entity's department. As Section 2.8.1 will explain, roles and groups are conceptually equivalent.

2.6.4 Lattice-Based Access Control

Lattices allow computations of upper and lower bounds

A lattice [38] $(L, \leq)$ consists of a set of security levels L and a partial ordering $\leq$, so that for every two elements $a, b \in L$ there exists a least upper bound $u \in L$ and a greatest lower bound $l \in L$. A lattice thus can uniquely answer questions like

- Given two objects at different security levels, what is the minimal level a subject must have to be allowed to read both objects?
- Given two subjects at different security levels, what is the maximal level an object can have so it can still be read by both subjects?

[10] Instead of writing 'his/her' throughout the text the term 'owner' is assumed to be gender neutral.

[11] Not to be confused with message authentication codes from Section 2.3.4.

Well known lattice-based models include Bell-LaPadula [32], Biba [33] or the Chinese Wall model [37].

2.7 Limitations of Access Control

Despite all described mechanisms, the potential of access control is quite limited. Access control, also in traditional centralized systems, is always based on trust and assumptions. There is no way, neither cryptographic nor organizational, to prevent improper operations without reasonable requirements to certain components.

Trust is inevitable

For example, an operating system performs access control of objects in the file system. Whenever there is an operation on an object, the corresponding system call executes code of the operating system kernel which acts as reference monitor or guard. The operating system ensures that the entity requesting access to an object is authorized. However, this implies trust in the operating system. One assumes that the operating system works properly and does not allow improper access to resources. Still, the correctness of access control by the operating system is not guaranteed, as

a) There could be implementation errors or a faulty design.

b) The operating system code could have been modified by the operator of the physical device (which is even simpler if the operating system is open source, but also possible for others).

c) The operating system code could have been modified by malicious software such as a virus or a worm.

d) The system could not be configured properly.

This example illustrates that even traditional systems are based on trust and assumptions. In a distributed system without centralized authorities, the situation is even more complicated as there is no physical control over the nodes that store objects. Further, knowledge about the physical whereabouts of objects is often not possible or even not desired. Cryptographic techniques only allow to *detect* improper modifications of objects, but they cannot *prevent* them. Nevertheless, the probability of data loss can drastically be reduced if objects are highly replicated and access decisions are based on a mutual consensus protocol. There is always a tradeoff between the level of availability and security versus efficiency.

Cryptography: only detection of modifications

Another problem that cannot be prevented is information dissemination. A reader having retrieved an object can distribute that object arbitrarily. Similarly, an entity knowing a private or secret key can leak that key to another entity. Under some circumstances, information dissemination will not even be detected. Recent research tries to find techniques such as trusted hardware and digital rights management (DRM) to prohibit information dissemination.

2.8 Groups

Definition *group*

We define a *group* as a number of entities with same rights and permissions with respect to accession, modification, and execution of objects. It allows modifying access rights of many entities by applying changes to the group. Groups have an owner who can change the permissions of a group and add or remove entities. Every object has a group assigned and every group can have zero or more objects. The owner also decides what objects are associated with which groups.

2.8.1 Duality of Groups and Roles

Groups ≡ roles

A group is an abstract unit consisting of a set of entities with certain privileges on objects. In principle, groups and roles are a dual concept: being member of a group implies playing the role of that group. Duality is also maintained in the sense that it is possible to be member of several groups as it is possible to play several roles. Moreover, joining a group is equivalent to obtaining a new role whereas leaving a group is equivalent to abandoning a role. Therefore, group management and security are related to discretionary access control (see Section 2.6.1).

2.8.2 Changing Group Membership

Requirements to groups are that when a group is joined, the new member receives all permissions of the group immediately. Conversely, if a member leaves a group, it must also lose all the privileges of the group. Furthermore, it must be feasible to determine whether an entity is member of a group or not.

2.8.3 Changing Group Permissions

Whenever permissions change, it must apply to all group members. However, it is questionable whether permissions need to be checked on each access to an object. As an example, one could think of the following scenario: An entity has opened a file with write permissions. Assuming that the group to which that entity belongs to loses the permissions to write to the particular file. It would then be improper to allow to the entity any further write operations. Therefore, it would be necessary to check the privileges on objects on every single access which results in an overhead. Hence, one often assumes that permissions do not change very often which is reasonable and sufficient for most systems.

2.8.4 Groups in Distributed Systems

In principle, all requirements and properties of groups remain unchanged from what has been outlined in the previous sections. The difference is that the information about the members of the group and the permissions are stored in a distributed

manner. Due to the distribution, some information might not always be available[12]. Hence, one could weaken the requirements on groups which insist on the same privileges for all group members at all times.

Weaker model for groups in distributed systems

For changing group membership, a weaker condition is that it must only be feasible to determine whether an entity is *not* member of a group. This prevents non-group members from accessing resources, but potentially also proper group members. Hence the propagation for new group members can be delayed while the deletion of group members must be propagated immediately to prevent improper access.

The same holds for changing group permissions. For permissions being added to a group, it is not absolutely necessary to immediately propagate the new permissions in the system. This can result in a situation where some members of a group cannot yet make use of the new permissions although others already can, due to delayed propagation.

On the other hand, removal of a permission from a group must result in an immediate removal of that permission for all group members. Otherwise, it would be possible that some members still have access permissions to an object although the group is no longer authorized to access that particular object. Note that there is an asymmetry for the requirements of adding and removing permissions.

Distributed systems also raise questions like: Where is the group information stored? Who can modify membership and permissions? How can improper modifications be prohibited? How long does it take to make changes effective?

2.8.5 Groups in Peer-to-Peer Systems

To make the information about members of a group permanently available, group members need to be stored within an object or file in the P2P system because there is no central authority. This object can only be written by the owner, but must be world readable such that verification of membership can be performed. In order to identify authorized users properly, the group object contains the users public keys that are used for authentication.

Only the owner of a group can add and remove members. He can either invite other entities or receive requests for joining a group. This implies that entities willing to become member of a group must wait until the owner adds them to the group. There are two principles how group membership can be achieved. Either, the owner decides on his own who is added to the group and then informs the corresponding entities by some means. Alternatively, entities can send a request to the owner which is then processed. In the latter case, those requests must also be stored in the system. For that purpose, one needs an additional file to which

[12] For example, an access control list could be stored on various hosts. If those hosts are not online, the system performing access control must handle this problem in some appropriate way.

requests can be appended to. The crucial point is how this file can be found in the system. One solution could be that groups and their request-files are made public by some means, for example by putting them on a web page or by publishing them into a well-known directory.

It is essential that the owner is the only one that can delete a group or change the permissions of a group. If several entities want to form a group to share their files, they first have to group themselves (choosing a group leader as the owner of a group). This group can then be used as the owner of the group that should be created such that all members of the group have owner permissions on the newly created one. If the group leader decides to leave the group, he needs to transfer his ownership to one of the other members or, if he is the last member, delete the group.

This chapter gave a short overview of the most important cryptographic schemes including access control. The last section on groups was not a cryptographic topic, but is nevertheless important for the understanding of the subsequent chapters. The next chapter gives an insight into the efforts made in the area of P2P storage and P2P security.

3 Related Work

In this chapter, we briefly discuss related work that has been carried out in the area of distributed object stores and explain how security and especially access control is handled. All practical peer-to-peer file systems must cope with security and access control. We will focus on security-related aspects and disregard the mechanisms used to store and retrieve objects.

3.1 OceanStore

OceanStore [42, 43] is a distributed, global-scale, secure, fault-tolerant system that provides persistent storage with two design goals: The ability to be constructed from an untrusted infrastructure and the support of nomadic data. All information that enters the infrastructure must be encrypted to achieve secrecy. Rather than assuming that servers are passive repositories of information, it is desired that servers participate in protocols for distributed consistency management. Therefore, most of the servers are working correctly most of the time and there is a class of servers that can be trusted to carry out protocols on their behalf. This responsible party is stated to be financially accountable for the integrity of the data.

OceanStore concerns with read and write access control. To perform access control, the owner of an object chooses an access control list for each object. An ACL entry extending privileges must describe the privilege granted and the private key of the privileged users. The information is made publicly readable such that servers can check whether a write is allowed.

Restricting read access is entirely a matter of restricting access to a decryption key. Only those users with read permission are in possession of the decryption key. To revoke read permission, the owner must request that replicas be deleted or re-signed with the new key. A recently revoked reader will be able to read old data from cached copies or from misbehaving servers which fail to delete or re-key.

Write access is restricted by digital signatures such that honest servers and clients can verify them against an access control list. Since decisions to commit data are performed by a quorum of servers called "inner ring", one can trust that only valid writing operations are accepted. Nodes must be able to perform commitment without access to cleartext or encryption keys. To guarantee authenticity, the GUIDs of the objects are self-verifying as they are a secure hash over the data.

The mechanisms for access control in OceanStore are not described in detail. Revocation of read permission is realized by re-encrypting or deleting all replicas. That approach is not only expensive, it is also insecure (see Section 5.9). The precise steps of a write operation and the assumptions of the servers which are responsible for executing the operation on behalf of the owner are not described. Although an ACL is chosen for each object, it is not explained how injection of out-of-date objects can be detected. Inner ring nodes that are required to be mostly available and reliable - an assumption that in general cannot be made in a P2P environment.

3.2 Farsite

Farsite [44] is a secure, scalable file system that logically functions as a centralized file server but is physically distributed among a set of untrusted computers. It ensures the secrecy of file contents with cryptographic techniques and maintains integrity of file and directory data with a Byzantine-fault-tolerant protocol. The majority of machines is assumed to be up and accessible for the majority of the time. Since a file system is a hierarchical directory namespace, it must have a root. An administrator creates a namespace root by choosing a unique root name and designating a set of machines to manage the root. These machines will form a Byzantine-fault-tolerant group. There are certificates for namespaces, users and machines. Certificates have to be validated with the public keys of certification authorities. A machine's private key is stored on the machine itself. A user private key is encrypted with a symmetric key derived from the user's password and then stored in a globally-readable directory. Revocation of certificates are accomplished by issuing signed revocation lists. Certificates might also expire. To protect user privacy and provide read-access control, clients encrypt written file data with the public keys of all authorized readers. The directory group enforces write-access control by cryptographically validating the requests from users before accepting updates.

Because directory groups only modify their shared state via a Byzantine-fault-tolerant protocol, one trusts the directory group not to make an incorrect update to directory metadata. That metadata includes an access control list of public keys of all users who are authorized writers to that directory and to files therein. When a client creates a new file, a symmetric file key is randomly generated. Then the client computes a one-way hash of each block of the file. This hash is used as a key to encrypt the block. The file key is used to encrypt the hashes rather than to encrypt the file blocks directly. This technique is called *convergent encryption*, because identical file plaintext converges to identical ciphertext, irrespective of the user keys. Integrity of a file is ensured by computing a Merkle hash tree [45] over the file data blocks, storing a copy of the tree with the file, and keeping a copy of the root hash in the directory group that manages the file's metadata. To perform committed updates, a client generates a random authenticator key and splits it into secret shares, which it distributes among members of the directory group. With this key, the clients signs each committed update using a message authentication code (MAC) [47]. The directory group jointly reconstructs the authenticator key and does not accept any further updates signed with that key.

Farsite's scalability is limited by two points of centralization: Certification authorities and root directory groups. The assumption that the majority of machines is up and available is quite demanding for a P2P network. Since the file key needs to be encrypted with the public key of each authorized reader, the approach does not scale for large groups of readers. The question is how one can prevent read access to a file to a former reader, if the keys are stored together with the file. The precise steps of the directory group for write access are not outlined, except for the fact that a Byzantine agreement protocol needs to be executed.

3.3 Ivy

Ivy [48] is a multi-user read/write peer-to-peer file system. Ivy has no centralized or dedicated components, and it provides useful integrity properties without requiring users to fully trust either the underlying peer-to-peer storage system or the other users of the file system. All data is stored in a DHash [49]. DHash ensures the integrity of each block with one of two methods. A *content-hash block* requires the block's key to be the SHA-1 [50] cryptographic hash of the block's value. A *public-key block* requires the block's key to be a public key, and the value to be signed using the corresponding private key.

Besides the integrity of a block, Ivy does not make any statements about access control. The authors envision that a participant's bad behavior is discovered after the fact. However, the mechanisms that need to be in place to make such detections possible are not described. Similarly, it is left open how a group of readers can be managed efficiently.

3.4 SiRiUS

SiRiUS [51] is a secure file system designed to be layered over insecure network systems. It aims at improving security without making any changes to the file system or network server. All access control information is stored, encrypted and signed together with the file data. SiRiUS supports two file access modes: read-only and read-write. The system is optimized to support sharing only for small groups. Ideally, SiRiUS can use an existing key distribution infrastructure, such as PGP. When a read or write access to a file is revoked, the revoked user should immediately lose access to that file without need for communication. To prevent access control rollback and to ensure that users always have the latest version of their files, SiRiUS must guarantee freshness.

All SiRiUS users maintain one key for asymmetric encryption (MEK)[13] and another for signatures (MSK)[14]. The data of a file is encrypted with a symmetric encryption key (FEK)[15] and signed with a signature key (FSK)[16]. Both keys are unique for each file. Possession of the FEK gives read-only access to the file while possession of both, the FEK and FSK, allows read and write access. Freshness is guaranteed through a hash tree [46]. The hash of the root directory additionally contains a timestamp and is updated periodically by the owner. If the content of a file or directory changes, the tree needs to be updated up to the root.

SiRiUS has several serious drawbacks. Although freshness seems to be guaranteed, it is easy to see that the single point of failure is the root. As not stated explicitly by the paper, it is unclear whether the root information is only stored by the owner or whether it is persistently and publicly stored in the file system. In the first case, authenticity of meta data can only be verified if the owner is available, an assumption which in general does not hold for P2P systems. In the second case, it is impossible to guarantee freshness of the root itself. One could argue that freshness can be verified by the timestamp, but then the question arises how other entities know about the most recent timestamp. Of course, if the timestamp is updated periodically, one can compute the correct value of a time stamp at a certain point in time. However, if the owner itself is allowed to be offline, a trusted entity needs to execute timestamping. Further, freshness is only guaranteed with respect to metadata, but not with respect to the data blocks itself. Another problem is the design criterion which only foresees a small group of entities per file. Moreover, whenever a reader leaves the group, all files of the group need to be re-encrypted, which is inefficient for large files or a large number of files per within a group.

[13] Master encryption key.
[14] Master signing key.
[15] File encryption key.
[16] File signature key.

3.5 PAST

PAST [62, 63] is a large-scale, Internet-based, global storage utility which is entirely self-organizing. Nodes are not trusted, they may join the system and may silently leave the system without warning. PAST assumes that most of the nodes in the overlay network are well behaved. Each PAST node and each user of the system hold a smart card. A private/public key pair is associated with each card. For each inserted file, a *file certificate* is created which basically contains a hash of the file's content and is signed by the smartcard. Privacy is ensured by use of a cryptosystem of the own choice while integrity is guaranteed by file certificates.

The use of smartcards makes key management fairly simple. There is no need to store private keys on the local machine. As the paper mentions, one must nevertheless be careful when dealing with smartcards. First of all, smartcards can be compromised by resource-rich attackers. Further, they need to be replaced from time to time since the certificates on the smartcard expire. And finally, many users do not have smartcard readers for their desktop computers. Of course, it is legitimate to assume future systems with support the use of smartcards widely. PAST does not make any statements about how files can be shared and permissions on files can be granted or revoked. Hence, there are no statements about access control and access control mechanisms. Further, freshness of objects has not been considered.

3.6 Kangoo

Kangoo [64] is a distributed file system that operates on untrusted storage. The core concept for security and access control is based on a data structure called *Cryptree* [65]. The basic idea is to leverage the file system's folder tree structure for key management. The meta data consists of a set of keys, each used in different situations. Cryptree follows the principle of lazy revocation, which means that a former reader can still access files by keeping copies of the encryption keys until the file has been written with a new key. A so-called clearance key allows to access some part of the Cryptree, namely the file - or if the key refers to a folder all subsequent sub-directories and files by only revealing one key. Cryptographic links graphically represent the encryption keys. In principle, authorized readers are in possession of a decryption key while proper writers hold a signing key. Those keys change if readers or writers lose their permissions on a file or folder.

The fact that Kangoo uses lazy revocation at the cost of security is acceptable if one considers the costs for re-encryption of all involved objects. In case that a reader or writer loses its privileges, some of the keys need to be refreshed and updated. On such an event, the new keys have to be distributed to all remaining entities. Kangoo maintains an encrypted list for each user of the system which contains the access keys to the Cryptree. Those lists need to be updated on a reader or writer removal. In other words, if 1 out of n authorized entities is removed, $n-1$ lists need to be updated with the new key and symmetrically encrypted. For large groups of

readers or writers such a mechanism does not scale. The concept of groups is not yet fully developed [66]. Moreover, although file storage and retrieval is based on P2P mechanisms, Kangoo is not a fully decentralized application: There is a central authority which performs access control on write requests. In Kangoo, the keys of Cryptree belong to the meta data part of the file or folder. Readers must request the latest identifier of a file or directory from the central server. Conversely, a writer encrypts the payload of a file with a fresh secret key and contacts the central server with the signed identifier of the file. The designated server only accepts the identifier if it is properly signed. Replays of old write operations are not possible since they contain a version stamp. The injection of old file versions on a read attempt can also be prohibited by the designated authority since it always knows the latest version. The Cryptree as a data structure could also be used in a decentralized system. Although the concept of the Cryptree is artful, the designated server simplifies hard problems of a real distributed reference monitor significantly. This thesis seeks for means to eliminate the need for a central server.

3.7 Celeste

Celeste [61] is a self-managing, secure, massively distributed, random-access, read/write data store with high availability properties. It maintains a list of versions of an object where any version can be read, and the latest version can be written. Reliability in Celeste is achieved through component redundancy, data replication, and introspection processes. Broadly, security consists of methods to protect both, the Celeste infrastructure and the data stored in the Celeste. It intends to include security aspects like data encryption, authentication, key management, authentication, serialization as well as authenticated, encrypted communication between all components. Trust must be limited to the absolute minimum in an efficient distributed system. Data encryption and authentication through digital signatures provides a basis where only those components that hold the relevant keys need to be trusted. Since confidentiality is achieved by using encryption, an adequate group key management scheme is about to be developed.

The white paper is not specific about the concrete measures regarding access control. In particular, the group key management scheme is not described. Further, it is not clear how write access control is performed. Since this thesis is carried out in cooperation with Sun Microsystems Laboratories, the theoretic access control schemes described in Chapter 5 will be partially implemented as a proof-of-concept in Celeste. The implementation and performance analysis are then outlined in Chapter 6 and 7.

4 Distributed Object Store

To properly describe access control, the functionalities of the underlying object store and the representation of objects need to be addressed. First of all, the requirements for an object store need to be listed, followed by the operations that must be supported. Then, an abstract description of an object is given which disregards the details of a particular implementation. Further, it will be necessary to explain the different entities of the object store and their relationship. To preserve generality, the system and objects are described abstractly, masking the details that must be considered by an implementation.

Object store vs. file system

The notion of an object store is different from the notion of a file system. File systems are hierarchical: There are directories which can contain subdirectories and files. The file system hierarchy is equal to a tree where the root of the tree is the root of the file system. There are directory services which allow to navigate over the directories and to query and retrieve sub-directories or files. In contrast, an object store is flat. There is no notion of directories. It only consists of objects distinguishable by their identifier. An object store is more general than a file system.

4.1 Requirements and Functions of an Object Store

An object store must handle two tasks:

1. Persistent storage of data and tolerance for a certain amount of node failures. Persistent storage
2. Efficient location and retrieval of data. Efficient retrieval

Practical systems must fulfill both requirements in order to be reliable and efficient.

GUID

To identify objects unambiguously, each stored object must be assigned a globally unique identifier (GUID). The generation thereof is left open to a concrete implementation. Depending on the generation scheme, there are different implications regarding security:

Self-verifying: The GUID is a hash of the object that can be used to verify the authenticity of an object. An explicit authenticator is thus not required.

Verifiable: The GUID is not derived from the object, but there is a strong binding between the identifier and the object, for example through a digital signature.

Derived: The GUID is derived by applying a function to some input values. To restrict the range, some parameters can be fix, for example the user's public key. It prohibits the user from choosing identifiers arbitrarily.

Freely selectable: The GUID is chosen arbitrarily.

For the last three schemes, the content of an object needs to be protected by an explicit authenticator. To guarantee a one-to-one relationship between a GUID and an object, the object needs to contain the identifier. Otherwise, it is be impossible to verify that a GUID matches the retrieved object. Without loss of generality and to keep the complexity of the presented schemes as low as possible, this thesis assumes that the object store uses self-verifying GUIDs[17]. As it is assumed that identifiers can be uniquely generated, the abbreviation ID is used.

Object store interface

The object store must support the following interface:

- Retrieval of an object Obj given its identifier ID_{Obj}
- Storage of an object Obj with a certain identifier ID_{Obj}

If one tries to retrieve a nonexistent object, the system will return a $NULL$ value. Notice that deletion of an object is a desirable, but not a mandatory functionality. One can argue that objects never have to be deleted as there is always sufficient storage space. The precise techniques and mechanisms that lie behind an object store system are subject of ongoing and past studies, some of them are mentioned in Chapter 3. From an abstract point of view, it is not necessary to know how objects are retrieved concretely. It is sufficient to assume the existence of such mechanisms without considering the underlying details.

4.2 Network Architecture and Communication Channels

The object store is built upon an unreliable network. The topology of the network is not known and can change at any time. Appropriate routing algorithms and

[17] Except for two special types of objects, as it will show in Section 4.5

mechanisms are assumed to be in place to handle dynamic networks. The communication channels are asynchronous. There are no guarantees regarding how long it takes for a message to arrive at some node. Messages can be lost or changed due to channel noise. An adversary can eavesdrop a channel or manipulate messages on the channel. Thus authenticity has to be guaranteed by digital signatures or MACs, confidentiality by encryption.

4.3 Object Representation and Storage

The representation of objects within a distributed object store can be manifold. To allow reasoning about access control, one needs at least a rough idea of how objects are organized and structured.

An object Obj can be stored as a single block within the object store. This approach is easy and straightforward to implement. Nevertheless, it can be more efficient to perform operations on smaller parts and by decomposing an object into v blocks[18]:

$$Obj \equiv \{obj_1, ..., obj_v\}.$$

Object blocks

To maintain confidentiality, each block is encrypted using a symmetric key SK_i[19]:

$$Obj_{enc} \equiv \left\{\{obj_1\}_{SK_1}, ..., \{obj_v\}_{SK_v}\right\}.$$

For brevity, an encrypted block $\{obj_i\}_{SK_i}$ will be written as obj_i^{enc} to indicate that it is a cipher:

$$Obj_{enc} \equiv \{obj_1^{enc}, ..., obj_v^{enc}\}.$$

One way of managing the keys is that each involved entity stores them locally or on an external storage device. Since local devices can be compromised or destroyed, it would require that keys are protected carefully. Most users do not want to care about their keys and tend to handle them incautiously. For a large number of objects, key management can be cumbersome. Whenever a block is encrypted with a new key, the key needs to be transmitted to all involved entities even if some of them are offline. For these reasons, it is best to store the keys SK_i themselves in the object store by suitable means. The presented approaches for access control will use different mechanisms to protect the secret keys such that only authorized readers can retrieve them.

Figure 4.1 illustrates the object layout schematically. For each encrypted block obj_i^{enc}, the self-verifying identifier is computed as $ID_{obj_i} = h\left(obj_i^{enc}\right)$, where h is a cryptographic hash function. The contents and the number of blocks can change over time due to write operations[20]. It is hence practical to enclose all block identifiers in a separate header object H_{Obj} that may contain further meta data.

Header object

[18] In many cases, read or write operations only affect a small part of an object. Example scenarios include object streaming or write operations that append blocks.

[19] Note that the secret keys SK_i do not necessarily need to be distinct.

[20] For example, a write operation can override an existing block and thus require to generate a new identifier for that particular block.

For each encrypted block obj_i^{enc}, H_{Obj} contains one entry consisting of the block identifier ID_{obj_i} and information about the key that was used to encrypt the block. Additionally, the header contains a version counter vc_{Obj} which is incremented on each write operation to the object. The identifier $ID_{H_{Obj}}$ of a header object is self-verifying. For efficiency reasons, there is an *anchor* object $Anchor_{Obj}$, which consists of $ID_{H_{Obj}}$. It will be shown that $Anchor_{Obj}$ needs to be treated differently. The identifier ID_{Obj} that is associated with the anchor does not change over time and can be interpreted as the "name" of an object. Read and write requests will both use ID_{Obj} to access an object. The object layout can be compared to the UNIX file system: H_{Obj} equals the concept of an *inode*, the blocks obj_i equal blocks in the file system referenced by the inode, and $Anchor_{Obj}$ is a reference to the inode similar to an entry in a directory.

Anchor object

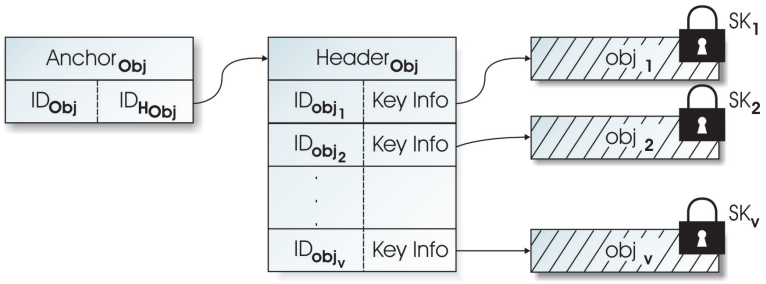

Figure 4.1: General representation of objects in the object store.

4.4 System Entities

Node

The basic unit constituting the object store is called *node*. A node is one instance of the object store program. A physical device running n object store applications thus represents n nodes. As mentioned in Section 4.2, there are neither guarantees about how nodes are connected in the network nor how the network topology changes over time.

Entity

The notion of an *entity* is frequently used as a general term for the distinct roles in the object store system such as owner, reader, writer or gatekeeper. In literature, an entity can also stand for a process or a host.

Owner

Access control is always performed on behalf of an entity which is in possession of a set of objects. That entity will be called the *owner* of an object. The owner can select entities $R = \{r_1, ..., r_m\}$ having the privilege to read an object. Similarly, he can grant entities $W = \{w_1, ..., w_k\}$ permission to write to an object.

Reader / Writer

An authorized *reader* will be abbreviated with r_i while w_j stands for a regular *writer*.

The owner transfers his privileges to perform access control to a set of represen-

tatives that act on behalf of the owner. This is necessary in case the owner is offline. These representatives are referred to as *gatekeepers* $G = \{g_1, ..., g_n\}$. Each gatekeeper has a state and performs some operations upon request from another entity. Only entities that are properly authorized by gatekeepers are allowed to access objects.

Gatekeeper

Nodes which store objects or blocks will be referred to as *storage nodes*.

Storage node

A node can play several roles at the same time. For instance, a node running a gatekeeper can both store blocks of a particular object and act as authorized writer simultaneously. However, there are different assumptions and requirements that must hold for distinct types of entities, as will be explained in Section 4.7.

4.5 Retrieval and Storage of Objects

Although it is beyond the scope of this thesis to address algorithms and data structures for retrieving and storing objects, it is nevertheless inevitable to discuss certain aspects and assumptions thereof.

There are only two types of objects which have a non-self-verifying identifier. One is the *anchor object* which was already introduced in Section 4.3. They are not stored on storage nodes, but handled by gatekeepers and need to be treated specially. On the other hand, there is a so-called *witness object* (cf. Section 5.10.1), which has an identifier that is non-self-verifying but verifiable. For the remaining objects, the identifier is a secure hash of the object. The goal is to stick to self-verifying objects whenever possible since the computation and verification of self-verifying identifiers is cheap[21]. First, the general case of self-verifying identifiers is considered before explaining the treatment of other identifier types.

4.5.1 Self-Verifying Objects

When a store operation is about to be performed, a request of the following form is issued:

Storing self-verifying objects

$$[ID, Obj].$$

The object store selects storage nodes to hold the object persistently. A storage node accepts the request if and only if the identifier is self-verifying by ensuring that $ID_{Obj} \equiv h(Obj)$. Similarly, if an object with identifier ID_{Obj} is read, the system verifies that the object matches its identifier. If the secure hash of the object's content retrieved from the storage node does not match the proclaimed identifier, it is likely that the storage node has maliciously altered the object's content or changed the identifier. With appropriate mechanisms in place, the corrupt storage node can be banned from the system.

[21] The verification can be done by applying a secure hash function. In contrast, an identifier for a witness object additionally requires the verification of a digital signature.

4.5.2 Non-Self-Verifying Objects

Storing non-self-verifying objects

Despite the fact that non-self-verifying identifiers are more expensive to compute, they have some advantages over normal identifiers. On the one hand, the identifier can be persistent, which means that it does not change over time. On the other hand, the identifier can be independent of the content of the associated object. ID_{Obj} of the anchor is such an example. It can be chosen arbitrarily or it can be the result of a function, for example a hash of a human-readable filename and the owner's public key. The witness object in Section 5.2.4 is another example for a special type of non-self-verifying identifiers called *secure version identifier* (SVID).

4.6 Secure Version Identifier

Self-verifying identifiers have the nice property that the ID allows to verify the authenticity of the object's content. In general, non-self-verifying identifiers do not have this property. This section presents a novel and secure way how a non-self-verifying identifier

1. Can guarantee the authenticity of an object and
2. Allows to determine the next version of the object by local computations.

Secure version identifier

This type of identifier is called *secure version identifier* (SVID). An object that is associated with a SVID must contain certain information explicitly:

- A public key PK of an entity.
- A version counter vc.
- An identifier or name ID_{Name}.

Unlike self-verifying objects, an object with an SVID needs to be signed with the corresponding private key PK^{-1}:

$$Obj^{sig} = \{h\,(Obj)\}_{PK^{-1}}$$

The identifier of the object can be computed as a function of PK, vc and ID_{Name}:

$$ID_{Obj} = h\,(PK, vc, ID_{Name})$$

A storage request then consists of the triple:

$$\left[ID_{Obj}, Obj, Obj^{sig}\right]$$

If the type of the object to be stored is declared as an object using SVID, a storage node must perform some verifications before accepting the object:

- Retrieval of the first three fields in the object which are PK, vc and ID_{Name}.

- Verifying that $ID_{Obj} = h\,(PK, vc, ID_{Name})$.
- Verification of the signature on the authenticator Obj^{sig} by applying PK.

If one of the verification steps fails, the store request is rejected. Verification on a read request is analogous. The generation of ID_{Obj} allows to verify that the object has been signed with the same key that was used to generate the identifier. The version counter allows to distinguish between objects with different versions while ID_{Name} separates distinct objects. An attacker is not able to forge an object since he does not know the correct private key PK^{-1}. Using a different private key for either generating the object's identifier or for signing the object will be recognized by storage nodes. If a malicious storage node accepts a malformed store request, the fraud will be detected when the object is retrieved later. SVID achieves the desirable property of linking an object unambiguously to an identifier without knowing the content of the object in advance.

Unambiguous link without knowing object content

Besides authenticity, SVID also allows to determine other versions of an object using local computations. The version counter vc that is contained in the object and used for the generation of the identifier can be incremented to determine the next object version:

1. Compute $ID'_{Obj} = h\,(PK, vc + 1, ID_{Name})$.
2. Retrieve the object with ID'_{Obj}.
3. If no object with ID'_{Obj} was found, then the object with identifier ID_{Obj} is the latest version. Otherwise, perform all verifications mentioned. Set $vc := vc+1$ and $ID_{Obj} := ID'_{Obj}$. Goto step 1.

Depending on the number of changes on the object, one can replace the linear identifier traversal by an alternative search method, such as exponential search where the version counter is set to $vc := vc \times vc$.

Linear vs. exponential search

Having a good understanding of the most important functionalities of an object store, it is now necessary to discuss the abilities of an adversary. Since there are different types of entities, it seems natural to think about the number of entities of each category that can be compromised by the adversary without having an impact on the security of the system.

4.7 Adversary Capabilities

An adversary can access objects on local nodes, read the contents (unless they are encrypted) and modify them arbitrarily. He can also inject or manipulate any messages in a protocol exchange. He can further change the code of the object store that is running on the nodes he controls, for example by disabling the sections dealing with access control. Therefore, nodes controlled by an adversary can behave

arbitrarily malicious.

Computationally-secure cryptography cannot be broken

However, the capabilities of an adversary need to be limited. First, it is assumed that the adversary cannot break computationally-secure cryptographic schemes such as RSA or 3DES. He can neither decrypt nor sign an object without knowledge of the secret or private key. Otherwise, authenticity and secrecy can be broken. Second, the overall number of malicious nodes needs to be restricted for each type of entity.

Assumption 4.7.1 *From the set of storage nodes holding a replica of an object, at least one storage node must be honest and available. This is, at least one storage node does perform object storage properly and does not override or manipulate the stored object maliciously.*

Although authenticity of objects on storage nodes is guaranteed by the characteristics of the identifiers, an adversary controlling all storage nodes can simply override all objects with arbitrary data or even delete the objects. A reader would be able to observe this manipulation, however the system is unable to recover the correct version which would degrade the availability of the system.

Assumption 4.7.2 *From a set of n gatekeeper nodes, at most t are allowed to be malicious at the time, where t is the threshold of a $(t+1, n)$-threshold secret sharing scheme [18]. Conversely, at least $t+1$ gatekeepers must be honest and available.*

This assumption trivially emerges from the fact that a $(t+1, n)$-threshold secret sharing scheme is only secure against t malicious participants. The requirement implies that $t < n/2$.

Assumption 4.7.3 *The number of malicious gatekeepers t is bounded by $t < n/4$, where n is the total number of gatekeepers that is assumed to be available all the time.*

A byzantine consensus protocol [41, 53] requires that $t < n/3$. As a consensus protocol is chosen that allows an arbitrary input domain (cf. appendix A), the requirement needs to be tightened to $n < n/4$. Since assumption 4.7.3 is stronger than assumption 4.7.2, one can state that $t < n/4$ must always hold.

Assumption 4.7.4 *As the owner is interested in securing his objects properly, the owner is assumed to be honest and not compromised with respect to operations involving his objects.*

However, an owner is allowed to behave malicious regarding operations on objects that do not belong to himself. If any of those assumptions is violated, access control can be broken and data can be manipulated or deleted. Before discussing distributed access control schemes, one has to bear in mind the general limitations of access control as described in Section 2.7.

5 Distributed Access Control

This chapter analyzes access control mechanisms in a distributed environment on the basis of an object store. The first section gives a short high level overview of the system. Then, gatekeepers and read/write access control is discussed. Intermediary sections highlight other important aspects of the system.

5.1 Introduction

The fundamental problem of a distributed object store is that there is no central authority that can be consulted to decide on an access attempt. The object store is in general untrusted and unreliable. The unpredictable behavior of peers requires mechanisms which are flexible, but retain security at reasonable costs. This chapter aims at examining, analyzing and evaluating different schemes for access control and compare them to each other. For all schemes, the authority which decides on an access request is distributed. A request must be approved by all honest participants of the authority. Such a scheme allows a certain number of participants to be arbitrarily corrupted without having any impact on the correctness of the protocols. 'Corrupt' means that the participants may inject faulty messages or values during protocol exchanges or simply refuse to collaborate. This implies that the schemes can tolerate a certain number of node failures which is relevant in a P2P setting.

5.1.1 A Trivial Approach

So far, a schematic description of the representation of objects is given without considering security. In general, objects need to be protected such that they can be neither read nor written without proper authorization. A straightforward way to protect objects is to use encryption and to append an authenticator. For self-verifying objects there is no need for an explicit authenticator since authenticity is achieved by the identifier. Read protection is ensured by the encryption while modifications can be detected by the identifier. In case the object has been altered, other replicas are retrieved until the authenticator is valid.

Scalability for large groups

For each authorized reader of the group, the data is encrypted once with each reader's public key. When a reader wants to read an object, the appropriate data part is decrypted. The approach does not scale for large groups of readers because the number of encryptions is linear in the number of readers. One can also think of encrypting data with the same key for all readers. However, the problem is then to transfer the key securely to the readers even if they are offline. On the other side, it is necessary to detect improper write operations and to prevent them from being realized. Cryptography provides little support for such functionalities. It will be shown that it is a challenge to prevent an entity, which previously had write permissions, from replaying old (and at that time authorized) write operations.

Minimize number of cryptographic operations

The schemes aim at minimizing the number of cryptographic operations on group changes. The trivial approach implies that one can get by with a linear number of operations, where linearity refers to the number of entities of a group. For large groups, a linear number of cryptographic steps, even when using symmetric cryptography, is normally infeasible.

5.1.2 Conceptual Overview

Before discussing various schemes for access control in peer-to-peer storage systems, the conceptual design on which all described approaches are based on is outlined. Chapter 4 explained the general layout of objects and which entities are involved. Section 4.4 mentioned the roles that entities can play in the system. Gatekeepers are introduced as representatives of the owner. Yet it is unclear how they behave within the object store system. Section 5.2.1 states why gatekeepers are a necessary prerequisite.

Many systems provide access control at the level of objects. This means that an object has an access control list to which subjects can be added as Section 2.5.5 outlines. In traditional systems, the number of subjects within an ACL is usually quite small. However, in a distributed system, objects can be shared among many entities such that maintaining an ACL for each object is expensive regarding storage complexity. It is more efficient to perform access control at the level of groups of objects.

Foremost, there is an owner entity who creates objects and stores them to the

object store system. If those objects are intended to be shared among other entities, they need to be assigned to a group. Being member of a group allows to read or write to an object of the group, depending on the permissions granted by the owner[22]. For each group, the owner selects a set of gatekeepers to control access. Therefore, gatekeepers need to maintain a state to decide which access attempts are authorized and which are not.

Gatekeepers guard access to groups of objects

The owner is the only entity that has the competence to add or remove readers or writers to a group. Hence all approaches implement a discretionary access control (DAC) scheme. Gatekeepers serve as a distributed reference monitor guarding and authorizing all access attempts.

To identify writing entities, the owner maintains an access control list (ACL) containing all authorized writers of a group. Gatekeepers consult the ACL whenever a writer accesses an object. This implies that a write operation is always executed via the gatekeepers. A writer needs to issue write requests to identify himself towards the gatekeepers. Gatekeepers need to execute an agreement protocol to ensure that the write request is consistent for all honest gatekeepers. Figure 5.1 gives a rough idea how the entities interact with each other.

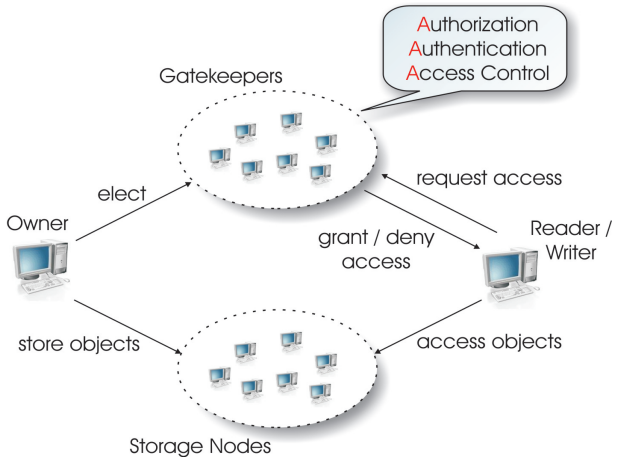

Figure 5.1: A high level view of the interaction between system entities: owner, reader, writer, gatekeepers and storage nodes.

The general idea of read permission is to allow readers to retrieve the decryption key of an object if and only if the reader is authorized. Three different schemes are presented which intend to minimize the number of cryptographic operations when performing a read operation or changing the membership of the readers of a group. It is essential that all keys are stored persistently in the object store to make them

[22]Based on those two basic permissions, other types of permissions can be derived such as append, delete, change of meta data and so forth.

available at all times. The three different schemes to manage the decryption keys are: List-based, tree-based or sharing-based. In the first two approaches, read access control is implicit by the fact that the decryption key is encrypted such that only authorized readers can access the key. In contrast, the last approach requires the gatekeepers to perform access control explicitly.

Implicit vs. explicit read access control

Recentness of objects

All schemes will make use of gatekeepers to guarantee recentness of retrieved objects. Readers should have the guarantee that the retrieved object is indeed the latest one (cf. Section 5.2.2). Since gatekeepers are involved in write operations, they also have the competence to identify the latest version of an object.

The structure of this chapter is as follows: First, it is described how gatekeepers are initialized and how they perform access control. Then, important group operations are discussed followed by write access control and the description of three different read access control schemes. Finally, it is explained how to cope with issues related to former readers[23] and self-organizing gatekeepers before a performance analysis is presented.

[23] A *former reader* is an entity that once had read permission on a group, but was then removed by the owner.

5.2 Gatekeepers

In the presented schemes, gatekeepers are the main component for access control in a peer-to-peer system. The owner transfers his privilege of performing access control on his objects to a set of gatekeepers. Access control can then be performed even if the owner is offline. Gatekeepers serve as a distributed reference monitor and decide whether access to an object is authorized or not. Multiple gatekeepers are chosen to tolerate faulty nodes, to minimize trust in a single peer and to be resilient against t Byzantine gatekeepers. To decide on the validity of a write access attempt, an agreement protocol needs to be executed. Depending on the outcome of the agreement protocol, the access is granted or rejected.

Gatekeepers are a distributed reference monitor

5.2.1 The Necessity of Gatekeepers

In Section 4.4, gatekeepers have been introduced as a system entity. Assumptions 4.7.2 and 4.7.3 state the requirements regarding the number of Byzantine gatekeepers. This sections explains why gatekeepers are a necessary requisite for performing distributed access control.

A straightforward scheme requires all operations to be authorized by the owner. The data to be written is sent to the owner who decides whether the writer is authorized and then performs a store operation to update the object. Depending on the scheme that is used, the owner can sign the new object to guarantee that he approved the operation. Reading entities can then verify the signature and decide whether the object was written legally or not. Furthermore, the owner has the sole competence to know which object version is the current one. In such a scheme, the owner serves as a centralized authority regarding his objects. Note that operations would be deferred in case that the owner is offline. This is very impractical since in general, one cannot assume the owner to be online all the time. The owner can select a single peer to perform access control on his objects. The problem is then that the selected peer must be fully trusted and that it has to be available all the time. In a P2P system, such an assumption is not realistic. Therefore, the owner transfers his privileges to perform access control to a *set* of representatives that act on behalf of the owner. Those representatives are called gatekeepers, as they guard the validity of write operations and possibly also of read operations[24].

A natural approach to realize access control would be to send the signing key to each gatekeeper. Although such a solution is simple and efficient, it is useless since a single malicious gatekeeper can sign arbitrary operations, even unauthorized ones. The owner would have to trust all gatekeepers to behave properly. While such an assumption makes sense in a centralized environment where some authority can be liable for its actions, it is not practical in a distributed setting. The nodes selected

[24] Read access control can be implemented implicitly through encryption by disclosing the keys only to authorized entities, which means that gatekeepers do not explicitly perform read access control.

as gatekeepers and their real identity are not known a priori. Therefore, it is more secure to require a certain amount of gatekeepers to jointly agree on the validity of an operation and to mutually sign it. The owner creates a $(t+1, n)$-secret sharing of the signing key and distributes key shares to all gatekeepers. The maximal number t of Byzantine or faulty gatekeepers needs to be chosen properly. The higher t is chosen, the more resilient access control can be performed. In contrast, efficiency decreases since more gatekeepers need to be contacted on an operation. To authorize an operation, at least $t+1$ gatekeeper must mutually sign it. This explains assumption 4.7.2.

Sharing of signing key

Before a signature or store operation can be carried out, the gatekeepers must ensure that all other gatekeepers agree on the same operation. If a malicious entity could convince different subsets of gatekeepers to perform a write operation on distinct data for the same object, the gatekeepers would be in an inconsistent state. Hence, they execute a consensus protocol and authorize the operation that all honest gatekeepers have agreed on. The necessity for a consensus protocol run ultimately leads to assumption 4.7.3.

In the presented schemes, gatekeepers do not even need to make use of digital signatures to authorize write operations. The validity of a self-verifying object is guaranteed by the property that the object's hash is equal to the identifier of the object. If a reader receives at least $t+1$ equal anchor objects from the gatekeepers, it is guaranteed that it is a valid anchor and hence the authenticity of an object can be definitely verified. Distributed signature generation is only necessary when a non-self-verifying object needs to be written. Section 5.10 outlines that this is the case for self-organizing gatekeepers which have to update the witness object jointly.

Note that the protocols state that there must be n gatekeepers available. This does not imply that n is the total number of gatekeepers that were initially chosen by the owner on group creation. The total number might be far larger than n (for example $n+m$, $0 \leq m$), such that a certain amount of gatekeeper failures can be tolerated. It is important that during a run of a consensus protocol, exactly n gatekeepers participate. On one side, the efficiency of the consensus protocol is enhanced as the number of participants is restricted to n. On the other side, the assumptions on a consensus protocol no longer apply if the number of participating entities is larger than n. Therefore, if more than n gatekeepers are available, it is inevitable that the gatekeepers first agree which n gatekeepers participate in the protocol run. For simplicity, it is assumed that $m = 0$ and that n gatekeepers are always available (except for t malicious or failed gatekeepers).

5.2.2 Freshness of Objects

Freshness = improper write operation

Freshness of objects is a crucial issue in an object store. Even latest research proposals do not address this problem although it is severe. Freshness of an object guarantees that the retrieved object is indeed the latest version. Approaches that

do not handle freshness properly implicitly allow improper write operations which implies that access control fails to some extent.

Assume that an attacker can inject an old object version to a reading entity. If there is no authority that the reader can contact to verify that the retrieved object is the latest, the reader has no means to determine the replay attack. The reader thus believes to hold the latest version. However, this implies that the attacker could overwrite the current object by replaying an old one. Access control must be able to prevent the replay attack to retain the semantics of a read operation. Gatekeepers can be contacted by readers to get the latest ID of an object which therefore renders replay attacks impossible.

Before explaining the details of the functionality of gatekeepers, a basic message format is described. Whenever necessary, the presented approaches will extend or modify the general format.

5.2.3 Message Format

The general message format is useful to keep the descriptions as simple as possible. Messages are exchanged between gatekeepers and the owner, readers and writers. First of all, each message contains a type field. That type allows to distinguish messages that are used for different purposes. For example, a message for adding an object to a group must be distinct from a message that aims at removing an object. Since gatekeepers guard access to a group of objects, each message must contain an identifier ID_{Group}. As distinct owners might choose the same identifier for one of their groups, a group additionally needs to be distinguished by the owner's public key PK_O. The public key also allows the gatekeepers to identify the owner and accept changes to the group. To identify the sender of a message, the sender's public key PK_S is included. Since messages can be replayed, a challenge c is provided. Then, the actual data part of the message is added followed by a digital signature. Note that if the owner equals the sender, PK_O equals PK_S. Figure 5.2 illustrates the layout of a message.

Challenge against replay of messages

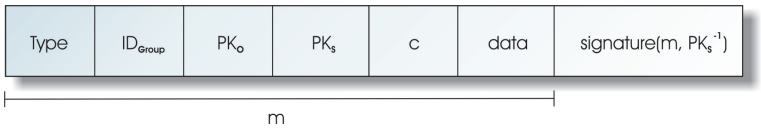

Figure 5.2: The general message format consisting of a group identifier, the owner's public key, the sender's public key, a challenge, the data part and a digital signature of the sender.

The digital signature is a hash of the message that is signed with the sender's private key:

$$signature(m, PK_S^{-1}) = \{h(m)\}_{PK_S^{-1}}$$

The challenge c is incremented in each message exchange. The receiving entity verifies the signature on the message and that the challenge was properly incremented. In the following descriptions, only the data part of the message is listed which is relevant for the request or response. The rest of the message is assumed to be created and sent with the data part. Whenever some part of the message differs from the default message format, it will be stated explicitly.

5.2.4 Gatekeeper Setup

The preliminary goal of an owner is to define a group to which objects are assigned. Entities can play two roles within a group, namely reader or writer. Access control is therefore at the level of groups, which means that groups have to be identifiable by their own identifier ID_{Group}. For each new group, the owner chooses and initializes a set of gatekeeper nodes G with the following steps:

1. The owner O creates a group identifier ID_{Group} such that there is no collision with one of his existent group IDs.
2. O generates a public/private-key pair $\left[PK_R, PK_R^{-1}\right]$ for the group of readers[25].
3. The owner creates an ACL_W and adds the public key PK_{w_k} of each authorized writer $w_k \in W$ (cf. Section 5.4). ACL_W contains a version counter vc_{ACL_W} which is initially set to one and incremented on each update.
4. ACL_W is stored in the object store after having computed the self-verifying identifier ID_{ACL_W}.
5. O then creates a non-self-verifying object, called *witness object* Obj_{ω_G} (see Figure 5.3) with a secure version identifier. The witness object contains the identifier[26] and public keys of all gatekeepers and is stored in the object store.
6. The gatekeepers G are initialized with[27]

   $$vc_\omega, vc_{ACL_W}, ID_{ACL_W}, \text{ and } PK_R$$

   where vc_ω and vc_{ACL_W} are version counters included in the witness object and ACL_W.
7. Since the initialization message of the owner could possibly be a replay message, each gatekeeper retrieves the witness object and verifies whether it contains his public key and whether the witness object is the latest one. Each gatekeeper then replies with $[ID_{\omega_G}, vc_{\omega_G} + 1]$ to the owner.

[25] This step is not necessary for the secret sharing-based approach described in Section 5.7. This also applies to step 6 where PK_R can be omitted.

[26] The identifier can be used to communicate with an entity. As for objects, the identifier for entities is unique.

[27] Section 5.2.3 explained that ID_{Group} and PK_O or PK_S respectively are by default sent along with the message.

8. The owner verifies the signature of the gatekeepers' reply messages to be sure that the update was accepted.

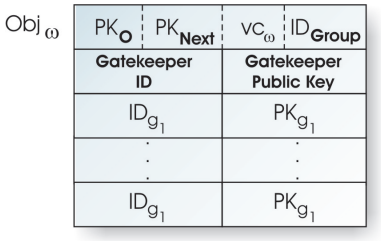

Figure 5.3: The content of a witness object Obj_{ω_G}. The owner's public key PK_O, the version counter vc_ω, the group identifier ID_{Group} and PK_{Next} are interpreted as meta data while the subsequent list denotes all gatekeepers.

Observe that the owner does not yet pass any information about the readers during the initialization phase except for the public/private key pair. Of course, such information needs to be transmitted to the gatekeepers similar to ACL_W. Since different schemes are examined to manage large groups of readers, that information will be added when describing those schemes. It will then also be explained where PK_R^{-1} is stored and how it is protected. The state of a gatekeeper can be summarized as follows:

PK_O and PK_R: The owner's public key PK_O and the reader group key PK_R.

$Anchor_{Obj}, vc_{Obj}$: There is an anchor and version counter for each object of the group.

ID_{ACL_W}: The self-verifying identifier of the ACL_W listing all authorized writers.

ID_{Group}: The group's identifier.

vc_ω, vc_{ACL_W}: Version counters for the witness object and ACL_W.

$counter_{g_i}$: A counter for each gatekeeper g_i which is used to prevent replay-attacks during synchronization operations (see Section 5.2.6).

Since there is no need for generating a public/private-key pair and sending it to the gatekeepers for the secret sharing-based approach described in Section 5.4, the gatekeepers' state does not contain PK_R. Instead, there is an encrypted list of key shares and share-shares for each block i belonging to an $Anchor_{Obj}$:

$\{s_j^{(i)}, s_{1,j}^{(i)}, ..., s_{n,j}^{(i)}\}^{enc}$: A list containing the j^{th} share of block i as well as share-shares of the other gatekeepers' shares (cf. Section 5.4.4). The list is symmetrically encrypted with SK_{Shares} together with the other shares of the same object.

The secret key SK_{Shares} that is used to encrypt the shares is only known to the gatekeeper. The reason for the encryption of the share list is that a gatekeeper can be compromised. In such a case, the shares need to be protected instead of being freely accessible. SK_{Shares} in turn is encrypted with the gatekeeper's public key and locally stored[28]. If a gatekeeper runs out of memory, he can choose to store the encrypted shares in the object store and only maintain a reference to the object. For simplicity, it is assumed that each object's shares and share-shares are encrypted together. Alternatively, one could choose to encrypt each block's shares individually, or conversely, all shares of all objects within a group.

5.2.5 Gatekeeper Localization

The problem of finding the set of active gatekeepers is a crucial issue in the system. Each time the membership of the gatekeeper group changes, a new witness object Obj'_{ω_G} needs to be created. For entities interacting with the gatekeepers it is mandatory to find out which gatekeepers have the authority to perform access control. They can voluntarily or incidentally leave the P2P system or become unavailable. In this case they need to be replaced.

Former gatekeepers can be unavailable

One solution to find the new gatekeepers would be to ask the old ones. However, this approach has several drawbacks. First of all, it is not guaranteed that old gatekeepers are still alive. Secondly, even if they are available the number of malicious gatekeepers could have been increased. This does not violate assumption 4.7.3 since it does not include former gatekeepers. Therefore, one cannot assume a trustworthy authority to be available to answer the question of the most recent set of gatekeepers.

Computation of ID_{ω_G}

Another solution out of this dead end is that each entity computes the witness object's identifier on its own, based on previous information. The witness object uses the secure version identifier explained in Section 4.6 which allows to determine the next version of the witness object with local computations. The name identifier ID_{Name} is set to the identifier of the group ID_{Group} to distinguish between witness objects of distinct groups. Figure 5.3 shows that the witness object contains two public keys: PK_O and PK_{Next}. For the computation of the next SVID, PK_{Next} is used, while PK_O serves as the key to verify the digital signature on the current witness object. Section 5.3.3 justifies why PK_{Next} is necessary.

5.2.6 Gatekeeper Protocols

Gatekeepers always need to be up-to-date to agree on the validity of read or write requests. Whenever a gatekeeper is temporary unavailable, he needs to synchronize with the other gatekeepers when re-joining the group. The protocols to achieve agreement among the honest gatekeepers are described in detail in this section. The theoretical considerations are based on the theory of consensus protocols. Pease et

[28]The gatekeeper's private key is assumed to be protected by a passphrase that is entered on start-up.

al. [53] proved that consensus can only be achieved if the number t of Byzantine nodes is $t < n/3$, where n is the total number of nodes participating in the consensus protocol. The precise consensus protocol can be found in appendix A.

Version Synchronization for List- and Tree-Based Schemes

Gatekeepers can be temporarily[29] offline due to network, software or hardware failures. During the time in which a gatekeeper node does not participate in the actions of the rest of the gatekeepers, information and updates can be lost. Whenever a gatekeeper rejoins the group, he must ensure that his state is synchronized with the other gatekeepers. This is done by contacting at most $2t+1$ gatekeepers and requesting them to send all their information about a group. This includes all $Anchor_{Obj}$, ID_{ACL_W}, ID_{Group}, all version counters and $counter_{g_k}$ for each gatekeeper g_k. The process of synchronizing the local state is as follows:

Synchronization for rejoining gatekeepers

1. g_j determines the current set of gatekeepers using the SVID. If g_j does not find his public key in Obj_{ω_G}, he recognizes that he has been removed from the gatekeeper group, clears all resources and stops the procedure.

2. Otherwise, he sends a message requesting the state of a group to at most $2t+1$ gatekeepers[30].

3. g_k receiving such a request ensures that PK_{g_j} is contained in the witness object and sends $counter_{g_j}$ to g_j.

4. g_j waits for at most $2t + 1$ valid signed messages[31], chooses the messages containing the same values and then replies with a message consisting of

$$[counter_{g_j} + 1, flag].$$

   The $flag$ field indicates whether the state *and* its hash or only the hash of the state should be transmitted. For efficiency reasons, only one gatekeeper is requested to send both the hash *and* the state. For the gatekeepers that do not belong to the $2t + 1$ involved ones, the $flag$ field states that $counter_{g_j}$ should be incremented by one such that all honest gatekeepers' counters have the same value.

[29] Concrete implementations must define "temporarily" by some heuristics.

[30] $2t + 1$ is a limit which will be mentioned in most of the following message exchanges. Since it is known that t is the possible number of malicious gatekeepers, at most $2t + 1$ signatures really have to be verified. Maximally t of them can carry a distinct value, namely those of the corrupt gatekeepers. Having $2t + 1$ valid signatures allows to choose the value of the honest gatekeepers which is simply the majority value. More than t invalid signatures indicate that there might be an attacker modifying the transmitted messages. It mostly makes no sense to verify the subsequent messages since it is likely that they were also manipulated.

[31] At most $2t + 1$ since in the best case, already $t + 1$ are sufficient.

5. g_k verifies that $counter_{g_j}$ has been properly incremented. If this is the case, he assembles his local state for the specified group. According to the $flag$ field, the gatekeeper either transmits the whole state or only the hash of the state. Then, the counter is updated to $counter_{g_j} := counter_{g_j} + 1$ and locally stored.

6. g_j verifies at most $2t + 1$ signatures and decides for the majority hash value of the state. If the transmitted state's hash equals the majority hash, then g_j has an up-to-date state. Otherwise, g_j continues polling one single gatekeeper at the time until a valid state is received. The polling messages include $counter_{g_j} + 2$.

One can observe that the protocol is complex regarding challenge-response rounds. The reason is that the local state can be very large which makes a denial of service attack quite straightforward. Although a message always contains a challenge c, the gatekeepers send an additional challenge in form of a counter to ensure that the messages coming from g_j are no replays. Moreover, gatekeepers only accept polling messages during a short time frame.

Version Synchronization for Sharing-Based Schemes

Synchronize share-shares

The protocol of the previous section needs to be adjusted for the sharing-based approach. The reason is that gatekeepers hold shares of the block keys. If the rejoining gatekeeper missed a write operation, he does not know the writer's key share. Therefore, he needs additional information from the other gatekeepers, namely the suitable share-shares to reconstruct the full key share (details are explained in Section 5.4.4). The previous protocol needs to be adjusted from step 4 on:

4. g_j waits for at most $2t + 1$ valid signed messages, chooses the messages containing the same values and then replies with a message consisting of

$$[counter_{g_j} + 1, flag]$$

   For the gatekeepers that sent the same value, $flag$ indicates that the state should be transmitted including their key shares for g_j. For all remaining gatekeepers, $flag$ states that $counter_{g_j}$ has to be incremented by one such that all honest gatekeepers' counters have the same value.

5. g_k verifies that $counter_{g_j}$ has been properly incremented. If this is the case, he assembles his local state for the specified group. This also includes the share-shares for each block i of an object: $\{s^{(i)}_{1,k}, ..., s^{(i)}_{n,k}\}$. Additionally, g_k picks the j^{th} share-share $s^{(i)}_{k,j}$ of his own share for each block i such that g_j can also help to synchronize other gatekeepers. For the transmission, all shares are encrypted with PK_{g_j}. Then, the counter is updated to $counter_{g_j} := counter_{g_j} + 1$ and locally stored.

6. g_j verifies at most $2t + 1$ signatures and decides for the majority hash[32] value of the transmitted states. Messages of gatekeepers which have an incorrect hash value are discarded.

7. Having a correct state, g_j needs to reconstruct his key shares for each block of each object. From the correct replies of the gatekeepers, he decrypts the key shares. To verify that a single share-share is correct, he must retrieve the key share object (KSO, cf. Section 5.4.4) which is identified by ID_{KSO} in the header object for each block. Each share-share can be verified against the hash value in KSO (red part of Figure 5.8). Invalid values are disregarded.

The main difference to the version synchronization for list-/tree-based schemes is that the key shares need to be reconstructed using key share objects to verify authenticity. This also has an influence on the performance of the version synchronization (cf. Section 5.11).

Gatekeeper Nomination

The owner can occasionally conclude not to believe in at least $n-t$ gatekeepers to be honest. He therefore needs means to withdraw their authority and to replace them by new gatekeepers. The owner removes the corresponding entries from the witness object Obj_{ω_G} and replaces them with the public keys of the new gatekeepers. He increments vc_ω and files Obj_{ω_G} to the object store. Next, he needs to inform both old and new gatekeepers about those changes (except those which were removed from the group of gatekeeper):

1. The owner requests the state from the gatekeepers following the description of Section 5.2.6.

2. The gatekeepers which have been selected to join the group are initialized with the state (cf. Section 5.2.4).

3. The owner sends to all gatekeepers a witness update message containing the new witness object's version counter vc_ω.

4. Since old gatekeepers are in possession of the old version counter vc'_ω, they only accept the update message if $vc_\omega > vc'_\omega$. Otherwise, they detect the replay message and discard the update.

5. As for the gatekeepers' initialization, he receives $[ID_{\omega_G}, vc_{\omega_G} + 1]$ from the gatekeepers as confirmation for the update.

The next sections explain access control on a read or write request. List-based and the tree-based write access control are identical, while the sharing-based scheme differs slightly. First, the procedure used for the list- and tree-based scheme is explained followed by the steps for the sharing-based approach.

[32]The hash of the state does not include the list of key shares.

5.2.7 List- or Tree-Based Access Control

The list- or tree-based approach relies on the fact that the group of readers is in possession of a reader group key, consisting of a public part PK_R and a private part PK_R^{-1} only known to authorized readers. Before explaining how gatekeepers perform access control, it is necessary to discuss the content of the header object.

Header Content for List- and Tree-Based Access Control

With regard to the structure introduced in Section 4.3, the base layout of an object remains unchanged. However, some additional values need to be added for the list-based approach to the header such that one entry consists of:

$$ID_{obj_i}, ID_{KLO}, \{SK_i\}_{PK_R}$$

where ID_{obj_i} is the identifier for block obj_i^{enc}, ID_{KLO} is the identifier of a *key list object* (KLO) that contains the key material and $\{SK_i\}_{PK_R}$ is the secret key encrypted with the readers' public key matching the private key PK_R^{-1} within KLO. Note that ID_{obj_i} and ID_{KLO} are both self-verifying identifiers.

Compared to the list-based header object, the content for the tree-based approach only differs in the identifier ID_{KTO} since the keys are now managed within a *key tree object* (KTO) instead of a key list.

$$ID_{obj_i}, ID_{KTO}, \{SK_i\}_{PK_R}$$

Figure 5.4 illustrates the layout of the object header whereas ID_{KeyObj} stands for either ID_{KLO} or ID_{KTO}.

Write Access Control

For simplicity, it is assumed that a writer w_k only writes to one particular block obj_i. The general case where the writer operates on a set of blocks can easily be adopted from the base case. A write request proceeds as follows:

1. A gatekeeper g_j receives a request from a writer w_k for a certain object with identifier ID_{Obj}. This message does not need to be signed, but requires the existence of a challenge.

2. Without verifying the writer's identity, g_j picks the object's version vc_{Obj} and sends to w_k:

$$[PK_R, vc_{Obj}]$$

3. g_j receives a write request from writer w_k consisting of

$$[ID_{Obj}, SK_i^{enc}, ID_{obj_i}, i, r]$$

4. The gatekeeper g_j ensures that his challenge from step 2 has been properly incremented: $r = vc_{Obj} + 1$.

5. g_j retrieves ACL_W and ensures that it is valid by comparing it with identifier ID_{ACL_W}. Then a lookup is performed to retrieve w_k's public key PK_{w_k}. If there is no entry for that writer, the request is aborted.

6. g_j retrieves the current object header H_{Obj} and replaces the i^{th} entry by

$$[ID_{obj_i}, SK_i^{enc}, ID_{KeyObj}]$$ [33]

and updates the version counter within H_{Obj} to $vc_{Obj} + 1$. The $ID'_{H_{Obj}}$ of the new header H'_{Obj} is computed. Figure 5.4 illustrates the precise content of the header object.

7. The consensus protocol A.1.2 is executed on $ID'_{H_{Obj}}$.

8. Let ID_{result} be the return value of the consensus protocol. The anchor's value is updated to ID_{result}. If ID_{result} is equal to $ID'_{H_{Obj}}$, then the new header H'_{Obj} is stored in the object store and $vc_{Obj} := vc_{Obj} + 1$.

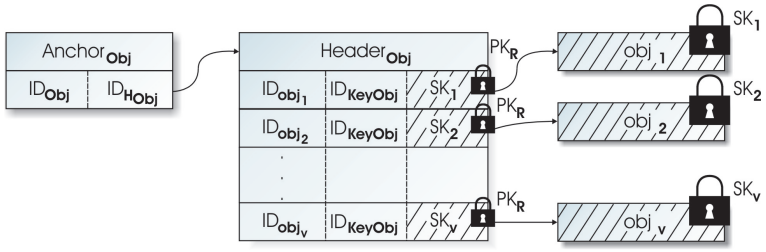

Figure 5.4: The content of a header object which consists of the encrypted key of the block and the identifier to the object that allows authorized readers to retrieve the corresponding private key. ID_{KeyObj} and PK_R can be distinct for each entry.

Two challenges for both directions

At first sight it is questionable why there is a need for sending an additional challenge in form of vc_{Obj}. The increased challenge r guarantees that the reply from the writer is fresh because write operations must refer to the current object version. On the other side, the challenge c of the message guarantees the writer that his communication with the gatekeepers is fresh.

[33] ID_{KeyObj} is either ID_{KLO} or ID_{KTO}, depending on whether the scheme is based on key lists or on key tree objects. ID_{KeyObj} is an information that gatekeepers hold in their local state.

Read Access Control

Gatekeepers do not have to perform read access control explicitly for the list- and tree-based schemes. Readers invoke gatekeepers only to find out what the latest $ID_{H_{Obj}}$ of an object is. A request of a reader consists of the object's identifier ID_{Obj} and does not need to be signed by the reader. Gatekeepers will send back

$$[Anchor_{Obj}, ID_{KeyObj}].$$

5.2.8 Sharing-Based Access Control

In the sharing-based scheme, the gatekeepers' task is to perform access control for both writers *and* readers. As for the list- and tree-based approach, the gatekeepers need to agree on a write operation before updating the anchor. The difference is that they need also to manage keys explicitly. A writer distributes his key shares on a write operation while readers need to retrieve those shares to reconstruct the full key.

Header Content for Sharing-Based Access Control

Compared to the list-based or tree-based header object, the content for the sharing-based approach does no longer require to maintain an encrypted secret key. Instead, there is an identifier ID_{KSO} to a key share object that allows to verify the authenticity of received shares (cf. Section 5.4.4).

$$ID_{obj_i}, ID_{KSO}$$

Figure 5.5 illustrates the layout of the header object while Figure 5.8 shows the key share object.

Write Access Control

A writer needs to create shares of his secret key SK which are distributed to all gatekeepers. Since authorized readers retrieve the shares from the gatekeepers, they must have means to determine the authenticity of the shares. This is accomplished by a key share object (KSO) for which the writer passes an identifier ID_{KSO}. The write access control procedure for a gatekeeper g_j is as follows:

1. A gatekeeper g_j receives a request from a writer w_k.
2. Without verifying the writer's identity, g_j generates a new challenge c_2 and sends it to w_k.
3. g_j receives a write request from writer w_k consisting of

$$[ID_{Obj}, ID_{obj_i}, \{s_j^{(i)}, s_{1,j}^{(i)}, ..., s_{n,j}^{(i)}\}^{enc}, ID_{KSO}, i, c_2']$$

Note that the share $s_j^{(i)}$ and the share-shares are encrypted with the gatekeeper's public key PK_{g_j}.

4. The gatekeeper g_j ensures that his challenge from step 2 has been properly incremented: $c_2' = c_2 + 1$.

5. g_j retrieves ACL_W and ensures that it is valid by comparing it with identifier ID_{ACL_W}. Then a lookup is performed to retrieve w_k's public key PK_{w_k}. If there is no entry for that writer, the request is aborted.

6. g_j retrieves the current object header H_{Obj} and replaces the i^{th} entry by

$$[ID_{obj_i}, ID_{KSO}]$$

Figure 5.5 illustrates the content of the header object. Next, the $ID'_{H_{Obj}}$ of the new header H'_{Obj} is computed.

7. The gatekeeper agreement protocol A.1.2 is executed on $ID'_{H_{Obj}}$.

8. Let ID_{result} be the return value of the agreement protocol. The anchor's value is updated to ID_{result}. If ID_{result} is equal to $ID'_{H_{Obj}}$, then the new header H'_{Obj} is stored in the object store. $\{s_j^{(i)}, s_{1,j}^{(i)}, ..., s_{n,j}^{(i)}\}^{enc}$ is decrypted and securely stored by encrypting it with SK_{Shares} of the gatekeeper.

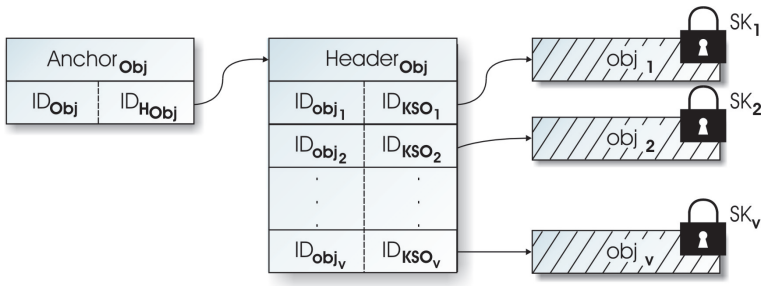

Figure 5.5: The content of a header object which consists of references to blocks and identifiers to the key share object (KSO) such that readers can verify the authenticity of the retrieved shares.

Should key share authenticators be verified?

One can think about the necessity of verifying that the shares' authenticators are valid by comparing them with the hash values in KSO. In fact, there is no need for gatekeepers to perform such a verification. Although a writer could generate some invalid shares and the gatekeepers would agree on a common object anchor, the writer's behavior can later be detected by a reader and the writer can be banned from the system[34]. A reader would then assume that all bits in the corresponding

[34]To prove a writer's malicious behavior, the writer must sign ID_{KSO} to provide provableness to 3^{rd} parties. Since the presented schemes do not consider mechanisms for entity exclusion, the writer's signature is set aside.

blocks are zero. Alternatively, if versions of objects are supported, an older version could be accessed.

Read Access Control

A request of a reader r_m to retrieve a particular object Obj consists of the object's identifier ID_{Obj}. A gatekeeper g_j proceeds as follows:

1. g_j performs an explicit challenge-response round[35], retrieves ACL_R and verifies it against ID_{ACL_R}.
2. If $PK_{r_m} \notin ACL_R$ then the request is aborted.
3. Otherwise g_j sends to r_m:

$$[Anchor_{Obj}, list^{enc}_{shares}]$$

where $list^{enc}_{shares} = \left\{ \left[s_j^{(1)}, ..., s_j^{(k)} \right] \right\}_{PK_R}$ with $s_j^{(i)}$ being the j^{th} share of block i assuming that the object has k blocks.

In contrast to the tree-based approach, gatekeepers need to encrypt some part of the reply message, namely the list of secret shares. In an efficient system, the gatekeeper would generate a new symmetric session key to encrypt the shares and then encrypt the session key with the reader's public key.

[35] This means, that the reader creates a nonce and sends it to $2t+1$ gatekeepers. The gatekeepers in turn use the current version counter vc_{Obj} of the object as a reply challenge and send it to the reader with the reader's challenge increased by one. If the reader is indeed an authorized entity, it can reply to the challenge with a signed message. Only if the reply was valid, the gatekeepers send the anchor and all key shares as described in the following step.

5.3 Operations on Groups

In the last section, the initialization of gatekeepers and their functionality has been described. This section aims at explaining how changes to a group are carried out and how gatekeepers need to be invoked.

5.3.1 Adding New Objects to a Group

An owner needs to add objects explicitly to a group. Section 4.3 already explained how objects are structured and introduced the notion of an anchor object. When adding an object to a group, the owner sends a message containing $[Anchor_{Obj}, vc_{Obj}]$ to all gatekeepers, where vc_{Obj} is the version field contained in the header object. The gatekeepers maintain a local storage which associates a group of an owner with a set of anchors and their versions. They will only accept the anchor if the sender equals the group's owner and if the anchor for that object does not yet exist. Removal of an object from a group is analogous to the process of adding an object.

5.3.2 Moving Objects Between Groups

Given two distinct groups A and B with an object Obj that has been assigned to group A. An owner may decide to move Obj from A to B such that readers in group B can access it and entities from group A cannot read the content of newly written blocks from that time on. Unfortunately, entities in group B do and must not know the keys of group A. Nevertheless, without the proper keys, the Obj is not readable for members in B. To address this problem, one has to distinguish between the list-/tree-based scheme and the sharing-based scheme.

Moving Objects in List- and Tree-Based Schemes

Readers of group B are not allowed to infer the keys of group A because otherwise, they would be able to read all objects. There are two possibilities with different complexity constraints.

No Intervention The simplest way is not to perform any operation besides removing Obj at group A and adding it to group B. As mentioned, readers of group B cannot read the content of the object unless blocks are overwritten. Although this approach is cheap, it does not meet the expected semantics of a read operation since some blocks are likely to be unreadable (cf. Section 5.8).

Block Key Re-Encryption Another solution is to create a new header object. That header object contains the same block references as the original one. However, the owner decrypts all block keys of the old header by using his backdoor key and adds them to the new header object encrypted with the latest PK_R of group B. For versioned objects, access to old versions is still impossible. However, the owner can set the reference to the previous version to $NULL$ and pretend that this is the first

version of the object.

Although re-encryption of the block keys is linear in the number of blocks, it seems more natural to follow this approach to achieve common-sense read semantics.

Moving Objects Using a Sharing-Based Scheme

Since the symmetric block keys are directly shared among the gatekeepers of A, the owner can request the shares of all keys, reconstruct the symmetric keys and re-share the keys among the gatekeepers of B. Readers in B can then access Obj without having the knowledge to read objects in A.

5.3.3 Ownership Transfer

The process of transferring ownership of a group from an owner O_{Old} to another entity O_{New} requires two steps: Foremost, the members of the group must be informed about the transfer of the ownership since the localization of gatekeepers is based on the owner's public key according to Section 5.2.5. Then, the gatekeepers need to be informed about those changes and accept the entity O_{New} as the new owner of the group.

In Figure 5.3, two public keys are depicted: PK_O and PK_{Next}. If the owner does not change, PK_{Next} is equal to PK_O. Whenever O_{Old} decides to transfer ownership to O_{New}, PK_{Next} is set to $PK_{O_{New}}$. After having stored the new witness object, the owner sends to each gatekeeper a message containing the version counter of the new witness object and the public key of the new owner:

$$PK_{O_{New}}, vc_{\omega}$$

From that point on, only O_{New} can perform changes to the group. Entities that are determining the latest witness object must use PK_{Next} for the computation of the next SVID.

5.3.4 Multi Ownership

In some situations, it is desirable to have more than one owner for a group. For example, there might be several administrators that can add or remove subjects or objects to or from a group. Hence, each of them needs to have the privileges of an owner. Initially, all entities that want to belong to a group of owners need to agree on a common public/private-key pair. That key pair is used as the owner's key. The gatekeepers do not recognize that the requests are issued from different entities. From their viewpoint, the owner is one logical unit. Note that each entity which is an owner has the authorization to transfer the ownership following the explanations of the previous section. This allows to withdraw the ownership from all other entities without their approval. Hence, entities willing to form a group of owners must ensure that they trust each other and be aware of the possibility to lose control over the group.

5.3.5 Groups of Groups

In general, one can think of a hierarchical layout of groups such that being member of some group implies being member of a certain number of subgroups. A straightforward way is to handle supergroups like ordinary groups. This means that each supergroup maintains an access control list and a key object if necessary and is guarded by a set of gatekeepers. All access control mechanisms work as explained in the previous sections. The gatekeepers of the supergroup can handle all requests that affect objects which were added to that group. All requests to objects of subgroups must be delegated to the responsible gatekeeper set. But there are some difficulties that must be taken care of.

First of all, the owner must add all members of the supergroup to the access control data structures of the subgroups. Let's consider the example from Figure 5.6. A

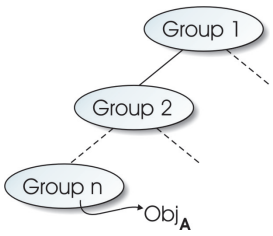

Figure 5.6: A scenario for a hierarchy of subgroups.

writer w being member of group 1 must be added to ACL_W of all subgroups. The same holds for a reader of group 1. Conversely, a leaving entity must be removed from all data structures of all subgroups. Therefore, changing reader or writer membership of a supergroup is as expensive as the number of subgroups.

Secondly, consider an object Obj_A which belongs to group n. If a member of group 1 attempts to access Obj_A, the request needs somehow to be directed to the gatekeepers of group n because the gatekeepers of group 1 do not know the anchor of Obj_A. As in many cases, there is an insuperable trade-off between space and time complexity. If access to an object should be efficient in time, the owner needs to provide a mapping of object identifiers to witness object identifiers. On a request to an object of a subgroup, the gatekeepers return the appropriate witness object identifier. That identifier can be used to locate the gatekeepers of the subgroup[36]. Unfortunately, the gatekeepers of the supergroup require to store the mapping of all subgroups. Moreover, if the owner adds or removes an object to or from a subgroup, all gatekeepers of the supergroups need to be notified.

To optimize the space complexity of gatekeepers, one can omit the mapping and

[36]Recall that the witness object of the subgroup might have changed. The returned identifier might only serve as a starting point to locate the most recent witness object following the procedure of Section 5.2.5.

shift the problem of finding the appropriate gatekeepers to the requesting entity. For that purpose, witness objects need to be extended with references to the witness objects of their direct subgroups. Figure 5.7 illustrates the new layout. Coming

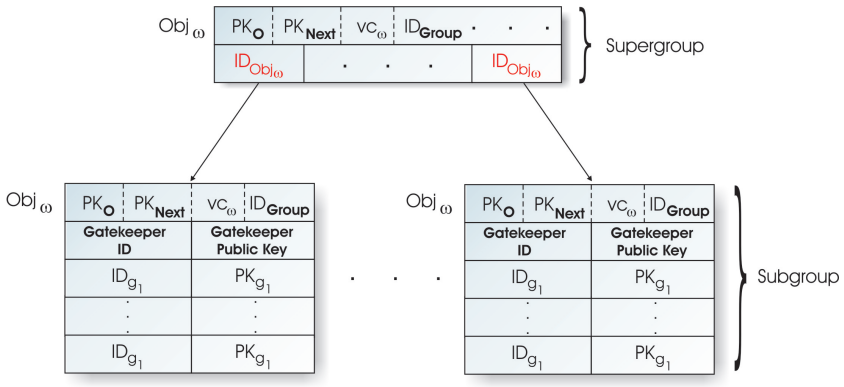

Figure 5.7: Witness objects in a hierarchical structure.

back to the example of Figure 5.6, a request for Obj_A is rejected by group 1 since Obj_A is unknown. The requesting entity then needs to query the subgroups until the responsible gatekeepers for Obj_A are found. This approach is only feasible if the overall number of subgroups is small.

Hierarchies are expensive

Since hierarchies of groups are expensive, they should be avoided. It is sufficient to follow the observation from Section 2.8.1 which states that roles and groups are a dual concept. In many cases, one entity has several roles which translates to: One entity belongs to several groups. Each entity must be aware of its group memberships and store this information locally. Entities then access the gatekeepers of a group directly[37].

[37] The problem of knowing the mapping of objects to groups is shortly addressed in Section 8.2.

5.4 Access Control for Writers

In this section, the procedure of write access control from a writer's point of view is explained. The steps for a write are the same for the list-based and tree-based approach. The sharing-based approach requires a different handling of the key material. Before discussing the messages that are exchanged, it is briefly summarized what the content of an access control list is, how it is constructed and how updates can be performed on.

5.4.1 Access Control Lists

To perform access control properly, the owner needs to provide additional information. The owner can grant permissions on his objects at his discretion by choosing the entities having privileges explicitly. He thus maintains an access control list ACL_W (cf. Section 2.5.5) listing public-keys or public-key certificates of all authorized writers $w_k \in W$. The owner can optionally add his own public key.

ACL_W is stored in a replicated way in the object store using a self-verifying identifier ID_{ACL_W}. Note that it does not make sense to encrypt the content of ACL_W. Although it can be desirable to hide the information of group membership, it is impossible to protect privacy by encryption in this case. Assume that ACL_W is encrypted and the decryption key is passed to all gatekeepers as each gatekeeper needs access to the full AC list to verify whether a writer is authorized. Thus, one malicious gatekeeper is already sufficient to reveal membership information. By assumption 4.7.3, the number of Byzantine gatekeeper is usually larger than one. In case that privacy of group membership is a critical issue, there is a more expensive solution: For each writer, the owner initially creates a fresh public/private-key pair and transmits it to the writer encrypted with his regular public key. The new public keys are written to ACL_W which is not encrypted. As long as the writer does not announce his new public key, only the owner and the writer know to whom the new public key belongs to.

Why not encrypting ACL_W?

5.4.2 Access Control List Update

Whenever a writer joins or leaves the group, ACL_W needs to be updated. Updates on ACL_W are exclusively performed by the owner O:

1. O updates ACL_W accordingly, increases the version counter vc_{ACL_W} by one, computes ID_{ACL_W} and stores it in the object store.

2. The owner contacts all gatekeepers and sends

$$[ID_{ACL_W}, vc_{ACL_W}].$$

3. The owner receives the update confirmations of all gatekeepers in the form of:

$$[ID_{ACL_W}, vc_{ACL_W} + 1].$$

Gatekeepers only accept ACL updates that were sent by the owner of the group and if vc_{ACL_W} is larger than the version counter that is locally stored.

As mentioned in Section 5.2.7 and 5.2.8, the procedure for write access control depends on whether a list-/tree-based or a sharing-based approach is taken. The two following sections explain the actions on a write request for both types of schemes separately.

5.4.3 List- or Tree-Based Writer Access Control Procedure

When an authorized writer w_k wants to commit his write operation, he needs to convince all honest gatekeepers that he is authorized for that kind of operation:

1. Assume that a writer $w_k \in W$ has written a block obj_i of an object Obj.
2. w_k generates a fresh secret key SK which is used to encrypt the written block resulting in obj_i^{enc}.
3. The identifier ID_{obj_i} is computed and the block is stored in the object store.
4. w_k finds the latest witness object Obj_{ω_G} as described in Section 5.2.5.
5. To retrieve the group's public key PK_R, w_k contacts $2t + 1$ gatekeepers and request PK_R passing the object identifier ID_{Obj}. This message does not have to be signed.
6. From a gatekeeper g_j, the reply message consists of:

$$PK_R, r$$

7. w_k makes use of the PK_R to encrypt the secret key:

$$SK^{enc} = \{SK\}_{PK_R}$$

8. w_k sets $r' = r + 1$ and contacts all n gatekeepers passing:

$$[ID_{Obj}, ID_{obj_i}, SK^{enc}, i, r']$$

9. The writer does not need to take further actions as the process of storing the block and header is done by the gatekeepers collectively.

The fact that all honest gatekeeper store a vc_{Obj} speeds up the process of writing an object by $n-1$ signatures. If each gatekeeper would choose a random number as challenge, then w_k would have to create n signatures to reply to all challenges. But since all correct gatekeepers send the same vc_{Obj}, the writer only has to create one signature.

5.4.4 Sharing-Based Writer Access Control Procedure

The secret sharing-based approach requires to protect each single share by encryption. Encryption is necessary to guarantee secrecy for the transmission from the writer to the gatekeepers. To guarantee authenticity of his shares, the writer creates a *key share object*.

Key Share Object

The goal of a secret sharing in the context of access control is that authorized readers can reconstruct the decryption key from the shares. The gatekeepers are charged with holding the shares and only reveal them to proper readers. Since t gatekeepers can be corrupted, they can also modify the stored shares. A reader who operates on manipulated shares will finally reconstruct an incorrect key which implies that decryption of data fails. Therefore, a reader must be able to verify that the retrieved shares are authentic. This also applies to gatekeepers that are executing the synchronization protocol in Section 5.2.6 which requires to reconstruct the key shares from share-shares.

In principal, a writer can sign all his shares and share-shares individually with his private key. The problem with this approach are the high costs for signature generation and verification. A writer has to create n shares and n^2 share-shares which all need to be signed, where n is the number of gatekeepers. Even worse, a reader which accesses an object with k blocks, where each block is encrypted with a distinct secret key, has to verify at least tk digital signatures. For large objects, this is computationally expensive. Therefore, the following approach minimizes the number of digital signatures, but requires the existence of a *key share object* (KSO), which allows to verify the authenticity of a single share or share-share.

KSO guarantees authenticity of shares

A writer who has written a block i creates n shares $s_1^{(i)}, ..., s_n^{(i)}$ at polynomial evaluation points $\alpha_1^{(i)}, ..., \alpha_n^{(i)}$ for the secret key SK_i. Additionally, the writer generates share-shares[38] of each share $s_j^{(i)}$, $j = 1, ..., n$:

$$\begin{aligned} s_1^{(i)} &\rightarrow s_{1,1}^{(i)}, ..., s_{1,n}^{(i)} \\ &\vdots \\ s_n^{(i)} &\rightarrow s_{n,1}^{(i)}, ..., s_{n,n}^{(i)} \end{aligned}$$

Then, the writer creates a *key share object* (KSO) which contains for each evaluation point $\alpha_j^{(i)}$ the hash $h(s_j^{(i)})$ of the corresponding share $s_j^{(i)}$ and the hash values of the share-shares (red part of Figure 5.8). The self-verifying KSO does not need to be encrypted or signed.

[38] For simplicity, it is assumed that the evaluation points are equal to $\alpha_1^{(i)}, ..., \alpha_n^{(i)}$ for the generation of the share-shares. Otherwise, the evaluations points need to be listed explicitly.

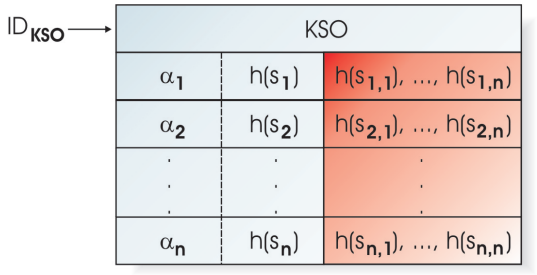

Figure 5.8: The key share object (KSO) contains authenticators for the shares and share-shares. For simplicity, the index i of the block has been omitted in the diagram.

Access Control Procedure for a Writer

Using the KSO, a writer can guarantee authenticity of his shares with a minimal number of cryptographic operations. Only the messages to the gatekeepers need to be encrypted and signed. Assume that a writer $w_k \in W$ has written a block obj_i of an object Obj. The procedure to convince gatekeepers to accept the write operation is as follows:

1. w_k generates a fresh secret key SK_i used to encrypt the written block, resulting in obj_i^{enc}.
2. The identifier ID_{obj_i} is computed and the block is stored in the object store.
3. w_k finds the latest witness object Obj_{ω_G} as described in Section 5.2.5.
4. w_k contacts all gatekeepers[39]. Each gatekeeper g_j sends a challenge c_j.
5. The writer creates a sharing of SK_i, generates the KSO with identifier ID_{KSO} as described previously and files it to the object store.
6. For each gatekeeper g_j, the writer computes

$$list_{shares,j}^{enc} = \{s_j^{(i)}, s_{1,j}^{(i)}, ..., s_{n,j}^{(i)}\}_{PK_{g_j}}$$

7. w_k contacts each gatekeeper g_j passing:

$$ID_{Obj}, ID_{obj_i}, list_{shares,j}^{enc}, ID_{KSO}, i, c_j + 1$$

8. The writer does not need to take further actions as the header is stored by the gatekeepers collectively.

Gatekeepers include ID_{KSO} into the header object such that readers can retrieve the key share object (see Section 5.2.8).

[39] There is no need for signing this message as the only reason for the request is to obtain a fresh challenge.

5.5 List-Based Access Control for Readers

List-based access control is one of the simplest, yet quite efficient solutions to manage the keys that readers need when accessing an object. Section 5.4 explains that blocks are encrypted with a symmetric key SK while Section 5.2.7 mentions that gatekeepers update the header object H_{Obj}. SK is encrypted with a public key PK_R of the group of readers.

5.5.1 Key List Object

The essential question is how keys need to be managed within a list such that the number of cryptographic operations is minimal when changing reader membership or when a reader accesses the list to decrypt a block. First, it is stated that there is no pre-shared secret between the owner and a reader. Hence, at least one public-key operation is inevitable per reader. The key list nevertheless tries to minimize the number of public-key operations since they are significantly more expensive than secret-key operations.

5.5.2 Reader Access Control Procedure Based on Key List Objects

The general idea of the key list object is to create a symmetric key for each reader which is encrypted with the corresponding public key. Each secret key is then used to encrypt the reader group key PK_R^{-1}. Figure 5.9 shows the structure of a key list.

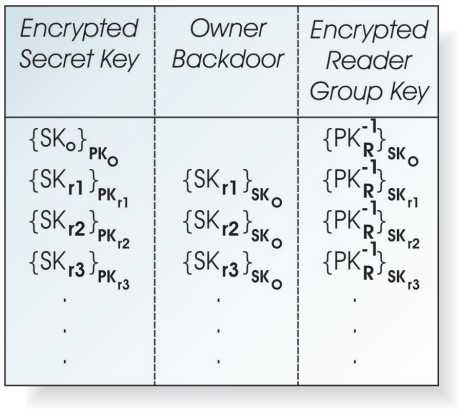

Figure 5.9: Structure of a key list object (KLO), where the first column contains the encrypted secret key for each reader and the owner, the second column makes the reader's symmetric key accessible to the owner, and the last column contains the reader group key encrypted for each reader.

As mentioned, there is no way to avoid one public key operation per reader. The intent of the second column is to allow the owner to access each reader's symmetric key by only using symmetric cryptography. The last column contains the reader

group key which is encrypted with the symmetric key of each reader.

During the gatekeeper initialization phase described in Section 5.2.4, the owner creates the key list containing the entries for all authorized readers. The key list with identifier ID_{KLO} is filed to the object store. Similar to the access control list for writers, ID_{KLO} and vc_{KLO} need to be transmitted to the gatekeepers and added to their state.

5.5.3 Joining Reader

The owner can change the membership of the reader group by adding or removing entries to the key list. For a joining reader r_k, the owner creates a new secret key SK_{r_k}. That secret key needs to be encrypted using the reader's public key PK_{r_k}. The owner will then use SK_{r_k} to encrypt the reader group key PK_R^{-1}. Since the owner needs to have a backdoor to perform updates on the key list, he also encrypts SK_{r_k} using his own key SK_O. As Figure 5.9 illustrates, the owner can access SK_O by using his private key PK_O^{-1}. Finally, the owner sends

$$[ID'_{KLO}, vc'_{KLO}]$$

to all gatekeepers to update their state accordingly and initializes the joining reader with PK_O, vc_{ω_G} and ID_{Group}. From all gatekeepers, the owner receives $[ID'_{KLO}, vc'_{KLO} + 1]$ as update confirmation.

5.5.4 Leaving Reader

As explained in Section 5.4.3, a writer makes use of the reader group key PK_R to encrypt the symmetric key that was used for encryption of the block. To protect the content of future write operations, the owner needs to create a new public/private-key pair $[PK'_R, PK'^{-1}_R]$. Then, he removes the entry from the key list which belongs to the reader to be removed. The owner first decrypts SK_O using his own private key. For each entry in the key list, the owner decrypts all secret keys of the backdoor column. Now having access to all symmetric keys, the owner can use them to encrypt PK'^{-1}_R and add the encryptions to the last column. The version counter vc_{KLO} is incremented by one and the list is stored to the object store using the new identifier ID'_{KLO}.

Next, the owner needs to inform the gatekeepers about this change by sending

$$[ID'_{KLO}, vc'_{KLO}, PK'_R]$$

to all gatekeepers. Each gatekeeper ensures that vc_{KLO} is larger than the version counter they are storing before accepting the update. The owner receives $[ID'_{KLO}, vc'_{KLO} + 1]$ as update confirmation from all gatekeepers.

5.5.5 Reader Access Control Procedure

A request of a reader r_j can be described as follows:

1. r_j finds the latest witness object Obj_{ω_G} as described in Section 5.2.5.
2. r_j contacts at most $2t + 1$ gatekeepers requesting the anchor and ID_{KLO}.
3. r_j retrieves a valid header H_{Obj} which can be verified against $ID_{H_{Obj}}$ of the anchor.
4. If obj_i is about to be read, the reader retrieves a replica and verifies it against ID_{obj_i}.
5. r_j extracts the ID_{KLO} from H_{Obj} of the i^{th} entry and retrieves a valid key list object.
6. Finding the entry matching the reader's public key, the reader can first decrypt the symmetric key SK_{r_j} before decrypting PK_R^{-1}.

Why requesting ID_{KLO}?

There is a subtle detail which is nevertheless worth to be mentioned. In step 2, the reader requests the anchor as well as the latest ID_{KLO}. At first sight, this might seem unnecessary since the header object H_{Obj} also contains an identifier ID'_{KLO} for each block. The problem is that an authorized reader might be unable to read certain blocks of an object since he might not have access to all key list objects. Section 5.8 will extend the key list and key tree data structures to solve this problem using the latest ID_{KLO}.

5.6 Tree-Based Access Control for Readers

The tree-based approach tries to minimize the number of cryptographic operations in the average case. The list-based approach is in the order of $O(m)$ symmetric key operations, where m is the number of readers within the group. Due to the property of trees, the number of symmetric key operations can be decreased to $O(\log m)$ in the average case. Operations are efficient even if m is very large. Open questions include what information the tree nodes store, which entities can perform what kind of operations on the tree, where the tree is stored and how it can be found.

5.6.1 Reader Access Control Procedure Based on Key Tree Objects

To handle group and key changes appropriately, an approach similar to group key management in VersaKey [39] is examined. The basic idea is to provide a tree-based key management scheme to realize efficient key updates. In VersaKey, the root key is called *traffic encryption key* (TEK) while inner node keys are called *key encryption key* (KEK). As opposed to VersaKey, the function of the root key is similar to the other keys in the tree as it will also be used for encrypting other keys[40].

Assume that an owner O shares an object $Obj \equiv \{obj_1, ..., obj_v\}$. The keys which are needed to decrypt the blocks have to be managed appropriately such that all authorized readers $R = \{r_1, ..., r_m\}$ can retrieve the content of the blocks efficiently. The overall goal of the tree is that all readers share a private key PK_R^{-1} which is set as the root of the tree. The leaves of the tree consist of symmetric keys encrypted with the readers' public keys. A reader can then access the private key at the root by finding the appropriate leaf and decrypt the path from the leaf up to the root. The owner creates the tree in the following manner:

1. A backdoor key SK_O is generated which is encrypted with the owner's public key PK_O.
2. O generates m keys SK_i for each reader r_i as leaves of the tree.
3. A binary tree is built up by generating fresh secret keys $SK_{i,i+1}$ for each i where $i \, mod \, 2 = 0$, such that $SK_{i,i+1}$ is the parent node of the leaves SK_i and SK_{i+1}.
4. The owner continues this procedure until the root of the tree is reached. The root, in contrast to all other inner nodes, does not contain a symmetric key, but the private key PK_R^{-1} of the group.
5. The owner creates a so called *key tree object* (KTO) that contains the *encrypted* tree. Each $SK_{i,i+1}$ is encrypted twice, once with his left child SK_i and once with his right child SK_{i+1}. As the leaf keys do not have any successors, they are encrypted with the readers' public keys. Since the owner needs

[40] However, the root contains an asymmetric key while the keys of the inner nodes and leaves are symmetric.

to perform updates on the tree, he also encrypts every leaf with his backdoor key SK_O. To allow a reader to find his encrypted leaf efficiently, the owner can attach the reader's identifier to a leaf or provide a lookup table.

6. The owner computes the self-verifying identifier ID_{KTO} and stores the tree in the object store.

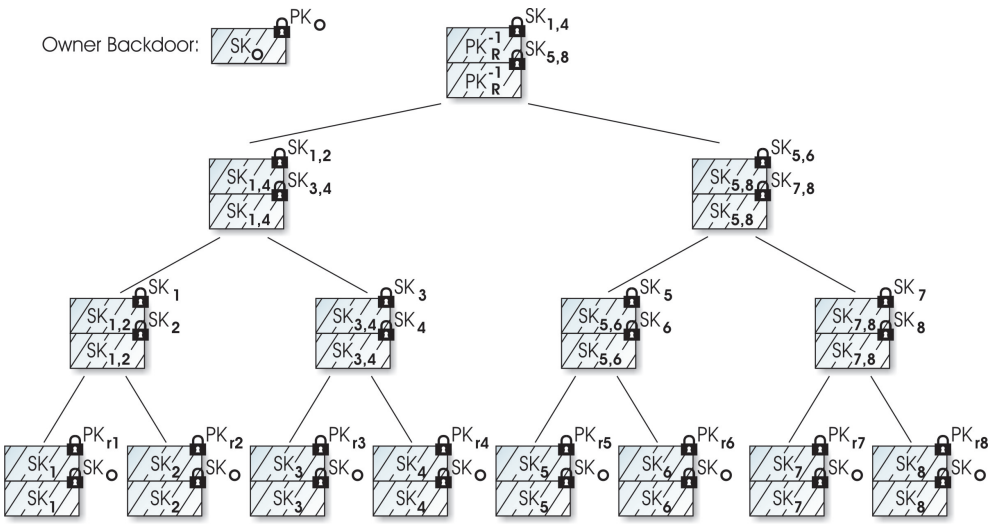

Figure 5.10: The key tree containing the encrypted keys for read access.

Figure 5.10 illustrates the final KTO containing the encrypted keys. The public/private key pair $\left[PK_R, PK_R^{-1}\right]$ for the readers is called *reader group key*.

During the gatekeeper initialization phase described in Section 5.2.4, the owner creates the key tree object containing the entries for all authorized readers. The KTO with identifier ID_{KTO} is filed to the object store. Similar to the access control list for writers, ID_{KTO} and vc_{KTO} need to be transmitted to the gatekeepers and added to their state.

5.6.2 Joining Reader

The key tree object KTO needs to be updated whenever a reader joins or leaves the group. For the joining reader, new keys need to be generated and the leaf has to be added to the tree. The rest of the tree should remain unaffected. All changes on the KTO need to be announced to the gatekeepers by sending the new identifier ID_{KTO} with the version vc_{ACL_W}.

An owner O can decide to add some reader r_k to the group of readers R by adding the reader appropriately to the KTO.

1. The owner O accesses his backdoor key SK_O by using his private key PK_O^{-1}.
2. For the joining reader r_k, O creates a fresh secret key SK_k.
3. SK_k is encrypted twice, once with the reader's public key PK_{r_k} and once with the backdoor key SK_O. The two encryptions form a new leaf that is added to the tree.
4. O decrypts the secret key of the parent node again using SK_O via the other child node and re-encrypts it with SK_k. The new encryption is added in the parent node of the new leaf. Then, the version counter vc_{KTO} is incremented by one.
5. O computes ID'_{KTO}, stores the KTO to the object store, announces $[ID'_{KTO}, vc'_{KTO}]$ to all gatekeepers and receives their confirmation $[ID'_{KTO}, vc'_{KTO} + 1]$.
6. O transmits $[PK_O, ID_{Group}, vc_\omega]$ to the new reader r_k such that the reader can retrieve the witness object Obj_{ω_G} by computing the SVID.

5.6.3 Leaving Reader

Which keys need to be replaced when a reader leaves?

When a reader r_k is removed by the owner O, all keys on the path from the leaf of r_k to the root need to be replaced. This approach has been adopted from VersaKey [39]. For each inner node on the path, a fresh secret key is created and encrypted with the two child keys (except for the leaving reader). For the root, a new public/private-key pair $\left[PK'_R, PK'^{-1}_R\right]$ is generated such that the leaving reader can no longer read the content of new blocks. To be able to update and re-encrypt the new keys, the owner makes use of his backdoor key SK_O. After having incremented the vc_{KTO} and the new identifier ID'_{KTO} has been computed, the KTO is stored in the object store. O then announces $[ID'_{KTO}, vc'_{KTO}]$ to all gatekeepers which send as confirmation $[ID'_{KTO}, vc'_{KTO} + 1]$.

Note that only the owner can remove readers. A reader that does not want to participate in a group has no means to remove itself. Hence, the system does not support voluntary leaves.

5.6.4 Reader Access Control Procedure

The access control scheme for readers is almost equivalent to the procedure described in Section 5.5.5. Instead of retrieving a list, the key tree object is fetched and decrypted appropriately.

5.7 Secret Sharing-Based Access Control for Readers

In the last two approaches, a reader could access the secret key that was used for block encryption by using the reader group private key. That private key was stored in a key list object KLO or in a key tree object KTO and could only be decrypted by authorized readers. Although that approach seems intuitive, it is not the only one which is applicable. The following proposal no longer depends on the KLO or KTO and secures the keys by sharing them among the gatekeepers. Although changes on the KTO are in logarithmic dimensions, frequent changes on large groups will finally sum up to a large number of de- and encryptions, including the generation of public/private key pairs. In contrast, the secret sharing-based approach does not require encryption of keys within a data structure, but aims at protecting the keys by sharing them among the gatekeepers.

5.7.1 Access Control Lists

There is an access control list ACL_W that contains all public keys of the authorized writers W. However, the schemes for list- and tree-based access control did not require an access control list for readers since access control was implicitly carried out by the data structures that contained the encrypted reader group key. As those data structures are obsolete due to the sharing approach, there is a need to explicitly list all authorized readers R in another access control list ACL_R. Optionally, the owner can add his own public key to ACL_R to be an authorized reader.

Why ACL_R?

During the initialization phase of the gatekeepers in Section 5.2.4, ID_{ACL_R} and vc_{ACL_R} need to be additionally passed to the gatekeepers. Those values have to be stored persistently as the state of a gatekeeper.

5.7.2 Joining and Leaving Reader

The procedure for adding or removing readers follows precisely the description for membership changes of writers in Section 5.4.2. Readers can be added to or removed from ACL_R at the owner's discretion. After the update of ACL_R, the owner sends the new $[ID'_{ACL_R}, vc'_{ACL_R}]$ to the gatekeepers which is confirmed with $[ID'_{ACL_R}, vc'_{ACL_R} + 1]$.

5.7.3 Reader Access Control Procedure

Unlike the list- and tree-based approaches where read access control was implicitly carried out by using a data structure for keys, it is essential to perform read access control explicitly on the gatekeepers. A reader r_j proceeds as follows:

1. r_j finds the latest witness object Obj_{ω_G} as described in Section 5.2.5.
2. r_j contacts at most $2t + 1$ gatekeepers requesting access to an object with identifier ID_{Obj} including an explicit challenge-response round.

3. r_j receives replies with $[Anchor_{Obj}, list^{enc}_{shares}]$ as outlined in Section 5.2.8.
4. The reader retrieves the header object H_{Obj} from the object store.
5. If a block i is about to be read, the reader extracts ID_{KSO}, retrieves KSO and compares the shares of the gatekeepers to their hash values in KSO. Invalid shares are discarded, whereas the others are used to reconstruct the block secret key.

Observe that since the challenge of the gatekeepers is vc_{Obj}, a replay attack is possible as long as vc_{Obj} does not change. Although the attacker cannot read the reply of the gatekeepers, it could have an impact on the availability of the system. Concretely, many reply requests would force the gatekeepers to encrypt all keys for the requesting reader every time which is computationally expensive. This is a typical denial-of-service attack. There are basically two ways to handle such attacks:

1. The challenge vc_{Obj} is replaced by a real nonce generated separately by each gatekeeper. This implies that a reader who issues a request has to generate and verify more signatures since the replies of the gatekeepers are all distinct.
2. The gatekeepers maintain a cache that is periodically updated which contains the last few replies. This also makes sense in case of a retransmission of messages that are lost on the way to the reader.

Since denial-of-service (DoS) attacks are a general problem of all kinds of systems, it seems reasonable to follow the original proposition. However, if it comes to a choice between one of the two mentioned DoS-mechanisms, the first one should be preferred. The reason is that the latter still allows a DoS attack which is directed against the bandwidth of the gatekeepers.

5.8 Read Semantics

Section 5.8 mentioned that reader might have problems to decrypt certain blocks of an object while others in the group do not have this problem. According to Section 2.8.2, an entity joining a group should immediately gain all permissions of that group. Section 2.8.4 suggests that some permissions can be delayed in distributed systems, such that a new member of a group does not necessarily have all permissions instantly. Nevertheless, it is assumed that being added to a group will *eventually* transmit all permissions to the new entity such that all members of the group have the same privileges.

For writers, this is indeed the case. The owner updates ACL_W and informs all gatekeepers about this change. The new writer will immediately gain the permissions that all other writers of the group have.

For readers, one has to distinguish once more between the list-/tree-based approaches and the sharing-based scheme. For the sharing-based model, readers always have the same permissions for the same reasons as writers: ACL_R is immediately updated and gatekeepers are notified about those changes. However, for the list- and tree/based approach, this does not hold. For brevity, the term *key object* (KO) is used instead of key list object (KLO) or key tree object (KTO).

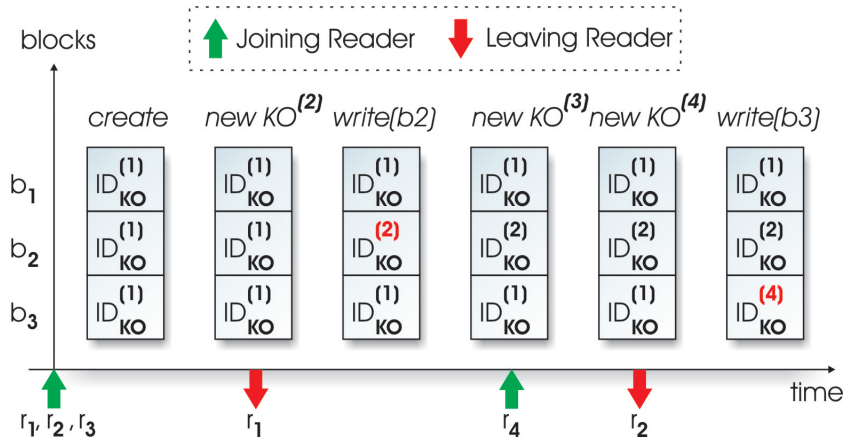

Figure 5.11: Changes on the reader group membership and creation of key objects as well as execution of write operations over time.

Readers of a group have different read semantics

The problem is illustrated in Figure 5.11. Initially, readers r_1, r_2 and r_3 join the group. Then, an object having three blocks is created and added to the group. All blocks use the same key object for encryption of the block keys, which is indicated by $ID_{KO}^{(1)}$ having the same version (namely 1). Reader r_1 is removed from the group. Consequently, a new key object with version 2 is created. The next write operation on block b_2 will use the new key object with $ID_{KO}^{(2)}$. The identifiers of the other blocks remain unchanged. Reader r_4 joins the group before r_2 is removed. Because

the owner adds r_4 to the key object, a new version is created. The same holds when r_2 is removed. Finally, a last write operation occurs on block b_3, updating the key object identifier appropriately. All blocks contain a different identifier for the used key object.

- What happens if r_3 attempts to read all three blocks b_1, b_2 and b_3?
- What happens if r_4 tries to read those blocks?

The behavior of a read access for r_3 differs from the one of r_4! Since r_3 is member of the group from the beginning, he can access all blocks since every key object contains an entry for that reader. r_4 can access b_3 without any problems since at the time when b_3 was written, the reader was already member of the group.

Accessing b_2 is complicated, but not impossible. Of course, if the reader uses $ID_{KO}^{(2)}$ as indicated by the object header, the read access will fail since version 2 of the key object does not contain an entry for r_4. Nevertheless, r_4 can use $ID_{KO}^{(3)}$ that he received when he was added to the group. Although the version of the key object is not correct, the reader group key is identical. Therefore, decryption of the block key will succeed.

Unfortunately, reader r_4 has no means to read b_1. The key object with version 1 was created long before r_4 joined the group. Since there was no write operation on that block, the key object was never updated. Hence, r_4 will fail when trying to decrypt the reader group key. As one assumes in general that entities do not maintain a state, even b_2 is unreadable. This leads us to the following proposition:

Proposition 5.8.1 *In list- and tree-based access control schemes, a reader can only read the content of blocks that were written after the reader joined the group.*

Proposition 5.8.1 does not directly conflict with the requirements on groups in Section 2.8.4. A reader will *eventually* be able to read all blocks since all blocks can be assumed to be written after some time since the reader has become a member of the group. Nevertheless, it is impossible to make any statement about how long it takes until a reader has read permissions on all blocks.

That disparity of read access control is very unfortunate. In many cases, being able to only read some small part of an object is useless. Therefore, it is desirable to allow a reader to access *all* blocks from the time on the reader is member of the group. The following sections will show how key lists and key tree objects have to be modified to allow access even to old key objects.

5.8.1 Complete Read Access with Key Lists

Linking key lists

The overall idea is to allow a future reader r_k to access a key list which was created long before the reader joined the group. It is not possible that there is an entry

which is encrypted with the reader's public key PK_{r_k}. But one can link the key lists such that being able to decrypt one key list allows to decrypt all key lists with older versions.

To make access to a key list as efficient as possible, the last column in Figure 5.9 which contained the encrypted reader group key now contains a reader secret key SK_R. That additional indirection allows a more efficient access to the previous key list. The reader group private key PK_R^{-1} is encrypted with SK_R just like SK'_R, which is the reader secret key from the *previous* key list. ID'_{KLO} is the identifier of the former key list. Figure 5.12 illustrates the new layout.

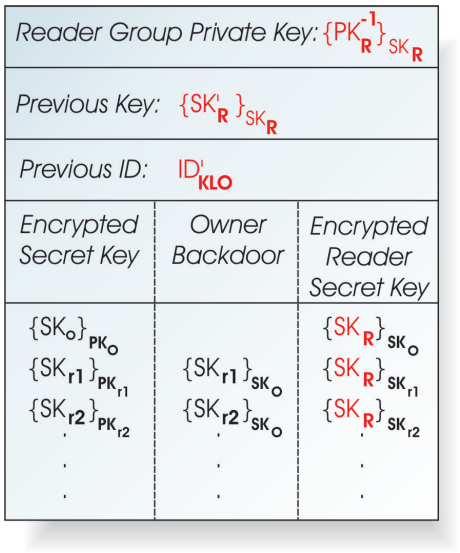

Figure 5.12: The key list extended with a reader secret key SK_R that can be used to access the reader group key PK_R^{-1} and the previous reader secret key SK'_R.

Since the secret key SK'_R for the previous key list object allows direct access to the previous reader group key $PK_R'^{-1}$, no public key decryption will be required when decrypting an older key list.

5.8.2 Complete Read Access with Key Trees

The original layout of the key tree object in Figure 5.10 remains mostly unchanged. Only the top of the tree has to be altered. The root of the tree is no longer the reader group private key, but the reader secret key SK_R. Like for the key list scheme, SK_R encrypts PK_R^{-1}. As mentioned in the last section, the additional indirection via SK_R allows a faster access to a previous key tree object by using symmetric-key cryptography. SK_R encrypts the secret key of the previous key tree object SK'_R. Figure 5.13 shows the new layout of a key tree object.

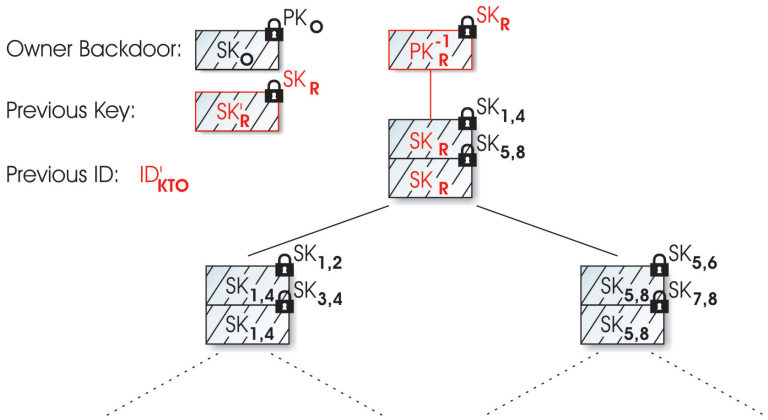

Figure 5.13: The key tree extended with a reader secret key SK_R that can be used to access the reader group key PK_R^{-1} and the previous reader secret key SK'_R.

Access to key tree in $O(1)$

Linking of key trees speeds up the process of decrypting the private key in an old key tree object. A reader has to access the latest tree and perform $O(\log m)$ decryptions, one of them being a public-key operation. But from that point on, all previous trees can be decrypted in $O(1)$ since SK'_R allows direct access to the root of the tree.

5.9 Protection Against Former Readers

The last three approaches for access control prevent readers from accessing data that was written *after* read permission has been revoked. This is equivalent to a scheme which encrypts the differences on an object since the reader left the group. While being sufficient for most scenarios, it might be intolerable for certain security sensitive applications. The goal of this section is to analyze mechanisms to prevent readers from retrieving and accessing old data after the loss of read privileges. Those readers are called *former readers*.

Encrypting differences

Former reader

The following considerations disregard recent technologies like trusted computing platforms and trusted platform modules (TPMs) [52]. Research in that area is still very young and a lot of concepts are not well understood. Although TPMs are already shipped with commercial computers, they are rarely used by software applications. From a security viewpoint, a trusted platform would make a lot of security related issues remarkably comfortable. As the ideas and implications of trusted computing are not well understood, such technologies are not considered further.

5.9.1 Perfect Solution and Feasibility

In a centralized system, revocation of read permission is fairly simple. The access control information that resides at the central authority is changed which prohibits all future unauthorized read attempts. In a distributed setting, revocation of read permissions is more complex. The previous sections explained that read access relies on the knowledge of a key that is only known to authorized readers. The goal is to prevent a former reader from reading the content of an object when either the object or the key is unknown to the reader[41].

The straightforward solution proceeds as follows: The owner determines all objects of the group which are affected by the leaving of a reader. He retrieves those objects and decrypts their content followed by a re-encryption of all objects with a fresh key that is unknown to the former reader. Then, he stores the newly encrypted objects in the object store, issues a delete request to all storage nodes that were in possession of the old objects and informs the gatekeepers with an update message. The former reader then has no means to read the content of the object because he does not know the key.

This simple solution has two major drawbacks regarding efficiency and security: Security of the approach depends on how the blocks are distributed among the storage nodes. Following the description of Chapter 4 where each storage node holds an encrypted block of an object, deletion of the old objects might fail because assumption 4.7.1 states that there is only at least one honest storage node. The

[41] Trivially, if the reader is in possession of both, the key to decrypt the object and the object itself, decryption can be carried out without interaction with the object store.

other storage nodes can be arbitrarily malicious and ignore the delete request which in turn allows former readers to access old objects. Efficiency is critical because the owner needs to retrieve all objects, de- and encrypt them and store them in the object store. For many or large objects, such an approach is infeasible albeit of the success of the delete operation.

Fractions

The layout of an object does not necessarily have to follow the description of Chapter 4. For example, the blocks could be further decomposed into *fractions* using erasure codes or other techniques. If the rate of an erasure code is r, then r fractions are required to reconstruct the full block. Assumption 4.7.1 needs then to be adjusted such that at least r storage nodes are honest and available, but there are always less than r corrupted storage nodes. Then, deletion succeeds since the honest storage nodes follow the request while the number of fractions of the corrupted ones is not sufficient to reconstruct the block.

Based on fractions, another idea is that the owner charges the storage nodes with re-encrypting the fractions on their own. However, this solution results in a lot of subsequent questions such as whether storage nodes should be able to read the content of the fractions, whether they have to decrypt the fraction and if so, how they know the decryption key. Moreover, erasure codes are in general not information-theoretically secure such as a secret sharing scheme. This means that even less than r fractions can reveal some information. One could also think of applying a secret sharing to each block instead of using erasure codes. Unfortunately, secret sharing schemes are not designed for sharing large secrets, for example mega- or gigabytes of data.

The following sections only adjust the object layout slightly. The goal is to omit delete operations and re-encryption of the data part of objects. Instead, the gatekeepers are used to prevent access of former readers in conjunction with the encryption of some part of the object's meta data.

5.9.2 Version Capabilities

In order to address the issue of a reader accessing an old version of an object, the model for the object representation needs to be slightly adjusted. Header objects contain a reference to the previous header similar to the linked key lists and key trees from Section 5.8. The new layout is shown in Figure 5.14.

It is the gatekeepers' responsibility to ensure that the chain of links is correctly constructed on each write operation. The consensus protocol cannot be used as previously described because a malicious king can force the acceptance of an arbitrary value by sending the same value to all players[42]. Since gatekeepers must ensure a correct linking, it is insufficient that honest gatekeepers update the anchor

[42] If assuming that the writer that initialized the write operation was malicious and distributed distinct values to each gatekeeper

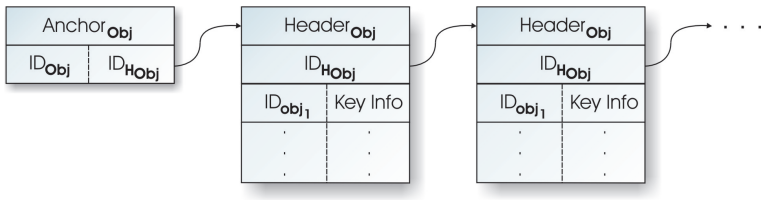

Figure 5.14: Versioning of objects through back references to headers of former object versions.

to some value, even if the value is equal for all gatekeepers[43]. For the layout in Figure 5.14, this means that the king must send the information of all written blocks to the other gatekeepers. Honest gatekeepers disregard the king's values if $ID_{H_{Obj}}$ is incorrect. The size of the king's message is in the order of the written blocks which is inefficient if many blocks are written.

Alternatively, one can decompose the header object into two parts, whereas the second part is called *reference object* Ref_{Obj}. On each write operation, a new H'_{Obj} is created that maintains an additional reference linking to the H_{Obj} of the previous write operation and one reference for Ref_{Obj}. The consensus protocol is performed on the content of the header object which are only two identifiers in the layout of Figure 5.15. Reference object

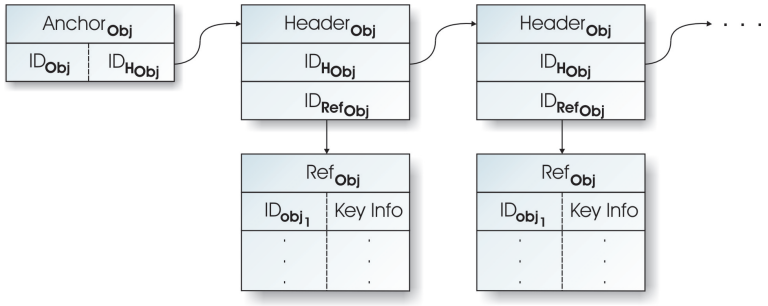

Figure 5.15: Alternative layout for objects with version where the block references are separated from the header object.

Note that the linking of headers only eases the way how old header objects are accessed. In principle, a reader could also cache old anchors locally or start an exhaustive brute-force search on all possible $ID_{H_{Obj}}$.

5.9.3 Former Reader in List- and Tree-Based Schemes

As explained previously, re-encryption of blocks itself is not feasible. Another intuitive way to address the problem of obsolete read permissions is to encrypt meta

[43] In the original object layout, this was not a problem since a reference to an inexistent header object is equal to overwriting all blocks with zeros.

data. The best way is to operate on the header object. To be able to decrypt the meta data, the header object itself needs a field for key information that has to be added to the figures 5.14 and 5.15. Depending on the content of the object header, one can draft two different schemes.

Full Header Re-Encryption

If taking the layout from Figure 5.14, there are two entries that are suitable to be re-encrypted: The identifiers of the blocks and the secret keys. In the following explanation, the secret keys are chosen to be re-encrypted. From a security point of view, encryption of the block identifiers is equivalent. For a single object Obj, the owner proceeds as follows:

1. The owner creates a new key object with reader group key $\left[PK'_R, PK'^{-1}_R\right]$ for the group that the object belongs to.
2. The owner retrieves the latest header object H_{Obj} and extracts all encrypted keys SK_i^{enc}.
3. He decrypts those keys by using the private key indicated by the ID_{KO} of each block entry[44] and re-encrypts all keys by using the new reader public key PK'_R.
4. If the system supports versioning of objects, the reference to the last header object is also encrypted using PK'_R.
5. The owner constructs the new header H'_{Obj} with the encrypted keys $SK_i'^{enc}$ and the information for the latest ID_{KO}, stores H'_{Obj} in the system and updates the anchor at all gatekeepers.

The leaving reader has no means to decrypt the secret keys of the object header as he has been removed from the key list or key tree respectively. Although blocks are not re-encrypted, the costs of the protocol are nevertheless remarkable. Let n be the number of objects in the group and m be the maximal number of blocks of an object. Then the costs are in the order of $O(n \times m)$ public-key operations.

Encryption of Link to Reference Object

The layout in Figure 5.15 allows to protect against former readers by only using two encryptions per object. Since the block references are separated from the header object, it suffices to only encrypt both references in the header object. A former reader can then neither follow $ID_{Ref_{Obj}}$ nor $ID_{H_{Obj}}$. The costs reduce to $O(n)$, where n is the number of objects.

[44] ID_{KO} stands for ID_{KLO} or ID_{KTO} depending whether the list- or tree-based approach is taken.

Analysis

Both schemes are equally secure, but differ in therms of complexity. Full header encryption is more expensive regarding cryptographic operations. But from a reader's point of view, the encryption of the header speeds up access to an object because all blocks are encrypted with the same public key. Although encryption of the reference object is more efficient, a reader needs one additional fetch operation for Ref_{Obj}. Moreover, if the following assumption does not hold, both schemes fail to protect against former readers:

Assumption 5.9.1 *Neither malicious gatekeepers nor malicious readers cache the identifiers or content of old object headers.*

5.9.4 Former Reader in Secret Sharing-Based Schemes

Readers always have to contact the gatekeepers to obtain the suitable secret key. This makes the solution trivial: Each gatekeeper always uses the latest ACL_R to determine whether a share should be revealed or not. This also holds if a reader tries to access a key for which he was once authorized. Hence, even in the presence of t malicious gatekeepers, the secret key will not be revealed and a read attempt is bound to fail. Note that this mechanism is independent of the object representation. There are no further costs to the protocol described in Section 5.7. The following assumption must hold:

Assumption 5.9.2 *The leaving reader does not have a local copy of one of the secret keys.*

The key could have been retrieved and cached when the reader was still authorized. The assumption is somewhat equivalent to assumption 5.9.1, but does not depend on the maliciousness of a single gatekeeper.

5.9.5 Conclusion

Since the secret sharing-based approach has no additional cost for protection against former readers, it is more efficient than the list- and tree-based approaches. Further, there are no additional changes needed regarding the structure of an object. Gatekeepers only have to verify the authorization of a reader by inspecting the latest ACL_R.

The list- and tree-based solutions require either to change the layout of a header object or to perform re-encryption of some part of all header objects of a group. For groups with many objects, partial encryption of the header object is too expensive. Another problem is the fact that a single corrupted gatekeeper can store information and hand it out to malicious readers when requested to bypass the protection measures. Finally, many object store systems constantly move objects between peers to ensure availability. One possibility is to follow references of objects starting at the anchor which is impossible if they are encrypted.

5.10 Self-Organizing Gatekeepers

The presented schemes implicitly assume that gatekeeper are long-living entities. The owner can change gatekeeper members at his discretion as outlined in Section 5.2.6. For a peer-to-peer network where nodes can join and leave arbitrarily, the assumption of long-living entities is crucial. In this chapter, the mechanisms that allow gatekeepers to organize themselves autonomously without the intervention of the owner are analyzed.

Recall that the public keys of all gatekeepers are stored in a so-called *witness object* Obj_{ω_G} as described in Section 5.2.4. Since the witness object is non-self-verifying, the owner has to sign the witness object and compute the identifier in a specific way (see Section 4.5). The owner could transfer the capability to sign the witness object to every gatekeeper by sending the signing key. But a corrupt gatekeeper could then sign and store any arbitrary witness object, which implies that the scheme could not even tolerate one malicious gatekeeper. Hence, the problem of creating a valid signature on the witness object without knowing the private key has to be solved. Gatekeepers are in possession of a *share* of the private key and partially sign the witness object. By executing a multi-party computation, the full signature can be reconstructed without ever computing the private key.

Mutual signatures

Re-sharing is hard

The second hard problem is to re-share the private key in a secure way such that every gatekeeper does not gain additional information that would help to infer the private key.

First, the procedure of generating a share of the private key is discussed, followed by the verification of the correctness of a partial signature and the reconstruction of the full signature. Then, the problem of creating new and correct shares of the private key to add new entities to the group of gatekeepers is seized. Two blackbox mechanisms are assumed to be in place:

1. One for detecting and deciding which gatekeepers are no longer available and need to be replaced.
2. One to determine and jointly deciding for new gatekeepers that replace the leaving ones.

The implementation of the two black box algorithms is far from trivial. Implicitly, gatekeepers have to perform consensus protocol runs to agree on joining or leaving members. Another crucial point is to avoid that the malicious entities convince the group to accept a new entity that is also controlled by the adversary such that assumption 4.7.3 fails. Those considerations lead to the the following assumption:

Assumption 5.10.1 *There exist two methods:*

1. *LeavingGatekeeper*() *determines which members of the gatekeeper group are removed from the witness object.*

2. *JoiningGatekeeper() determines which entities join the gatekeeper group and have to be added to the witness object.*

Both protocols return sets of entities $Set_{Leaving}$ *and* $Set_{Joining}$*, with* $|Set_{Leaving}| = |Set_{Joining}|$*. All honest gatekeepers that execute those methods will finally obtain the same set of leaving and joining members.*

The fact that those algorithms already achieve consensus regarding the membership of the gatekeeper group implies that subsequent protocols do not have to involve further consensus protocol runs.

5.10.1 Signature Key Sharing

For a new group of gatekeepers, the owner initially creates a public/private-key pair $\left[PK_G, PK_G^{-1}\right]$. Then, he creates n shares d_i of the private key PK_G^{-1} using Shamir's $(t+1, n)$-secret sharing scheme [18] and distributes share d_i to gatekeeper g_i. As the gatekeepers need to prove the correctness of their signature without revealing their share, the owner needs to extend the witness object. The following considerations closely follow the description of [40]. For each gatekeeper g_i with private key share d_i, the owner chooses a random number r_{g_i} and computes the witness ω_i as:

$$\omega_i = r_{g_i}^{d_i} \, (mod \, n')$$

where n' is the RSA modulus of the public key PK_G. Then, the entries of the witness object consist of:

$$ID_{g_i}, PK_{g_i}, r_{g_i}, \omega_i, \alpha_i$$

where ID_{g_i} is the identifier of gatekeeper g_i, PK_{g_i} is the public key and α_i is the x-coordinate of the polynomial share. The owner's public key PK_O needs to be replaced by the gatekeepers' public key PK_G in the witness object since PK_G is used for signing Obj_{ω_G}.

5.10.2 Signature Share Verification

To explain the procedure of shared signature verification, the notation common for RSA is introduced. Instead of writing PK_G for the gatekeepers' public key, one uses $[n', e]$, with both n' and e being publicly known. n' is known as the RSA modulus while e is the public exponent for encryption. The RSA modulus is constructed by choosing two prime numbers p and q, $p \neq q$, such that $n' = pq$. The Euler function $\varphi(n')$ is computed as $\varphi(n') = (p-1)(q-1)$. e is chosen such that $gcd(e, \varphi(n')) = 1$. Similarly, the private key is replaced by d, such that $d \times e = 1 \ (mod \, \varphi(n'))$ [27]. The following approach is adopted from [40].

Common RSA notation

An entity A that tries to verify a partial signature of g_i on a message[45] m proceeds as follows:

[45] For example, a message can be a hash value.

1. A sends the message m to be signed to a gatekeeper g_i.
2. g_i partially signs m and sends back $m_{sig,i} = m^{d_i} \, (mod \, n')$.
3. A chooses two random numbers $p, q \in \varphi(n)$ and creates a challenge c by computing: $c = m^p \times r_i^q \, (mod \, n')$, where r_i is g_i's random number from the witness object.
4. g_i signs the challenge c using his shared key: $c_{sig,i} = c^{d_i} \, (mod \, n')$.
5. A checks that $c_{sig,i} \equiv m_{sig,i}^p \times \omega_i^q \, (mod \, n')$.

The equation holds since:

$$c_{sig,i} = c^{d_i} = (m^p \times r_i^q)^{d_i} = (m^p)^{d_i} \times (r_i^q)^{d_i} = m^{pd_i} \times r_i^{qd_i} = (m^{d_i})^p \times (r_i^{d_i})^q = m_{sig,i}^p \times \omega_i^q \, (mod \, n')$$

Observe that this scheme is possibly vulnerable against chosen-plaintext or chosen-ciphertext attacks. The interested reader is referred to [59].

Why verifying the validity of partial signatures?

The reason why such a complicated verification of a partial signature is required is that one invalid partial signature suffices to corrupt the full signature reconstruction. Since the number of malicious gatekeepers is t, the scheme intends to be robust against t invalid signatures.

5.10.3 Signature Reconstruction

Assume that $t+1$ valid partially signed messages $[m_{sig,1}, m_{sig,2}, ..., m_{sig,t+1}]$ have been received. From the witness object Obj_{ω_G}, one can extract the appropriate polynomial evaluation points $\{\alpha_1, \alpha_2, ..., \alpha_{t+1}\}$. Then, the coefficients of the polynomial are:

$$k_i = \prod_{\substack{j=1 \\ j \neq i}}^{t+1} \frac{\alpha_j}{\alpha_j - \alpha_i}, \qquad \forall i \in [1, ..., t+1]$$

Finally, the full signature on m can be computed:

$$m_{sig} = \prod_{i=1}^{t+1} (m_{sig,i})^{k_i} \; (mod \, n')$$

The equation holds since:

$$m_{sig} = m^d \, (mod \, n') = m^{\sum_{i=1}^{t+1} k_i d_i} \, (mod \, n') = \prod_{i=1}^{t+1} \left(m^{d_i}\right)^{k_i} \; (mod \, n') = \prod_{i=1}^{t+1} (m_{sig,i})^{k_i} \; (mod \, n')$$

5.10.4 Gatekeeper Signature Agreement

In this section, a protocol that allows the gatekeepers to create a full signature on the witness object is outlined. For simplicity, the standard communication model with a complete (fully connected) synchronous network of pairwise authentic channels among the players is assumed. Assume that the gatekeepers have already decided which entities need to be removed and added to the witness object. The protocol for generating a full signature and storing the witness object is as follows:

Distributed signature generation

Protocol 5.10.1 GatekeeperSignatureAgreement(Obj_{ω_G})

1. *Determine* $Set_{Leaving} := LeavingGatekeeper()$ *and* $Set_{Joining} := JoiningGatekeeper()$
2. *Replace all entities in* Obj_{ω_G} *which are in* $Set_{Leaving}$ *by the entries in* $Set_{Joining}$ *to obtain* Obj'_{ω_G} *and set* $vc'_\omega = vc_\omega + 1$.
3. *Compute* $ID'_{\omega_G} = h\left(PK_G, vc'_\omega, ID_{Group}\right)$.
4. *SendToAll(*$[ID'_{\omega_G}, h(Obj'_{\omega_G}), Obj'^{sig,i}_{\omega_G}]$*).*
5. *Wait for* $> t$ *messages having an equal identifier and hash for the new witness object and which carry a correct partial signature.*
6. *Compute the full signature* $Obj'^{sig}_{\omega_G}$ *and file the witness object to the object store.*
7. *Initialize the joining entities* $g_j \in Set_{Joining}$ *with*

$$PK_G, ID_{Group}, vc'_\omega, Set_{Joining}$$

   and some keying material (see Section 5.10.5). The joining gatekeepers will have to start the synchronization protocol of Section 5.2.6.
8. *Return* vc'_ω.

The protocol will work if more than t honest gatekeepers are participating since the reconstruction of the full signature requires at least $t+1$ correctly signed messages. No consensus protocol run needs to be executed. This is possible since the protocols $LeavingGatekeeper()$ and $JoiningGatekeeper()$ implicitly achieve consensus.

5.10.5 Share-Share Generation and Distribution

If a gatekeeper g_i becomes permanently unavailable, the owner wants to provide means to other gatekeepers to transfer g_i's share d_i to a new gatekeeper g_{n+1}. The gatekeepers that participate in share distribution must not gain any information about d_i. Otherwise, if t malicious gatekeepers would obtain a new share, they

Why are share-shares needed?

could reconstruct the private key on their own.

The procedure is the same as described in Section 5.4.4 where a writer created shares and share-shares for his block encryption key. It is the owner who creates a KSO for his signature key and who sends the identifier ID_{KSO} to the gatekeepers during initialization. The KSO only changes if the owner creates a new public/private key for the gatekeepers and/or a new sharing of the private key. A version counter vc_{KSO} is added to the KSO such that gatekeepers only accept updates which have a higher counter than the previous one. Note that the owner does not need to sign the KSO because the object is self-verifying and all honest gatekeepers hold the same ID_{KSO} similar to the anchor objects. Joining gatekeepers can use the KSO to verify the authenticity of the retrieved share-shares before reconstructing the share of the signature key.

5.10.6 Ownership Transfer

As explained, witness objects are signed by a shared private key PK_G^{-1} of the gatekeepers. Section 5.10.1 already mentioned that PK in the witness object is set to PK_G instead of PK_O. This change has also an effect on the ownership transfer procedure of Section 5.3.3. O_{New} must replace the current gatekeepers' signing key by a new one which is unknown to O_{Old}. Otherwise, O_{Old} could create and sign arbitrary witness objects. O_{Old} must therefore set PK_{Next} to the new public key of the gatekeepers. However, that public key might be unknown to O_{Old}. In this case, PK_{Next} is set to $PK_{O_{New}}$. The first action of O_{New} is to create a new witness object that contains his newly created $PK_{G_{New}}$ and distribute the shares of $PK_{G_{New}}^{-1}$ to the gatekeepers. Besides, O_{New} can also choose to replace some gatekeepers.

5.10.7 Conclusion

The previous sections outlined how self-organizing gatekeepers would proceed to create a valid signature on the new witness object and how to distribute the shares to the joining entities. Unfortunately, the description of the share-share generation and distribution has some major drawbacks. The assumption on the number of malicious gatekeepers needs to be tightened:

Assumption 5.10.2 *From the set of current* ***and*** *former gatekeeper nodes, at most t are allowed to be malicious, where t is the threshold of a $(t+1, n)$-threshold secret sharing scheme. Conversely, at least $t+1$ of the current gatekeepers must be honest and available.*

Note that the scheme could even tolerate more than t malicious entities over the whole lifetime of the gatekeeper group if the joining and removing entities are both malicious. Then, the joining entity would receive a share which the adversary already has known. However, in the general case, it would also happen that an honest gatekeeper is replaced by a malicious one. Assumption 4.7.2 and 4.7.3

did not include the former gatekeepers. It was possible that a previously honest gatekeeper has been offline for a long time and thus was removed from the witness object. After that, the gatekeeper could have turned into a malicious entity without any effect on security. The situation is now different since gatekeepers need to mutually sign the witness object. If former gatekeepers keep their key share, they can help reconstruct the full signature and then sign the witness object on their own.

Other solution: re-sharing with 3D polynomial

One could demand that the *JoiningGatekeeper*() protocol would only select malicious gatekeepers with negligible probability. A better solution would be to create a re-sharing of the secret such that the new shares are statistically independent of the old ones. For further information, the description in [60] is recommended that uses a three-dimensional polynomial and can detect malicious input. The protocol is complex and includes broadcast primitives which results in high protocol costs (either for the number of rounds or the number of messages). Although the scheme in [60] is not discussed in this thesis, it is important to emphasize that such mechanisms exist.

An alternative to a re-sharing is a distributed RSA key generation protocol as presented in [34, 35].

Despite the schemes of [60, 34, 35], the sharing and consensus protocol can still only tolerate t malicious entities in the gatekeeper group at the time. Hence, the protocol *JoiningGatekeeper*() nevertheless needs to select malicious gatekeepers with only small probability.

5.11 Complexity Analysis

The schemes presented in Sections 5.2 and 5.4 - 5.7 have different complexity constraints, which are examined in this section. This complexity analysis aims at counting the number of cryptographic operations. The costs of a scheme vary for different types of entities. The complexity of an operation is thus analyzed separately for readers, writers, gatekeepers and the owner. For a compact representation of the tables, the following abbreviations are used:

PK_{Gen}**:** Public/private-key generation.

PK_{Enc}**:** Public-key encryption.

PK_{Dec}**:** Public-key decryption.

DS_{Gen}**:** Generation of a digital signature.

DS_{Ver}**:** Verification of a digital signature.

SK_{Gen}**:** Generation of a fresh secret key.

SK_{OP}**:** A secret-key operation, which is either a de- or encryption.

$Msgs$**:** Number of exchanged messages.

$Consensus$**:** Execution of an agreement protocol.

Public-key de- and encryption have been separated since the costs of practical algorithms vary for each of those functions. The same holds for signature generation and verification. In contrast, symmetric de- and encryption is assumed to be equally expensive which allows to accumulate both in a variable counting the number of secret-key operations (cf. Table 7.1).

The complexity of store and block retrieval operations are not listed in detail as it depends on the functionality and caching strategy of a concrete object store. Therefore, it only makes sense to count the number of store and retrieve operations. As the complexity analysis aims at determining the number of cryptographic operations instead of the number of requests to the object store, a short summary of those costs is given on the next few lines.

The gatekeepers need to fetch the most recent version of the access control list to authorize writers and - only for the sharing-based approach - also readers. Further, they have to retrieve the latest header object if it turns out that the write request was authorized and store the agreed header after the consensus protocol run. For frequent requests, it makes sense to cache this information to minimize latency and save bandwidth.

Readers and writers both have to find the witness object to communicate with the latest group of gatekeepers. In general, one cannot predict the number of

retrievals to get the latest version. However, for every retrieved witness object one digital signature needs to be verified. Let m be the average number of attempts to fetch the latest witness object. Then m digital signatures have to be verified. The owner does not need to search the witness object as long as gatekeepers are not self-verifying since he stores the latest version counter vc_{Obj_ω} locally which allows direct access to the latest witness object. Gatekeepers only have to retrieve the witness object on an update of the owner or during the initialization phase and verify its signature.

Finally, readers have to fetch the header object and the blocks which are intended to be read. For the list- and tree-based schemes, it is further required to retrieve the data structure that stores the encrypted keys. The number of retrieve operations depends on the number of blocks that were written with different reader group keys. If k is the number of blocks to be read, at most $O(k)$ fetch operations for the key data structures are necessary[46]. Since all key data structures as well as ACLs are self-verifying, no digital signatures need to be checked.

In the following considerations, n defines the overall number of gatekeepers while t is the number of corrupt gatekeepers, $t < n/4$. k denotes the number of blocks which are read or written. For analyzing the complexity of group membership changes, m stands for the number of entities that already belong to the group or which are added or removed, depending on the context of the analysis. In the average case, a constant number of entities c which is unrelated to m is assumed to be added. Note that the complexity analysis does not consider loss of packets or manipulation of packets which would force a retransmit.

The costs for the former readers has already been analyzed in section 5.9. Regarding self-organizing gatekeepers, the costs are not analyzed in detail. However, the complexity is in the order of $O(tn)$ because consensus protocol runs are required.

5.11.1 Gatekeeper Operations

This section analyzes the complexity of operations and protocols on gatekeepers such as the initialization of gatekeepers, the consensus protocol, version synchronization and the nomination of new gatekeepers.

Gatekeeper Initialization

Before gatekeepers become operational, the owner needs to initialize them. The initialization message consists of parameters relevant to the group. The costs for initialization are one signature generation and n signature verifications for the

[46] Under the assumption that between two used key data structures, there is only a constant number of unused data structures.

owner[47]. Gatekeepers, only need one signature generation and verification. The owner needs to transmit the initialization message to all gatekeepers. Table 5.1 gives an overview of the costs.

Entity	DS_{Gen}	DS_{Ver}	*Msgs*
Owner	$O(1)$	$O(n)$	$O(n)$
Gatekeeper	$O(1)$	$O(1)$	$O(1)$

Table 5.1: Costs for the initialization of gatekeepers for the owner and for a gatekeeper.

Since gatekeepers can verify whether the initialization message was a replay or not by inspecting the corresponding witness object, the number of sent messages n is minimal.

Consensus Protocol

Many operations require gatekeepers to communicate with each other and to decide for a common value. This is done by a consensus protocol. In appendix A, the king consensus protocol is explained. Since the message complexity is in the order of $O(tn^2)$, the costs for signature generation and verification must also be in $O(tn^2)$ for all gatekeepers and $O(tn)$ for a single gatekeeper as depicted in Table 5.2.

Entity	DS_{Gen}	DS_{Ver}	*Msgs*
Single gatekeeper	$O(tn)$	$O(tn)$	$O(tn)$
All gatekeepers	$O(tn^2)$	$O(tn^2)$	$O(tn^2)$

Table 5.2: Complexity of the consensus protocol based on king consensus.

Version Synchronization

It might happen that a gatekeeper is temporary offline which implies that some updates that happened on the other gatekeepers are lost. When the gatekeeper recovers and connects to the object store system, it needs to synchronize with the other gatekeepers as Section 5.2.6 suggests.

[47] As mentioned before, the creation and verification of the digital signature on the witness object are not counted.

Entity	DS_{Gen}	DS_{Ver}	*Msgs*
Existing gatekeeper	$O(1)$	$O(1)$	$O(1)$
Joining gatekeeper	$O(1)$	$O(n)$	$O(n)$

Table 5.3: Costs for version synchronization of gatekeepers for joining and existing gatekeepers in a list-/tree-based scheme..

List- and Tree-Based Synchronization Due to the gatekeeper counter $counter_{g_j}$, the number of digital signature generations can be reduced to a constant number because $counter_{g_j}$ serves as a common challenge for each gatekeeper. However, the joining gatekeeper needs to verify the received signatures. The version synchronization request is targeted at $2t + 1$ gatekeepers. Since the joining gatekeeper totally broadcasts $2t + 1 + n + t = n + 3t + 1$ messages, the number of signatures to be verified is also $n+3t+1$ which is in $O(n)$. The last t in the formula emerges because there might possibly be t additional polling messages to receive a proper state. Note that the number of digital signature generations is still constant, namely 4, because all polling messages contain the incremented counter. The existing gatekeepers have to generate at most 3 messages and 3 digital signatures. Additionally, at most 3 digital signatures have to be verified. If no polling messages are needed, the constant number reduces to 2.

Sharing-Based Synchronization In the sharing-based approach, an existing gatekeeper needs to symmetrically decrypt the shares of all objects, where u denotes the number of objects of the group. To decrypt SK_{Shares}, one public-key decryption is necessary and one public-key encryption must be performed to secure the shares for the transmission. A joining gatekeeper requires at most $2t+1$ public-key decryptions to access the shares and the precise number of messages is reduced to $n + 2t + 1$. Table 5.4 summarizes the costs. Note that the verification of the share-shares does not require cryptographic operations since only hash values have to be compared.

Entity	PK_{Enc}	PK_{Dec}	DS_{Gen}	DS_{Ver}	SK_{OP}	*Msgs*
Existing gatekeeper	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(u)$	$O(1)$
Joining gatekeeper	-	$O(t)$	$O(1)$	$O(n)$	-	$O(n)$

Table 5.4: Costs for version synchronization of gatekeepers for joining and existing gatekeepers in a sharing-based scheme..

Gatekeeper Nomination

The nomination of new gatekeepers according to Section 5.2.6 involves the owner and the new as well as the existing gatekeepers. Let m be the number of joining gatekeepers and n, as before, the number of existing gatekeepers. Then the complexity for nominating new gatekeepers can be determined as outlined in Table 5.5.

Entity	DS_{Gen}	DS_{Ver}	*Msgs*
Owner	$O(1)$	$O(n)$	$O(n)$
Existing gatekeeper	$O(1)$	$O(1)$	$O(1)$
Joining gatekeeper	$O(1)$	$O(1)$	$O(1)$

Table 5.5: Complexity of gatekeeper nominations.

To be more precise, the joining gatekeepers only have to verify 2 digital signatures of the owner: One for the initialization message and one for the state update message. The existing gatekeepers have to create maximally 4 digital signatures to respond to the owners state request and to confirm the update and to verify at most 4 signatures of the owner, 3 because the version synchronization protocol is carried out and 1 because of the update message of the owner. The precise number of messages for the owner is computed as follows: $n + 3t + 1$ messages are required for the synchronization protocol followed by m messages to initialize the joining gatekeepers and a broadcast of n messages for the witness update on all gatekeepers, which is in $O(n)$. The complexity for the sharing-based gatekeeper nomination is analogous.

In the following sections, the complexity for changing group membership of readers for each scheme is analyzed. Recall that it is the owner that has the sole competence to add or remove entities to/from one of his groups. The notification of all n gatekeepers always has a message complexity of $O(n)$, and requires only one signature to be generated and n to be verified for the update confirmation. The reason is that each object has a version counter. That version counter can implicitly be used as a challenge-response mechanism. Gatekeepers only accept updates where the new version is higher than the last one, thus injections of old update messages will not succeed. If not mentioned otherwise, those costs are silently assumed to be added.

5.11.2 Changing Group Membership Using List-Based Access Control

The list-based scheme from Section 5.5 contains an entry for each authorized reader. Each entry consists of three parts: The first entry is the reader session key encrypted with the reader's public key, the second entry is the owner's backdoor to the reader's session key and the last entry is the reader group private key encrypted with the

session key. Since the list is a trivial data structure, it holds that the best case equals the average case equals the worst case.

Adding Readers

The costs for adding readers is linear to the number of added readers m. Table 5.6 depicts the costs for adding readers.

Nr. Readers	PK_{Dec}	PK_{Enc}	SK_{Gen}	SK_{OP}
m	O(1)	$O(m)$	$O(m)$	$O(m)$

Table 5.6: Complexity for adding readers to a group using the list-based approach.

The number of public-key decryptions is 1 since the owner needs to access his secret key SK_O to be able to decrypt the reader group private key. Since for each reader, a new secret key needs to be generated and encrypted with its public-key, the number of secret-key generations and public-key encryptions is m. The number of secret-key operations is $2m$ because the owner needs to establish the backdoors.

Removing Readers

In contrast, removing readers is proportional to the number of remaining readers in the list. Given that the number of readers is m before the removal of entities, the costs of removing some number of entities is given as in Table 5.7.

Nr. Readers	PK_{Gen}	PK_{Dec}	PK_{Gen}	SK_{OP}
1	$O(1)$	$O(1)$	$O(1)$	$O(m)$
c	$O(1)$	$O(1)$	$O(1)$	$O(m)$
$m/2$	$O(1)$	$O(1)$	$O(1)$	$O(m)$

Table 5.7: Complexity for removing readers from a group using the list-based approach.

If only one reader is removed, exactly $2m - 1$ secret-key operations are required as the owner needs to decrypt $m - 1$ backdoor keys and encrypt the new reader group key m times for all remaining readers and once for himself. If the number of removed readers is $\frac{m}{2}$, then $m + 1$ secret-key operations must be carried out. For a constant number c, the number of secret-key operations is $2(m - c) + 1$. Hence, the number

of secret-key de- and encryptions is proportional to the number of remaining readers which is in $O(m)$.

5.11.3 Changing Group Membership Using Tree-Based Access Control

The analysis of the tree-based approach is not as trivial as for the list data structure. Since the tree can - in the worst case - be less efficient than the list, it is required to analyze best, average and worst case behavior of the tree separately. First, the complexity for adding a certain number of readers to the tree is considered. The costs all refer to the owner since the owner is the only entity that can change group membership. The second row uses for the number of readers a constant c. This is realistic since it is rarely the case that the number of added or removed readers is proportional to the number of existing readers, therefore c is independent of m.

Adding Readers

In general, the cryptographic operations needed when adding readers to the group is proportional to the number of added entities as shown in Table 5.8.

Nr. Readers	PK_{Dec}	PK_{Enc}	SK_{Gen}	SK_{OP}
m	$O(1)$	$O(m)$	$O(m)$	$O(m)$

Table 5.8: Complexity for adding readers to the key tree.

Tree requires m additional SK_{OP} and $2\times$ SK_{Gen} than list

The number of generated secret keys and the number of secret-key operations is in the order of $O(m)$, m being the number of added entities, since in general, a new subtree for the joining readers needs to be created. Consequently, given m readers, the space complexity of the tree is in $O(m)$. Because the focus lies on a practical approach, it is also worth to describe the precise number of secret-key generations and secret-key operations. If the total number of joining readers is given by m which form the leaves of the tree, then the total number of nodes including the root and the leaves is $2m - 1$, which is also the number of secret-key generations. Since every inner node is symmetrically encrypted twice and since the owner encrypts the the leaves with his own secret key SK_O, the total number of symmetric encryptions is $2(m - 1) + m + 1 = 3m - 1$. One has to be added because the owner needs to decrypt the parent node of the subtree. This means that the key tree needs m additional secret-key operations and twice as much secret-key generations compared to the list-based approach.

If the existing tree has some unused branches, there is no need for a full creation of a subtree. An algorithm that aims at creating a balanced tree would use those free branches to add new readers. If there are m readers to be added, exactly m secret keys would have to be created followed by $m \log m$ symmetric de- and $2m$

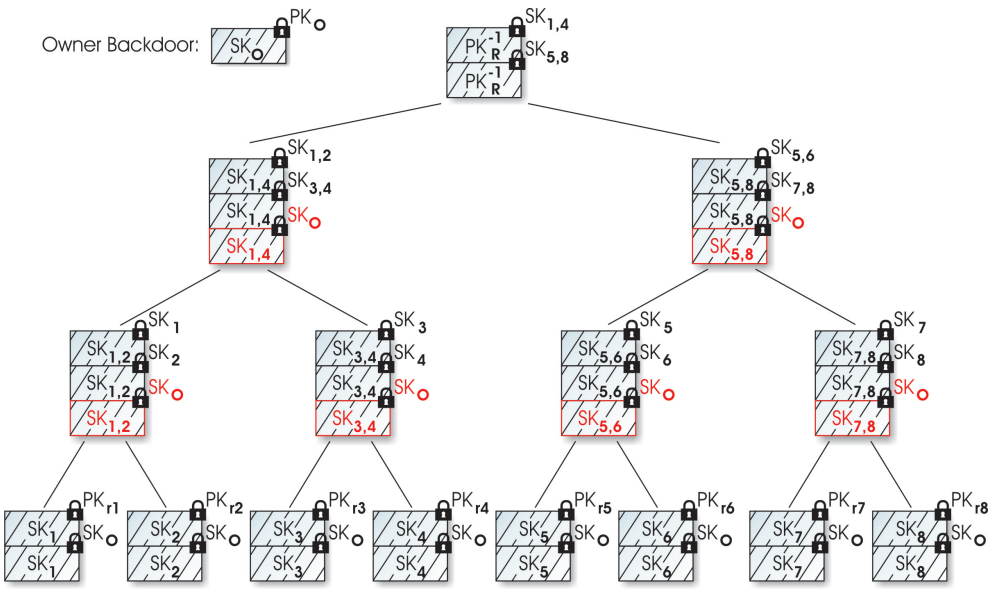

Figure 5.16: The key tree optimized for the number of computations by encrypting all inner nodes with the owner's secret key SK_O.

symmetric encryptions resulting in totally $m\,(\log m + 2)$ symmetric-key operations. The decryptions are necessary because the new leaves need to encrypt the key of the parent node which first has to be decrypted starting at a leaf node. Although the number of encryptions is equal to the list-based scheme, the decryption of the nodes makes the tree still behave worse than the list and worse than a full subtree creation as mentioned above. If the tree is modified as Figure 5.16 suggests where each inner node is additionally encrypted with the owner's secret key SK_O, the costs for decypting an inner node becomes constant instead of $\log m$, which reduces the total costs for decryption to m instead of $m \log m$. The total number of cryptographic operations sums up to $3m$, which equal to a full tree creation. The asymptotic costs remain the same as depicted in Table 5.8. Note that in the optimized tree in Figure 5.10, more space is needed, namely $6m - 3$ instead of $4m - 2$. This is significantly more than for the list which needs $3m + 2$ entries.

Tree needs $\frac{1}{3}$ more storage space than list

Removing Readers

Since the tree is worse than the list when considering joining readers, one would expect some advantages for the removal of readers in the average case. Let's start with a worst case scenario. In the worst case, every second reader is removed from the tree. Table 5.9 gives an overview of the worst case analysis.

Again, for practical reasons it is interesting to stick to a precise number of operations instead of using the O notation. For one reader, $\log\left(\frac{m}{2}\right)$ secret-key generations, $\frac{1}{2}\log m\,(\log m + 1)$ symmetric decryptions and $2\,(\log m - 1)$ symmetric encryptions are required. The tree already behaves significantly better than the list with

Tree is better when removing one reader

Nr. Readers	PK_{Dec}	PK_{Gen}	SK_{Gen}	SK_{OP}
1	$O(1)$	$O(1)$	$O(\log\, m)$	$O(\log^2 m)$
c	$O(1)$	$O(1)$	$O(m)$	$O(m)$
$m/2$	$O(1)$	$O(1)$	$O(m)$	$O(m)$

Table 5.9: Worst case complexity for removing readers from the key tree.

more than eight readers when only removing one entity. In the worst case where half of the readers is removed, all secret keys need to be replaced and encrypted, resulting in $m-2$ secret-key generations, $\frac{m}{2}$ symmetric de- and $\frac{3}{2}m-2$ symmetric encryptions, which yields in totally $2m-2$ symmetric operations. The list-based approach needed $m+1$ secret-key operations and no secret-key generations. Hence, when removing half of the readers, the list-based scheme is better than the tree-based approach. Note that the tree in Figure 5.16 even increases the number of encryptions such that the total number of symmetric encryptions is $2m$ and the total number of symmetric operations sums up to $\frac{5}{2}m$. Table 5.10 shows the costs for the best case scenario.

Tree requires $2\times$ more SK_{OP} than list

Nr. Readers	PK_{Dec}	PK_{Gen}	SK_{Gen}	SK_{OP}
1	$O(1)$	$O(1)$	$O(\log\, m)$	$O(\log^2 m)$
c	$O(1)$	$O(1)$	$O(\log\, m)$	$O(\log^2 m)$
$m/2$	$O(1)$	$O(1)$	-	$O(\log\, m)$

Table 5.10: Best case complexity for removing readers from the key tree.

Since the height of the tree is logarithmic in m and since removal of an entity is proportional to the height of the tree, all operations also become logarithmic. This is a significant improvement compared to the list which was in $O(m)$. Let's examine the values in detail. While the complexity for the removal of only one reader remains unchanged, the costs for removing a constant number and half of all readers has been drastically reduced. First, observe that the best case for the removal of readers is when all readers form a subtree such that it suffices to remove the branch to the subtree. Hence, removing half of all readers is equivalent to removing either the left or the right child of the root note. Since no reader knows one of the secret keys of the opposite subtree, there is no need to generate new secret keys. The number of secret-key operations is in $O(\log\, m)$ since the owner needs to decrypt a path from a leaf to the root to re-encrypt the new reader group

Best case for tree is logarithmic

key at the root. If c readers are removed, then there is a need to generate $\log(\frac{m}{c})$ secret keys because the subtree with the c leaving readers can be removed. The number of symmetric operations consists of $\frac{1}{2}(\log m\,(\log m + 1) - \log c\,(\log c + 1))$ symmetric de- and $2\log(\frac{m}{c})$ symmetric encryptions which is in $O(\log^2 m)$.

When using the extended tree from Figure 5.16, the best case complexity can be further reduced to a constant or logarithmic size as Table 5.11 indicates. The reason is that the owner has a backdoor at each node which makes logarithmic decryptions unnecessary.

Nr. Readers	PK_{Dec}	PK_{Gen}	SK_{Gen}	SK_{OP}
1	$O(1)$	$O(1)$	$O(\log\ m)$	O(log m)
c	$O(1)$	$O(1)$	$O(\log\ m)$	$O(\log\ m)$
$m/2$	$O(1)$	$O(1)$	-	$O(1)$

Table 5.11: Best case complexity for removing readers from the extended key tree of Figure 5.16.

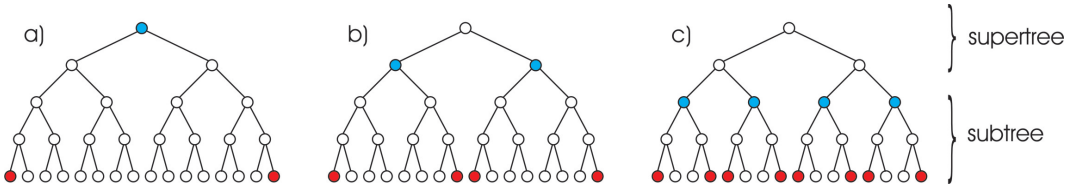

Figure 5.17: Three situations for removing a constant number of readers for $N = 16$: a) two removed readers, b) four removed readers, c) eight removed readers. The blue node denotes the point where the path of two neighboring removed readers intersect.

Since it can be assumed that the best case and the worst case are very rare, it is more interesting to reason about the average behavior of the tree if a constant number of readers c is removed. This means that, in the average case, every $(m/c)^{th}$ reader is removed which is equivalent to removing every $(\frac{m}{c})^{th}$ leaf of the tree. For simplicity, c is assumed to be a power of 2 and $c \neq m$. Since an upper bound in the average case is of interest, the leaves are chosen to be removed such that they have a maximal distance to each other with respect to the parent node where the paths of the leaf nodes meet. Figure 5.17 illustrates this situation for $m = 16$ and $c = 2$, 4 and 8. A blue node represents the first common parent node of two leaves. The number of subtrees is $\frac{c}{2}$ and the distance between two leaf nodes in a subtree is $2\frac{m}{c}$ including the two leaves that are removed. Let's first consider the complexity of the part of the tree which is above the blue nodes. This part is called *supertree*.

The supertree consists of $\frac{c}{2} - 1$ nodes and for each of them except for the root, a new secret key needs to be generated:

$$SK_{gen}^{super} = \frac{c}{2} - 2$$

The construction of the tree then requires that all those nodes including the root are symmetrically encrypted twice, once with the keys of each direct child node.

$$SK_{op}^{super} = c - 2$$

The analysis for the subtrees is slightly more complex. By choosing c to be a power of 2, the subtrees are symmetric and it suffices to compute the costs for only one subtree. The number of secret keys to generate is equal to the number of nodes on the path from the leaf to the blue parent node which only must be counted once. For one subtree, there are $2 \log \frac{m}{c} + 1$ secret key generations times the number of subtrees $\frac{c}{2}$:

$$SK_{gen}^{sub} = c \log \frac{m}{c} + \frac{c}{2}$$

Each of the new secret keys needs to be encrypted twice except for the nodes whose child was one of the removed nodes. Those nodes only need one encryption. Consequently, there are $4 \log \frac{m}{c}$ encryptions for one subtree multiplied by the number of subtrees $\frac{c}{2}$:

$$SK_{enc}^{sub} = 2c \log \frac{m}{c}$$

The number of decryptions is given by the number of nodes on a path to a new secret key which have to be decrypted. For one subtree, there are $\log\left(\frac{m}{c}\right)\left(\log\left(\frac{m}{c}\right) + 1\right)$ decryptions, again multiplied with the number of subtrees $\frac{c}{2}$:

$$SK_{dec}^{sub} = \frac{c}{2} \log \frac{m}{c} \left(\log \frac{m}{c} + 1\right)$$

Summing up the number of secret-key generations and symmetric cryptographic operations reveals

$$SK_{gen} = c \log \frac{m}{c} + c - 2 = O(\log m)$$
and
$$SK_{op} = 2c \log \frac{m}{c} + c - 2 + \frac{c}{2} \log \frac{m}{c} \left(\log \frac{m}{c} + 1\right) = O\left(\log^2 m\right)$$

Average case: logarithmic instead of linear

Sticking to the asymptotic notation, it turns out that the average case is optimal: $SK_{gen} = O(\log m)$ and $SK_{op} = O\left(\log^2 m\right)$. For the sake of completeness, the complexity for the optimized tree of Figure 5.16 is also given:

$$SK_{gen} = c \log \frac{m}{c} + c - 2 = O(\log m)$$
and
$$SK_{op} = 2c\left(2 \log \frac{m}{c} + 1\right) - 3 = O(\log m)$$ [48]

[48] The detailed costs are:

- $SK_{op}^{super} := \frac{3}{2}c - 4$
- $SK_{dec}^{sub} := c \log \frac{m}{c}$
- $SK_{enc}^{sub} := \frac{c}{2}\left(6 \log \frac{m}{c} + 1\right)$

Since the optimized tree can make use of the owner backdoor encryption on each node, the asymptotic runtime can be improved to $O(\log m)$ for both, the number of secret-key generations SK_{gen} and the number of symmetric de- and encryptions SK_{op}.

5.11.4 Changing Group Membership Using Sharing-Based Access Control

The sharing-based scheme is completely orthogonal to the list and tree approach. There is no data structure that contains keys for readers. Instead, the gatekeepers need to manage the shares of keys explicitly and only reveal them to authorized readers. Like the authorized writers, all authorized readers are itemized in an access control list ACL_R. Hence, there are no costs in terms of cryptographic operations since ACL_R is stored in plaintext.

5.11.5 Read and Write Access

After the analysis of the costs for changing the membership on all three schemes, the costs for read and write access are analyzed and the perspective of both gatekeepers and reading/writing entities is considered.

List-Based Access Costs

To make a statement about the complexity of read and write operations, k is defined as the number of blocks that are accessed, while n is the number of gatekeepers of the group.

Write Access The list- and tree-based schemes behave similarly when considering write operations. A writer needs to generate one fresh secret key to encrypt the data of the written blocks and then encrypts the secret key using the reader group public key PK_R. To get the latest version of the public key, the writer needs to contact the gatekeepers. After having carried out the encryption, the writer again contacts all gatekeepers passing all necessary information about the newly written blocks resulting in totally $n + 2t + 1$ messages. Table 5.12 gives an overview of the costs for a writer that has written k blocks.

PK_{Enc}	SK_{Gen}	SK_{OP}	DS_{Gen}	DS_{Ver}	$Msgs$
$O(1)$	$O(1)$	$O(k)$	$O(1)$	$O(t)$	$O(n)$

Table 5.12: Costs for a writer accessing k blocks of an object.

As Section 5.4.3 mentions, when contacting the gatekeepers for requesting the reader group public key, there is no need for a signature. Further, only $2t + 1$ gatekeepers are contacted and their signature is verified on their reply, which is in $O(t)$. Later, when the writer again contacts the gatekeepers to finalize the write

operation, he only needs to generate one digital signature and to send the same message to all n gatekeepers. This is possible since the common challenge from the gatekeepers is the latest version counter of the object.

The costs of the gatekeepers on a write access are dominated by costs of the consensus protocol listed in Table 5.2. Apart from the consensus protocol, all other operations are constant as shown in Table 5.13.

DS_{Gen}	DS_{Ver}	*Msgs*	*Consensus*
$O(1)$	$O(1)$	$O(1)$	$O(1)$

Table 5.13: Costs for a gatekeeper on a write request.

Totally, one gatekeeper needs $O(tn)$ signature generations, verifications and messages which results in a total complexity of $O(tn^2)$ for all gatekeepers.

Read Access Read access only requires a constant number of cryptographic operations. First, a reader needs to request the anchor from the gatekeepers. The message does not need to be signed and is sent to $2t + 1$ gatekeepers with the same challenge. The signatures of the received messages need then to be verified. After that, the key list object is retrieved and the key is decrypted. For the decryption of the reader group private key, one asymmetric and one symmetric decryption is required. After having decrypted the reader group private key, it can be used to decrypt the symmetric block key, which in turn allows to decrypt the content of the block. Since the reader is assumed to access k blocks and since it depends on how many different reader group keys were used, a best, average and worst case analysis is given as shown in Figure 5.14.

Case	PK_{Dec}	SK_{OP}	DS_{Ver}	*Msgs*
best	$O(1)$	$O(k)$	$O(t)$	$O(t)$
average	$O(k)$	$O(k)$	$O(t)$	$O(t)$
worst	$O(k)$	$O(k)$	$O(t)$	$O(t)$

Table 5.14: Costs for a reader accessing k blocks of an object using lists.

A closer look reveals significant differences for the number of digital signature verifications and public-key decryptions for the best and worst case. In the best case, one writer has written all blocks and encrypted the block key with the same reader group public key. A reader then has to perform two public-key decryptions, first to infer its secret key and then to decrypt the block key using the reader group private key. All k blocks need to be decrypted with the secret key in addition to

the symmetric decryption of the reader group key. In the worst case, all block keys are encrypted with different reader group public keys. The reader then needs $2k$ public-key and $2k$ symmetric decryptions. The number of messages is in $O(t)$ instead of $O(n)$ since the reader contacts only $2t + 1$ gatekeepers instead of n (cf. Section 5.5.5). The number of digital signature verifications directly follows from the number of exchanged messages.

Section 5.8.1 presented how to link key lists to each other to achieve the expected read semantics. Additionally, the linking of the data structures has the comfortable side-effect that read access becomes significantly cheaper. The reason is that the number of public-key decryptions can be reduced to k instead of $2k$ in the worst case because one can access previous versions by only using symmetric-key cryptography. Table 5.15 shows the costs for read access.

PK_{Dec}	SK_{OP}	DS_{Ver}	*Msgs*
$O(k)$	$O(k)$	$O(t)$	$O(t)$

Table 5.15: Costs for a reader accessing k blocks of an object using linked lists. Each list object contains a backward reference to the previous version of the list along with a key that allows direct access to the reader group private key which reduces the number of asymmetric decryptions to k instead of $2k$.

Gatekeepers are not involved in the read access control procedure except for the anchor request. The costs for a reply to an anchor message is one signature generation on one message.

Tree-Based Access Costs

In the last section, the analysis of the list-based approach revealed a complexity of $O(k)$ asymmetric and symmetric operations. This section examines the costs when the list is replaced by a tree.

Write Access The costs for write access in the tree-based approach are equal to the complexity for the list-based scheme as explained in the previous section.

Read Access For each tree that the reader accesses, one asymmetric decryption is necessary for the leaf of the tree. Conversely, one asymmetric decryption with the reader group key is needed when accessing a symmetric block key. Hence, the complexity regarding asymmetric cryptographic operations is the same for the list and tree approach. Table 5.16 gives an overview of the tree for a reader. Variable m denotes the number of readers which is a relevant factor since the number of symmetric operations depends on this value.

Case	PK_{Dec}	SK_{OP}	DS_{Ver}	*Msgs*
best	$O(1)$	$O(k + \log m)$	$O(t)$	$O(t)$
average	$O(k)$	$O(k \log m)$	$O(t)$	$O(t)$
worst	$O(k)$	$O(k \log m)$	$O(t)$	$O(t)$

Table 5.16: Costs for a reader accessing k blocks of an object using tree-based access control with m readers.

The analysis in Table 5.16 reveals that the asymptotic costs for symmetric-key operations are in $O(k \log m)$ which is more by a factor of $\log m$ than the list-based scheme. The important difference between a list and the tree is that the number of symmetric operations on the tree depends on the number of readers which is not the case for the list. The precise number of pubic-key decryptions for the best and worst case are 2 and $2k$ as for the list-based approach. For the number of secret-key operations, the best case requires $k + \log m$ while the worst case degenerates to $k(1 + \log m)$. At last, it is noteworthy that with linked key data structures like those in Section 5.8, access to old keys are all in $O(1)$ as for the list-based scheme. The number of secret-key operations is then $2k + \log m$, $\log m$ to access the first tree, k for the old trees and k to decrypt all blocks. It is even not required to communicate with the gatekeepers after having retrieved the identifier of the latest version. Table 5.17 shows the costs for the linked key tree data structure.

PK_{Dec}	SK_{OP}	DS_{Ver}	*Msgs*
$O(1)$	$O(k + \log m) \approx O(k)$	$O(t)$	$O(t)$

Table 5.17: Costs for a reader accessing k blocks of an object using tree-based access control with m readers. Each tree object contains a backward reference to the previous version of the tree along with a key that allows direct access to the reader group private key.

Linked tree and list are approximately equally expensive

What is worth mentioning in Table 5.17 is the approximation of $O(k+\log m) \approx O(k)$. This can be justified for two reasons: First, the logarithmic function grows very slowly. Secondly, the number of blocks to be read can be assumed to be significantly larger than the logarithm of m in the average case. Therefore, $O(k + \log m) \approx O(k)$ is a realistic approximation. The costs for the gatekeepers are the same as for the list-based approach.

Sharing-Based Access Costs

Unlike the other two schemes, the number of cryptographic operations in the sharing-based approach depends on the number of gatekeepers. There is no data structure that stores the keys for the readers explicitly. Instead, all keys need to be managed by the gatekeepers. Since a single gatekeeper cannot be trusted, the keys need to be shared among the gatekeepers with a $(t+1, n)$ secret sharing scheme.

Write Access As Table 5.18 illustrates, the number of public-key encryptions, digital signature generations and verifications is linear in the number of gatekeepers n which implies a linear growth in n compared to the other two approaches. The reason is that the shares are individual for each gatekeeper which requires separate encryption for each of the n shares. Moreover, each message needs to be signed separately.

PK_{Enc}	DS_{Gen}	DS_{Ver}	SK_{Gen}	SK_{OP}	$Msgs$
$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(k)$	$O(n)$

Table 5.18: Costs for a writer accessing k blocks of an object using secret sharing.

The complexity for the gatekeepers on a write access differs only by the fact that the encrypted shares need to be decrypted and re-encrypted with a symmetric key. The re-encryption is necessary in case that the nodes are compromised. Although an adversary has access to all shares, they are encrypted with a secret key which in turn is encrypted with the gatekeeper's public key. Without knowledge of the private key, the adversary cannot retrieve the content of the encrypted shares. Table 5.19 illustrates the complexity analysis for gatekeepers. Since it was decided to encrypt all shares of one object together, only two secret key operations are necessary - one to decrypt the shares and one to encrypt the new shares of the affected object.

PK_{Dec}	DS_{Gen}	DS_{Ver}	SK_{OP}	$Msgs$	$Consensus$
$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Table 5.19: Costs for a gatekeeper on a write request.

Read Access Read access control is mostly influenced by the fact that the signatures on the replies from the gatekeepers need to be verified. Since the shares are encrypted, it is inevitable to use public-key decryptions to access them. Table 5.20 summarizes the costs for a single reader.
Since the gatekeepers use as challenge the witness counter, the number of digital signature generations is 2. Due to the fact that the protocol requires an explicit

DS_{Gen}	PK_{Dec}	SK_{OP}	DS_{Ver}	$Msgs$
$O(1)$	$O(t)$	$O(k)$	$O(t)$	$O(t)$

Table 5.20: Costs for a reader accessing k blocks of an object using secret sharing.

challenge-response round, the reader needs to send $2(2t + 1) = 4t + 2$ messages to the gatekeepers. Since at most $2t + 1$ messages need to be decrypted, the number of public-key decryptions is bound by $O(t)$.

Gatekeepers only have to perform a constant number of operations to reply to a request of a reader as Table 5.21 illustrates. They require 2 digital signature generations and verifications and 2 messages as well as one symmetric decryption for the shares of the object.

PK_{Enc}	DS_{Gen}	DS_{Ver}	SK_{OP}	$Msgs$
$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Table 5.21: Costs for a gatekeeper on a read request.

The costs for all operations of the initial proposals in Section 5.5, 5.6 and 5.7 have been listed in detail. The next section concludes the results and compares the approaches to each other.

5.11.6 Conclusion

Based on the analysis of the previous sections, one can now compare the different approaches to each other. The operations on the gatekeepers are in the same order for all schemes. The interesting differences occur on group membership changes and read/write access attempts. The schemes will therefore be compared in those categories before an overall statement is given.

Changing Group Membership

Due to the different approaches, the costs for changing the membership depends on the concrete scheme. Further, one has to distinguish between adding and removing readers and writers and also consider best, worst and average case situation, whereas the average case is the most meaningful.

List/Tree: $O(m)$ SK_{Gen} & SK_{OP}

Adding Readers The number of secret-key operations for both, the list- and tree-based scheme is in $O(m)$. Although the asymptotic runtime is equal, the tree needs twice as much symmetric-key generations and additional m secret-key

operations compared to the list. Also regarding the space complexity, the list requires $25 - 50\%$ less entries. The reason is that the tree needs to be built up, which requires additional efforts. Less space for list

The sharing-based approach only requires operations in $O(1)$ because readers are maintained in an ACL which does not require cryptographic computations, but only the notifications of the gatekeepers. But since the notification also occurs for the list- and tree-approach, the sharing-based approach is optimal for adding readers. Sharing: $O(1)$

Adding Writers Adding writers is equally expensive for all three schemes. The access control lists for the writers is updated and the gatekeepers are informed, which is in $O(1)$.

Removing Readers In general, removing readers from a list is in $O(m)$ cryptographic operations because the entries for the remaining entities need to be re-encrypted. The costs in the tree-based scheme is different for best, worst and average case and the number of removed readers. If only one reader is removed, the complexity is in $O(\log m)$ secret-key generations and $O(\log^2 m)$ secret-key operations for the original tree and $O(\log m)$ for the extended tree. However, when removing c or a number proportional to m readers, the worst case reveals $O(m)$ secret-key generations and operations. Using precise numbers, the tree turns out to be worse by a factor of $2 - 2.5$ secret-key operations in the worst case. Further, the list does not require any secret-key generations. Conversely, the best case of the tree requires between zero and $O(\log m)$ secret key generations and between $O(\log m)$ and $O(\log^2 m)$ secret key operations. In a rare case, the extended tree even comes by with $O(1)$ secret-key operations and zero secret-key generations. But as stated in the text, the best and worst cases can be assumed to occur rarely. While the average case for the list remains unchanged, the tree can be expected to use $O(\log m)$ secret-key operations and $O(\log^2 m)$ secret-key generations for the normal tree and $O(\log m)$ operations for the extended tree. List: $O(m)$ SK_{OP} Tree: $O(\log m)$ SK_{OP} and $O(\log^2 m)$ SK_{Gen}

As for adding readers, the costs for removing an arbitrary number of entities can be done in $O(1)$ cryptographic operations when using the secret-sharing scheme. Sharing: $O(1)$

Removing Writers Since all schemes use an access control list for writers, the removal of any number of writers is equally expensive for all schemes, namely $O(1)$.

Read / Write Access

As for the analysis of group membership changes, the costs for read and write access mostly depend on the used data structures.

Read Access Read access in the list-based model requires public-key decryptions and secret-key operations in the order of blocks k, which is $O(k)$. The tree requires more computations since it has $O(k \log m)$ secret-key computations. The reason List: $O(k)$ SK_{OP}

for this discrepancy is, that access to the tree depends on the number of readers m within the tree while the list only has a constant number of cryptographic operations. This analysis induces to assume that the tree is significantly worse than the list for read access. However, this is only partially true. When considering the mechanisms of Section 5.8 where a linking of the data structures was shown, only the access to the latest tree is in $O(\log m)$ as opposed to $O(1)$ in the list, which is practically feasible even for a large number of readers m. After the decryption of the latest tree, access to all old versions can be done in $O(1)$. Hence, if k is very large, the first $\log m$ decryptions do not preponderate which results in $O(k + \log m) \approx O(k)$.

Tree[49]: $O(k + \log m) \approx O(k)$ SK_{OP}

Read access in the sharing-based scheme is more expensive regarding the number of asymmetric operations since it requires $O(t)$ public key decryptions and $O(t)$ digital signature verifications.

Sharing: $O(t)$ PK_{Dec} & DS_{Ver}

Write Access The costs for write access is equal for the list- and tree-based scheme. In contrast, the sharing-based scheme has a higher complexity for the number of public-key encryptions, digital signature generation and verification because the shares need to be created and signed separately for the transmission to each gatekeeper. The number of secret-key operations and the number of messages is equal for all schemes. As opposed to the list- and tree-scheme, the gatekeepers in the sharing approach additionally have to perform $O(1)$ secret-key operations for the shares of an object while writers have to asymmetrically encrypt the shares and share-shares.

Overall Comparison

Since the sharing-based approach is the most extraordinary one, that scheme is first highlighted:

If it is necessary that changes on the membership of the group either for readers or writers must be efficient, it best to decide for the sharing-based scheme. Unlike the other approaches, membership changes are all in $O(1)$, which is optimal. However, both read and write access are more expensive than in the list and tree approaches regarding asymmetric-key operations. The reason is, that the number of digital signature generations and verifications as well as the number of public-key de- and encryptions are in the order of the number of gatekeepers. A read attempt requires digital signature verifications in the order of $O(t)$ where t is the number of Byzantine gatekeepers. The same holds for a write process where even all gatekeepers n are invoked instead of only $2t + 1$. Another issue to consider when using secret sharing is that the state of the gatekeepers heavily grows. Moreover, if for some reason, all gatekeepers leave the network, the owner has no means to recover his objects since the keys are only known to the gatekeepers. One could require that the gatekeepers also encrypt the key shares for the owner and store them in the object store, but

Sharing: cheap group updates but expensive read/write access

[49] Only when using linked key trees. Otherwise, the costs are $O(k \log m)$

the owner has then the problem of finding the latest object containing the shares[50]. The synchronization of a gatekeeper is more expensive since the key shares of each object's blocks need to be reconstructed and verified which is inefficient if the state of a gatekeeper is large (i.e. if there are many and large objects). Regarding former readers, the sharing-based scheme is more robust since access control is explicitly performed for readers. However, the synchronization protocol for a system that maintains object versions is impractical since the number of verifications on the shares is very large.

Coming back to the list- and tree-based schemes: According to the analysis, a key list is optimized for frequently adding readers while the tree has better performance on removal of readers in the average case. While asymptotically, adding readers to a list is equally expensive as adding them to a tree, the asymptotic behavior of the list on removal is $O(m)$ while the tree only has $O(\log m)$ or $O(\log^2 m)$ in the average case, depending on the kind of tree that is used. This is a significant improvement. Access to the list can be done in a constant number of computations whereas the number of steps in the tree are logarithmic to the number of readers of the group. The tree is thus worse by a factor of $O(\log m)$ compared to the list. However, since Section 5.8 states that it is more natural to establish backward links between the versions of the data structures regarding read semantics, the additional costs of $\log m$ becomes negligible when considering the costs of $O(1)$ for accessing an old version of the tree compared to the number of blocks k to be read. The costs are then in $O(k+\log m) \approx O(k)$ which practical even for a large number of readers m.

Reader removal: tree is logarithmic, list is linear in average case

Concluding, the tree is the preferred data structure since it has a logarithmic behavior in the average case while the list is linear. Only when adding readers, the tree's performance is worse than for the simple list because the inner nodes of the tree need to be created. The sharing-based approach should only be used in systems that require fast and frequent group updates.

[50] Unless the objects containing the keys use a SVID.

6 Implementation

This chapter describes a prototypical implementation of the tree-based access control scheme that was explained in the previous chapter. The implementation intends to be a proof-of-concept which is used to measure how the mechanisms scale in a real-world setting with all its difficulties. The underlying object store is Celeste [61] which is written in Java 5.0. Celeste is being developed by Sun Mircosystems Laboratories in partial cooperation with ETH Zurich. Before explaining some delicate aspects of the implementation, a short overview of Celeste is given.

6.1 Celeste

Celeste is a self-managing, secure, massively distributed, random-access, read/write data store with high availability properties. The most important aspects regarding security were already mentioned in Section 3.7. The actual implementation did not feature any access control mechanisms.

6.1.1 Celeste Layers

The three basic Celeste layers

Three basic layers of Celeste can be identified: The Celeste file system layer, the object store layer and the overlay network layer (cf. Figure 6.1). The file system layer exposes a file-like interface to the user and abstracts the mapping to objects. The object store layer can functionally be divided into two sub-layers: A Celeste part and a Distributed Object Location and Retrieval (DOLR) part. The Celeste part can be seen as the client side of the system. It offers interfaces to create, delete

DOLR

and modify objects. The underlying DOLR layer implements an overlay network on top of the basic IP network, and constructs its own naming and routing scheme with a Distributed Hash Table (DHT) [67, 68, 69, 70, 71]. The DOLR layer is responsible for publishing and unpublishing objects, GUIDs of objects and nodes as well as invoking and retrieving them. Moreover, the DOLR layer fulfills persistency and refreshing tasks, which means that objects are replicated to guarantee maximal availability.

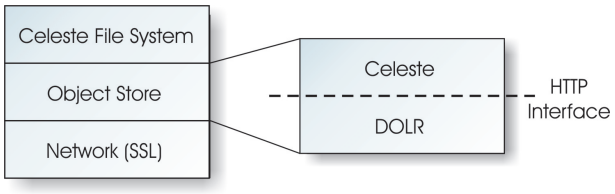

Figure 6.1: The three basic layers of Celeste. The object store layer can be further subdivided into two parts.

The Celeste part and the DOLR part are separated by a HTTP interface. There are no direct method calls from a class in the Celeste part to one in the DOLR part. Instead, a HTTP connection is established to exchange data and information. The idea is that the Celeste file system and Celeste part of the object store layer can be used on a thin client while computationally expensive operations can be shifted to a central server. The connection is not secured for the sake of simplicity. Note that a further reason for segregation between the Celeste and DOLR part is that private user data (i.e. secrets, unencrypted content) is never exposed to the Celeste node, or the network as a whole. The Celeste part signs and possibly[51] encrypts the content of the written object before it is passed to the HTTP interface.

Why HTTP interface?

Communication between DOLR nodes is protected using SSL [26] to achieve authentic and confidential communication channels.

For the implementation of the access control mechanism, the focus is only on the object store layer.

Celeste Objects

The organization of Celeste objects is similar to the description in Section 4.3. All objects are uniquely identified by self-verifying GUIDs. Every write operation creates a new object. It is not possible to effectively overwrite existing blocks. Celeste maintains versions of objects. The entry point into the version chain is called the *active object*, which has an AGUID.

AObject

[51] The original implementation only signed the object's content. The implementation for access control additionally adds encryption.

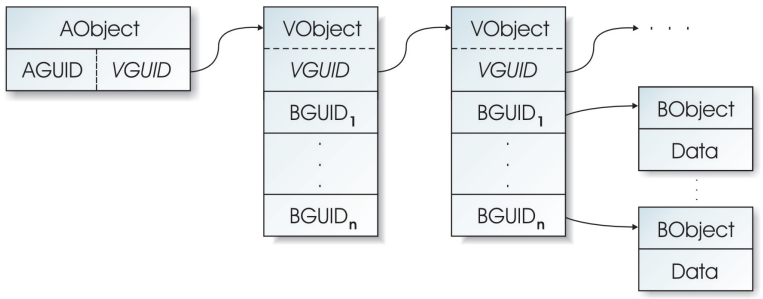

Figure 6.2: The layout of Celeste objects: AObject, VObject and BObject (fragments were omitted for simplicity).

The AGUID is the only identifier which is not self-verifying. It can be seen as the name of an object which never changes. The active object contains the VGUID of the latest *version object*. A version object consists of a list of BGUIDs to the *block objects* and the VGUID of the previous version. By storing the VGUID of the previous version in the VObject, Celeste is able to traverse the list of versions starting at the AObject. The BObjects are further decomposed into so-called *fragments*. Figure 6.2 shows a graphical illustration of the Celeste object as described. Note that this is just one possibility to implement objects. Other approaches might choose a different layout.

VObject
BObject
Fragment

Celeste Serializer

One of the most important components in Celeste is the *Serializer* [72]. The task of the Serializer is to maintain the ordering of VObjects. The Serializer is a replicated component that can tolerate node failures. It can also recover from a network partition. When an object is created, it is associated with a Serializer by storing the Serializer's GUID with the object. All read and write attempts are directed to the Serializer which then ensures that the VObject's reference to the previous version is correct and also updates the AObject's reference. On a network partition, the object's Serializer in each partition can still operate and accept write operations. This leads to different generations of the same object which are be merged by branching histories in the VObjects when the network reconnects.

Serializer

Profiles and Nodes

Each user who joins the Celeste system creates a new profile and files it to the object store. The profile contains among other information: The name of the user, a public key, a signed certificate and a private key[52]. Since the profile is publicly stored, the private key must be protected. This is done by locking the profile with

Profiles

[52]The presence of the private key in the profile is optional. It could as well reside on a smart card or be provided by other external media. Storing it in the profile is a deliberate security-convenience tradeoff.

a passphrase. When the profile is retrieved, the passphrase unlocks the profile. Profiles are the root of all user-related identification and authentication mechanisms and are handled at the Celeste part of the object store layer.

Nodes

There is also the concept of a *node*. Each node owns a public/private-key pair and a self-signed certificate. Since owners use SSL for the communication, they must know with which node they are communicating. The problem of relating a public key to a node has been solved by setting the node's GUID to the hash of the public key. This allows authentic and secure channels between any node. Observe that nodes are orthogonal to user profiles since they reside at a lower abstraction level, namely at the DOLR part of the object store. Nodes transmit messages to other nodes via a SSL connection. The addressing uses the GUID of the target node. The translation of a GUID to a real network address is done by the DOLR layer.

GUID is a hash of the public key

6.2 Implementation

This section explains important parts of the access control system implementation in Celeste. The implementation was separated into two stages: First, the necessary data structures were designed and implemented. The data structures include the access control list, the witness object, the gatekeepers' storage for managing groups, and the key tree object. To enhance the author's personal programming skills, the data structures were implemented following the *extreme programming* paradigm [74], where a unit test is created before the corresponding functionalities exist. Second, the data structures were embedded into Celeste whereas some modifications to the Celeste source code were necessary. The implementation was carried out within less than three weeks. Unit tests [75] turned out to be particularly useful for the development cycle.

Extreme programming

The following sections aim at giving an idea of the efforts that were done in order to implement the most important concepts of Chapter 5 related to the tree-based approach. Thus the text does not include code fragments or detailed information, but provides a high level and compact view of the implementation. The CD that is provided with the printout of the thesis includes a detailed Javadoc documentation.

6.2.1 Overview

The implementation affected two of the sub-layers mentioned in Figure 6.1: The Celeste part and the DOLR part of the object store layer. Reading and writing including de- and encryption, signature generation and verification all happen at the level of the Celeste part. The reason is that signing and public-key decryption requires unlocking of a profile which depends on a PIN. For security reasons, the PIN is never passed across the HTTP connection. Therefore, all operations related to group initialization, membership changes of gatekeepers, object creation, object addition or removal to/from a group, addition or removal of readers or writers

to/from a group, write and read operations are bound to the object store layer.

Gatekeepers, on the other hand, are independent of profiles. They are implemented at the DOLR layer. Hence, every node connected to the system can serve as a gatekeeper. The identification of gatekeepers can be done by using the node's key material as mentioned in Section 6.1.1. Due to the extension for access control, the implementation of a node was modified such that requests can be sent to nodes to retrieve their certificate. The owner uses this functionality when initializing a group.

Storage Client The most important functionalities of the Celeste layer are encapsulated in a storage client class which implements the Celeste interface. The interface offers method signatures to read, write, fetch, delete and create objects and profiles. The functionalities for access control reside in a new class which also implements the Celeste interface to be backward compatible. However, most operations require additional parameters[53] for access control which the interface does not offer. To keep the interface stable, the extended storage client offers additional methods to set those parameters either via a new method or using the constructor. The Javadoc documentation states explicitly that certain calls to other methods are required in order to carry out access control properly. That design decision allows to smoothly replace the old storage client. Since the extended storage client is not yet fully implemented, it uses the original storage client for certain operations (cf. Figure 6.3).

Backward compatibility through stable interfaces

The functionalities for distributed access control in Chapter 5 are complex. Furthermore, since the different functions can be associated with distinct types of entities, the implementation encapsulates those functionalities in separate classes. Those classes are then invoked from the extended storage client which initialized them and holds a reference. Such a design strictly follows the object-oriented principles of data encapsulation and information hiding. Figure 6.4 illustrates the relationship between the extended storage client and the owner, reader and writer classes. Since the entities share some common functionalities, those functionalities are encapsulated into a so-called system entity class to avoid code duplication.

One class for each entity type

As this chapter only gives a brief overview, Figure 6.4 does not list all functions and fields except for the most important ones.

Gatekeeper Gatekeepers are situated at the level of *DOLR nodes*. A DOLR node basically allows to transmit a DOLR message to some other node and to receive messages. Incoming messages are dispatched by an application framework to a DOLR application that previously registered for the corresponding type of message. The gatekeepers were implemented as one of these DOLR applications.

DOLR nodes

Gatekeepers as DOLR application

The *handleMessage()* method in Figure 6.5 is not just one method, but a set of meth-

[53] For example the name of the group.

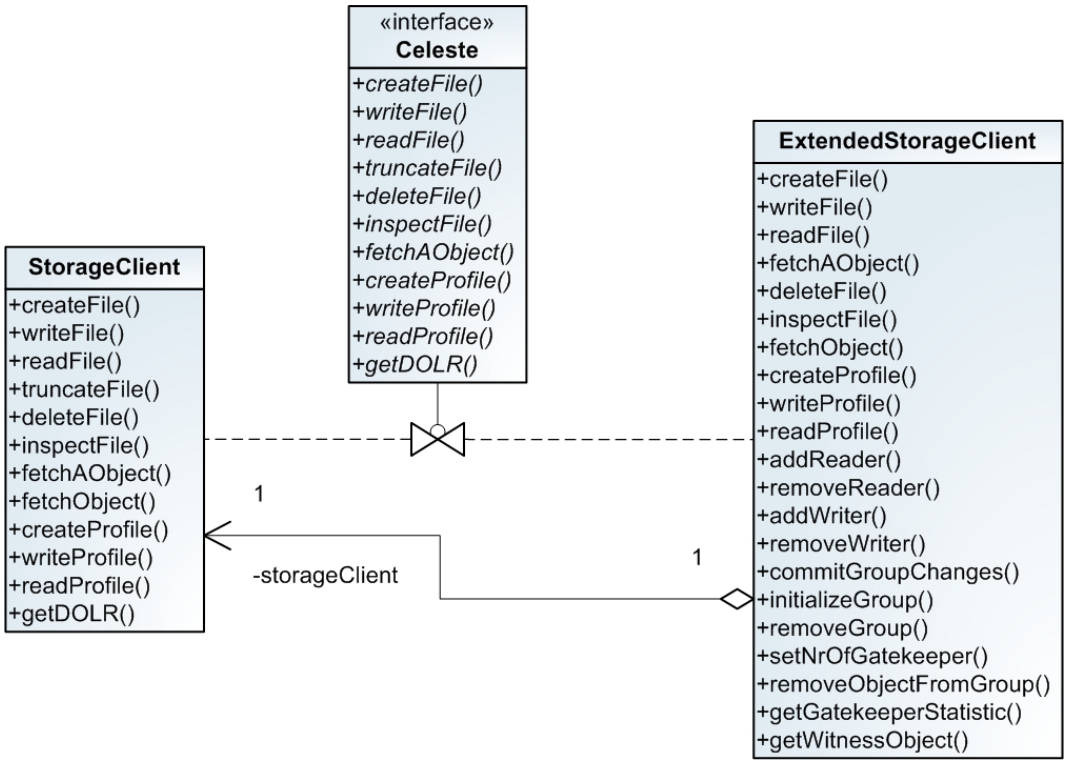

Figure 6.3: UML diagram of the Celeste interface which is extended by the original storage client. The extended storage client offers access control capabilities.

ods. Each of these methods implements the function needed for a certain type of message such as group initialization, read or write requests. The content of a DOLR message first needs to be parsed before it can be dispatched by the *routeToNode()* method. In some cases, it is necessary to verify a signature on a DOLR message. On a write operation, the consensus manager is invoked to get an instance of a consensus module which then communicates with the other gatekeepers of the group.

Both reader and owner have to deal with the key tree object, the central data structure of this implementation. The key tree is explained in the following section. Afterwards, the design of the gatekeeper storage is described.

6.2.2 Key Tree Object

The complexity analysis revealed a promising behavior for the key tree in the average case. Therefore, the tree described in Section 5.6 and depicted in Figure 5.10 was implemented. Measurements aim at verifying the claim that the tree is efficient in the average case. Although a binary tree is a trivial data structure, the key tree is complex due to the distinct content of different types of nodes. The implementation intends to be both well-designed and efficient. The complete

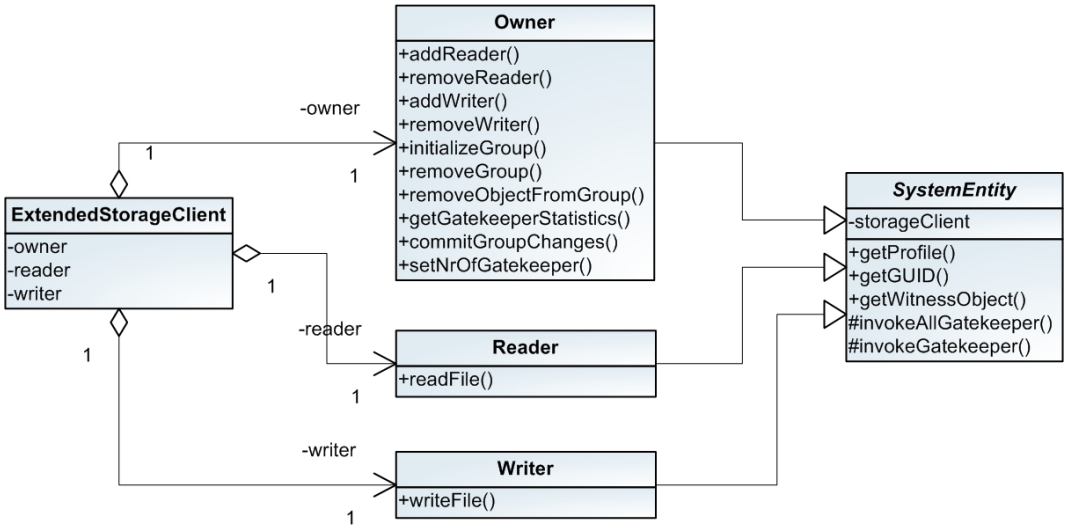

Figure 6.4: The extended storage client delegates calls to specialized classes implementing the functionalities of the owner, reader and writer. The system entity encapsulates functions shared by all three entities.

realization consists of six classes, as shown in Figure 6.6.

The key tree object offers a high-level interface to the user of the tree and allows to add or remove readers and to build, respectively encrypt, the tree. Changes on the key tree are not immediately carried out because they require cryptographic operations. Instead, when adding or removing entities, those entities are stored in a list. The lists are processed when the *generateTree()* method is invoked by the owner. There are other additional fields to store the version counter of the tree and the name of the group that the tree belongs to. A detail mentioned in Section 5.6.1 is the lookup table, which is also located in the key tree object. The lookup table is implemented as an ordinary hash table mapping each reader's profile GUID to the matching leaf node. When a reader accesses the tree using the *getPrivateKey()* method passing his GUID, public key and private key[54], the lookup table is consulted to find the correct leaf. An exception is thrown in case that the reader's GUID is not contained in the table. The key tree object is extended by a linked tree, which stores a reference to the previous tree to achieve the desired read semantics discussed in Section 5.8.

Batch mode

Lookup table

The remaining four classes represent the different types of nodes of the tree. There exists an abstract node class which encapsulates the basic methods and fields that each node requires. Examples are the references to the left and right child or the encryptions of the contained key with its left and right neighbor. This is the reason why the inner node class does not contain any significant functions. The root and

[54]Or the reader's unlocked profile.

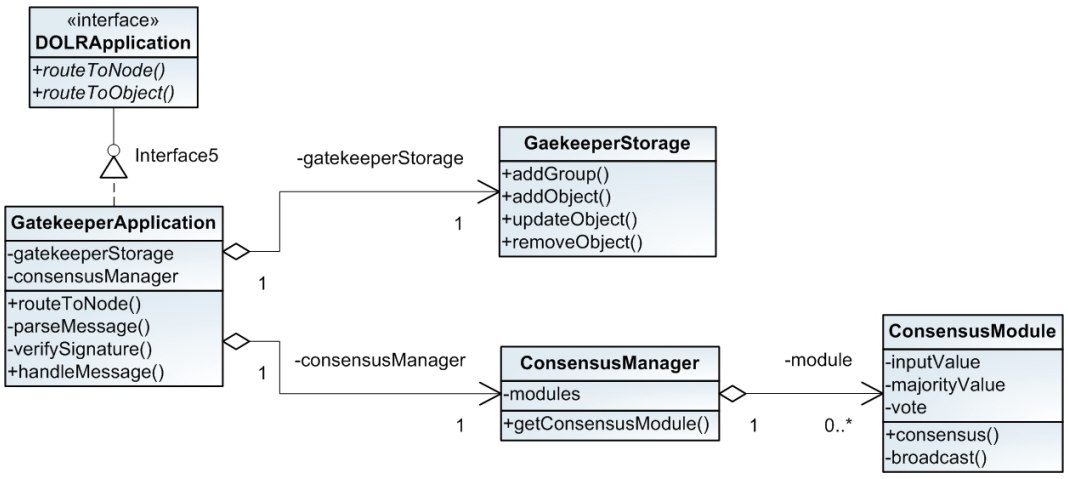

Figure 6.5: The gatekeeper application uses a consensus manager and storage to save group information persistently on the local device.

the leaf nodes are slightly more complex.

A leaf node must contain the public key of the reader to which it is associated. When the tree is constructed, the public key of the reader is used to fill up the index table. But the most important aspect of a leaf node is the *getPrivateKey()* method which allows a reader to access the private key stored at the root node. This method will recursively invoke the method on its parent node until the root is reached, which then returns the decrypted private key. Another detail to mention is the invalidation method that is called by a leaf if a reader is removed to invalidate the keys of all parent nodes.

The attentive reader might have realized that the root node differs from the one in Figure 5.10. This becomes even more apparent if one considers that there is an extension of the tree which allows the linking of key trees as described in Section 5.8. Therefore, the root does not only contain the readers' private and public key, but also a symmetric reader key as depicted in Figure 5.13. Consequently, the root also offers direct access to the reader group private key by offering an overloaded method *getPrivateKey()*. The owner's backdoor key is also stored at the root node and must be decrypted by the owner if entities are added or removed from the tree by calling *decryptOwnerSecretKey()*.

Recursion and overloading

What is not visible in the UML diagram in Figure 6.6 is that the tree heavily uses recursion and method overloading to provide a tight and structured design.

One last point to examine is the strategy used when readers are added to the tree. As mentioned, the key tree object maintains two lists to add and remove readers. In a first pass, all obsolete readers are removed by invalidating the respective nodes

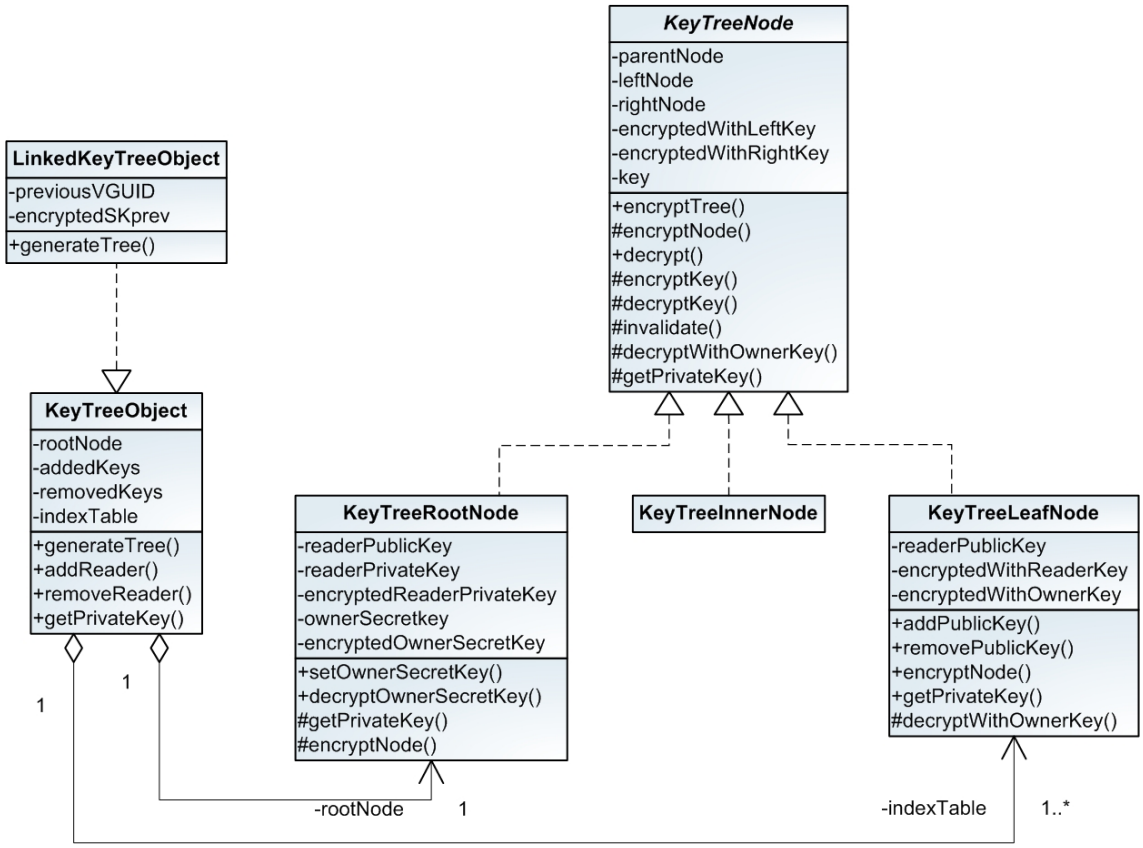

Figure 6.6: The key tree consists of four classes implementing the different node types: An abstract key tree node with common methods and fields and implementations is used for the root, inner and leaf nodes.

in the tree. Nodes are also removed if they no longer have at least one child node. In a second pass, the readers are added to the tree following a simple algorithm. The algorithm aims at generating a balanced tree in an efficient way by adding the readers at adequate branches.

Figure 6.7 gives a visual representation of the algorithm. In short, the algorithm adds subtrees to the existing tree such that the new subtree does not increase the height of the tree. This can be done until there is only one free branch. At this point, a subtree for all remaining readers is created which might change the height of the tree. However, this is not a problem since the tree can be assumed to be volatile and thus, subsequent modifications might re-balance the tree. One last point is that if there is no free branch, a new inner node is inserted which replaces either the left or the right child of the root, choosing the one with lower height. The existing subtree is appended to the left of the new inner node and a new subtree containing the keys of the joining readers is added on the right branch of the inner node. Figure 6.8 visualizes how the algorithm proceeds.

Preserve height of the tree

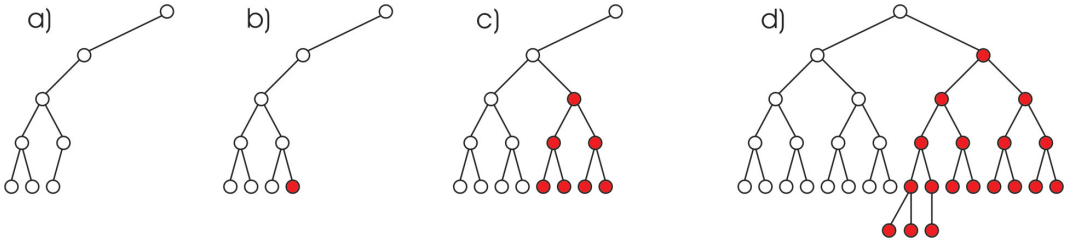

Figure 6.7: Visualization of the algorithm that adds readers to the tree. First, the algorithm adds subtrees such that the height of the tree does not grow as shown in a), b) and c). In the last free branch, a subtree for the remaining remaining readers is added which might unbalance the tree as step d) shows.

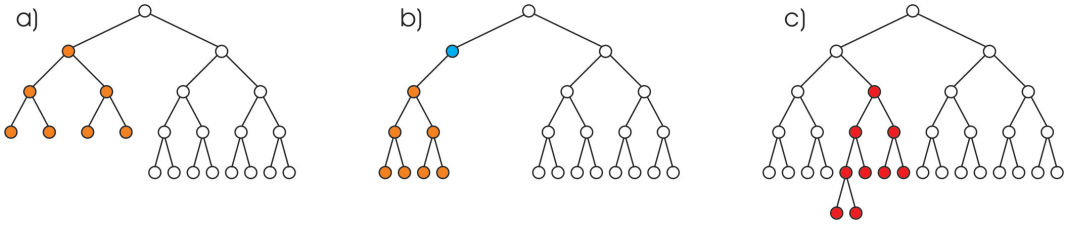

Figure 6.8: Adding readers to a full tree proceeds in three steps: a) The child of the root with the lower height (orange nodes) is selected to be moved; b) A new inner node (blue) is created and the selected subtree is appended as the left child; c) The new readers are added as a new subtree (red) that is appended on the right of the orange node. The resulting tree does not necessarily have to be balanced.

6.2.3 Gatekeeper Storage

Responsive data structures

The data structure to hold the gatekeepers' state is another important aspect of the implementation. One design decision was that the data structure should not behave like a unresponsive container. Instead, it should simplify the implementation of the gatekeepers by performing verifications on the stored data. One example is that the gatekeepers' storage verifies on an update of either the access control list or the key tree object so that the version counter is higher than the current one. This makes the implementation of the gatekeepers more concise. The implementation only requires two classes: *GatekeeperStorage* serves as a container that holds references to *GatekeeperStorageEntry* and dispatches requests to them. In turn, a storage entry contains all information of one group. This includes the owner's public key, the reader group public key, the group name, the VGUID of the ACL and KTO and their version counters as well as a list of object identifiers mapping to the most recent VGUID, which equals to the anchor in Section 4.3.

When an update on a gatekeeper is performed, be it on the access control list, the key tree object or a normal object, the gatekeeper updates his storage data structure

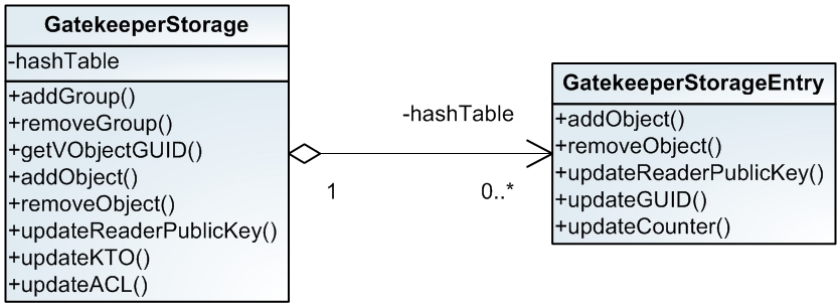

Figure 6.9: A storage entry maintains the state of one group, including all anchor objects. The gatekeeper storage then organizes all storage entries.

and immediately stores it on the local device. What has not been mentioned so far is that all data structures are serializable. This means that a data structure is converted into an equivalent byte array which can later be deserialized by passing it to the appropriate constructor.

Serializable data structures

There are other data structures such as the access control list or the witness object which are not discussed further. Their implementation is straightforward.

6.2.4 Unimplemented Features

The access control mechanism has been successfully implemented in Celeste and is operational. However, since it is a prototype, some of the features described in Chapter 5 have not been realized for three reasons:

1. They are not of any significant importance regarding the functionality of the system.
2. They require additional and unproportional efforts compared to the outcome.
3. They involve major changes to the existing Celeste system.

From the beginning on, it was intended that the implementation serves as a proof-of-concept and not as a complete and detailed implementation. Therefore, some simplifications seemed appropriate if they do not alter the general concept. The general concept includes the existence of gatekeepers and the key tree object comprising the most important interactions with the gatekeepers .

Challenge-response was one of the details that did not find its way into the implementation. This mechanisms would be of importance to avoid replay attacks and to guarantee freshness of messages. From a conceptual point of view, they only add additional complexity and do not have a direct influence on the general procedure.

Another point is the consensus protocol. An approximation of the protocol was implemented which is not equal to the king consensus protocol (cf. Section 5.2.6). For simplicity, it executes only two rounds where messages are exchanged between the gatekeepers. Nevertheless, it is believed that this is sufficient to estimate the performance of the system.

The theoretical description explains that the version object maintains additional entries for the key material such as a reference to the key tree object. The implementation does not follow this proposal because decryption is performed at the Celeste part of the object store layer. But at this level of abstraction, there is no direct access to blocks. One would have to shift the en- and decryption to the DOLR layer instead. For the prototype, it is sufficient to write the whole object instead of only certain blocks. This allows to carry out both de- and encryption at the Celeste part of the object store layer.

Last but not least, there is still the Serializer which executes write operations. The Serializer is thus a trusted component, and this is what gatekeepers intend to improve. Since the Serializer is deeply rooted in the Celeste system, the decision was made not to refactor the Serializer. But it is clear that a full and correct implementation would require to encapsulate the functionality of the Serializer within the gatekeepers. Nevertheless, the implementation was extended such that the Celeste system is able to read old objects passing a VGUID instead of an AGUID. This allows direct access to a certain version of an object which is of particular importance for the witness, access control and key tree objects. Or in other words: Although malicious entities can write unharmed to an object via the Serializer, their updates will not be considered since the gatekeepers hold the direct VGUIDs of the objects. Further, the tree maintains his own backward-linking which cannot be influenced by an attacker. Note that shifting de- and encryption to the DOLR layer implies trust in the node that executes those operations and also requires to think about the usefulness of the HTTP interface. Clearly, this is an issue which is outside the scope of a simple prototypical implementation.

7 Performance Analysis

This chapter gives the results of a performance test for both the original implementation and the one with access control capabilities. The goal of the performance test is to make a statement about the behavior of the tree-based access control scheme in a real-world setting and to compare the results to the theoretical estimations of Section 5.11. The tests measured the time to initialize a set of gatekeepers and the data structures for readers and writers. Additionally, the time for read and write operations was measured and compared with the performance of the original system.

7.1 Test Environment

The test environment consisted of 25 Linux machines, each with an Intel Pentium 4 3 GHz processor and 1 GB RAM. The hosts all belong to the department of electrical engineering at ETH and are connected by a switched 1 Gbit university-internal network. The Celeste source code was uploaded to all hosts using bash shell scripts[55]. The shell script initialized and started the performance test as soon as all nodes were running.

The test suite was written in Java. Every time an operation starts, the current system time is obtained and then compared to the system time after the operation finished. For each type of test, a log file is created that contains information about the parameters of the executed test, such as the number of readers or writers in the group and the number of gatekeepers.

[55] Most of them written by Marcel Baur.

In contrast to measurements with Celeste, the performance test on the key data structure was carried out separately on a multiprocessor machine with four Intel Xeon 3 GHz processors and a main memory of 2 GB. CPU time was not exclusively granted for the duration of the test and was shared with other processes and users.

7.2 Group Initialization and Membership Changes

This section analyzes the performance on group initialization. Initialization involves several steps and requires creating and updating data structures in the system. First of all, gatekeepers have to be selected and set up to guard access to a group. Then, entities can be added to or removed from a group, either as readers or as writers.

7.2.1 Gatekeeper Initialization

When an owner initializes a group, he needs to select entities as gatekeepers. The Gatekeepers are initialized with all necessary parameters such as the owner's public key or the name of the group. In addition to this procedure, the owner creates empty data structures for the access control list and the key tree object and stores them in the object store. As for updates on group membership changes, all gatekeepers are initialized in parallel. Figure 7.1 shows that the initialization is linear with a very small gradient.

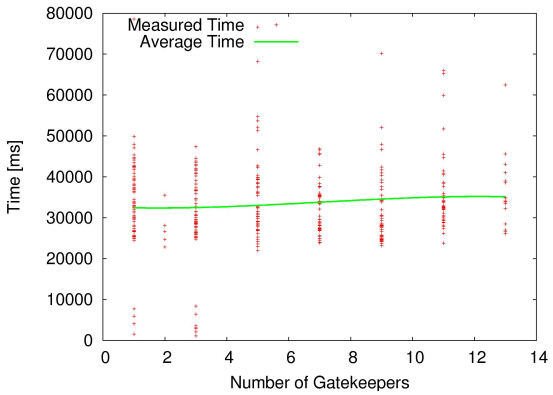

Figure 7.1: Average initialization time of gatekeepers is linear with a very small gradient.

The following two sections examine the time needed to change membership of either readers or writers.

7.2.2 Writer Membership Changes

In general, changes on the members of the writer group are assumed to be linear. There are no cryptographic operations and no consensus protocol run required. The

owner simply needs to retrieve the latest access control list and witness object, change the content of the ACL and inform all gatekeepers about this change.

Adding Writers

When initializing a group with m writers, the runtime turned out to be linear in the number of writers. The implementation allows to pass writers by their profile name that is then used to fetch the profile from the system and extract the public key for the access control list. The more writers are added to a group, the more time is needed to fetch all profiles. This explains the linear behavior in Figure 7.2.

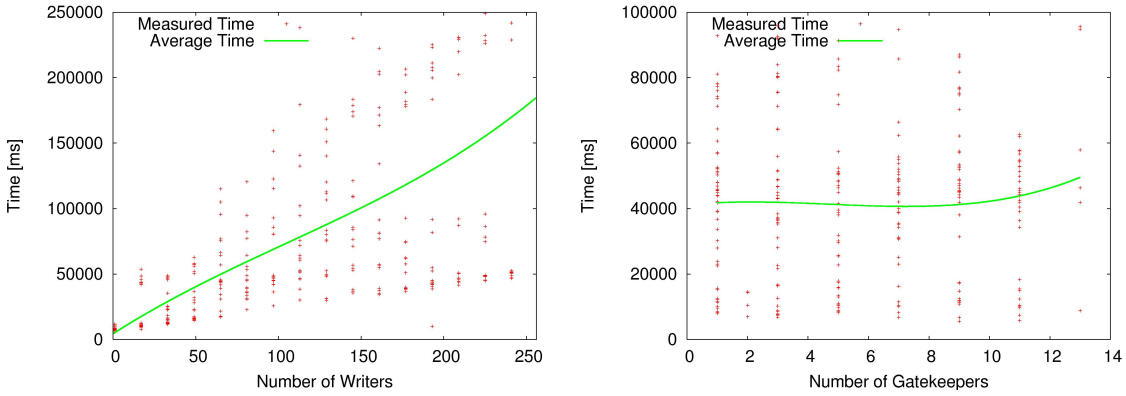

Figure 7.2: Measured and average time to add a linear number of writers to a new group with respect to the number of writers (left) and the number of gatekeepers (right) in the group.

The updates on the gatekeepers are constant because all gatekeepers are notified in parallel. The number of writers in the group has no influence on the time to update all gatekeepers.

Removing Writers

After the previous test had initialized a group of writers, the following test intends to determine the time needed to remove half of them. As before, the profile names of the writers to be removed are passed as parameters. Since the access control list also contains the identifier of the corresponding writer profile, there is no need to invoke the object store as before. Entries can be removed directly by using the profile GUID. Figure 7.3 shows that the trend is constant in the number of writers.

Figure 7.3 shows that the time to remove writers increases after 100 entities. With increasing number of GUIDs, the efficiency of the object store decreases dramatically. Because of decreased efficiency, the probability that a publish message for new GUIDs is not propagated in the system increases with the number of objects in the

Inefficient object store

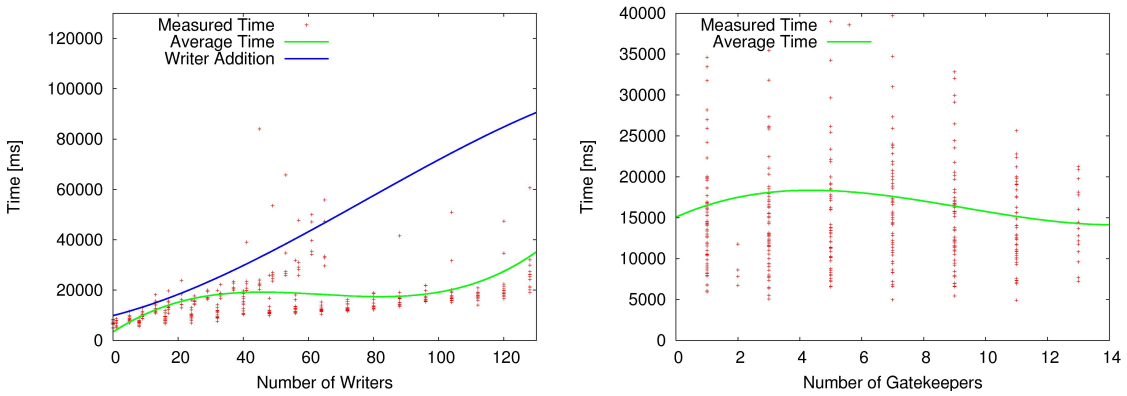

Figure 7.3: Measured and average time to remove half of the writers from an existing group with respect to the number of writers (left) and the number of gatekeepers (right) in the group. The x-axis shows the number of entities that were removed. The group's size was therefore twice as large before the removal.

system. In particular, the time for single operations increases the more profiles were involved [76].

7.2.3 Reader Membership Changes

Changes to the group of readers can be assumed to be more expensive than for writers. The reason is that the tree data structure is more complex than the simple access control list. It is required to perform cryptographic operations on each modification of the tree. One would expect to see the additional complexity for the cryptographic operations in the measurements on reader membership changes.

Adding Readers

When a new tree with a linear number of readers is created, the time for the initialization is linear in the number of readers for the same reason as for the access control list. The profile names are passed to the routine which retrieves all profiles from the object store. Figure 7.4 shows the time for this operation. Again, the notification of the gatekeepers is more or less constant.

Removing Readers

For the removal of readers, the same holds as for the removal of writers. Each leaf stores the reader's profile GUID together with his public key. The remove operation can then use the profile GUID to determine the leaves to be removed. There is no need to retrieve profiles for a remove operation. Hence, the time to remove half of the entities is constant as Figure 7.5 shows.

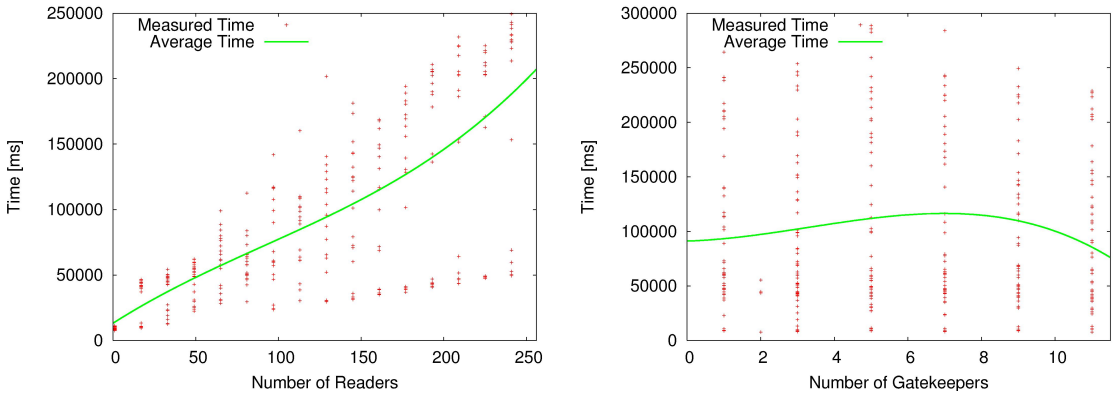

Figure 7.4: Measured and average time to add a linear number of readers to a new group with respect to the number of readers (left) and the number of gatekeepers (right) in the group.

7.3 Access Time

This test aimed at determining the time needed to read and write objects in the object store, once using the extended version that supports access control and once with the original system. Due to the additional complexity of access control, it is expected that the measured time is not as good as in the native system. As before, there are two variables which are of interest: The number of readers/writers and the number of gatekeepers.

7.3.1 Write Access

The time needed to write to an object was as anticipated. A write operation is always independent of the number of writers in the group. The cryptographic keys can be generated and used without the need for a data structure such as the tree. Of course, the size of the access control list grows linearly in the number of writers which means that gatekeepers need more time to fetch the access control list if the number of writers is large. Since one entry in the access control list is in the order of a few bytes, write access can still be assumed to be constant with respect to the number of writers. Figure 7.6 illustrates this circumstance.

When comparing the time for a write access compared to the number of gatekeepers, the measured time is linear in the number of gatekeepers as shown in the left plot of Figure 7.6. The most important reason is that gatekeepers need to agree on the written object using a consensus protocol. For each gatekeeper, the message complexity is in the order of $O(tn)$, where n is the number of gatekeepers and $t < n/4$. Although the prototypical implementation only executed a constant number of rounds, the complexity of the consensus protocol is obvious. Moreover, a write operation involves additional retrieve operations in the object store. A writer

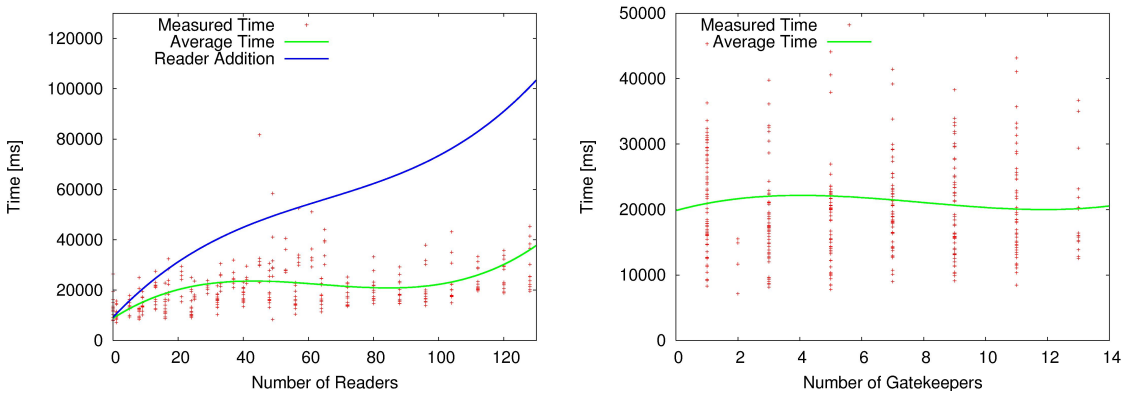

Figure 7.5: Measured and average time to remove a half of the readers from an existing group with respect to the number of writers (left) and the number of gatekeepers (right) in the group. The x-axis shows the number of entities that were removed. The group's size was therefore twice as large before the removal.

needs to retrieve the latest witness object and the key tree object. A gatekeeper must also determine the witness object to communicate with the other gatekeepers and fetch the latest access control list.

A separate experiment with ten gatekeepers and ten writers shows the distribution of the costs. For a write operation with one megabyte of data, the time to store the written object and the time to wait for the gatekeepers' replies is the dominating factor. The cryptographic operations are negligible as Figure 7.7 shows.

The implementation waits until all gatekeepers have completed their operations without using a timeout. This means that the slowest gatekeeper dominates the write operation. Besides updating their local state, gatekeepers basically have to retrieve the writers' access control lists and perform a consensus protocol. Compared to the retrieval of the witness object in Figure 7.7, the time to fetch the ACL is significantly higher. The reason is that Celeste uses locks to ensure serialized access to an object. Since all gatekeepers need to retrieve the ACL concurrently, the locking mechanism forces gatekeepers to wait until the object is unlocked. This also holds for the consensus protocol that internally uses locking mechanisms. The time to retrieve the witness object is included in the time for the consensus protocol. It can be assumed that the time to retrieve the witness object is equal to the to retrieve the ACL. The costs of a gatekeeper are shown in Figure 7.8.

7.3.2 Read Access

Read access is independent of both the number of gatekeepers and the number of readers. Once more, the costs for the retrieval of the objects is responsible for the additional time compared to the original object store without access control.

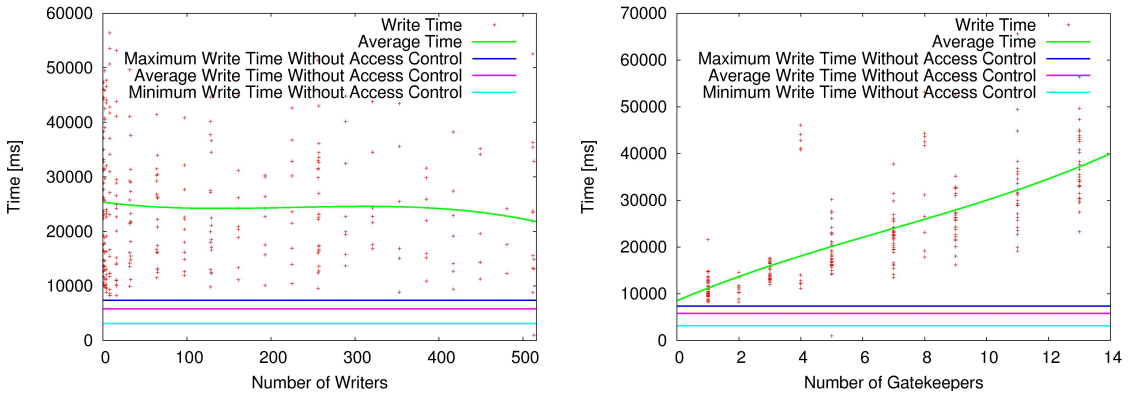

Figure 7.6: Measured and average time for a write operation given a certain number of writers (left) and gatekeepers (right).

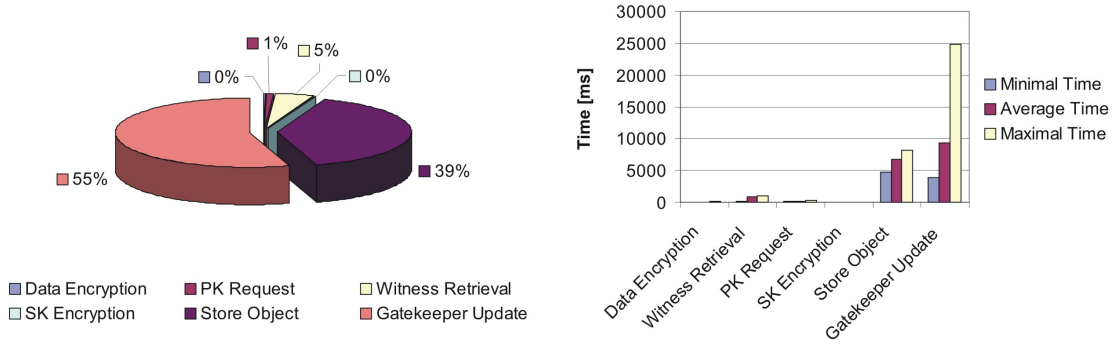

Figure 7.7: Storing the written object and updating the gatekeepers with the new object identifier are the dominating costs for a write operation.

A reader needs to retrieve at least three objects - two more than in the original implementation. Moreover, there is one broadcast to all gatekeepers and the decryptions on the tree and the object. As explained in the previous section, the number of object retrievals is the significant factor for the read procedure. The number of cryptographic operations and the parallel broadcast to all gatekeepers is negligible compared to those costs (cf. Figure 7.10).

Moreover, the results in Figure 7.9 underline the statement from Section 5.11.5 which argued that $O(k + \log m) \approx O(k)$. Or in other words: The time to retrieve all necessary objects is much larger than the time to decrypt the tree. The number of gatekeepers is of no importance since they can be contacted in parallel and there is no consensus protocol to be executed as for a write operation (cf. right plot of Figure 7.9).

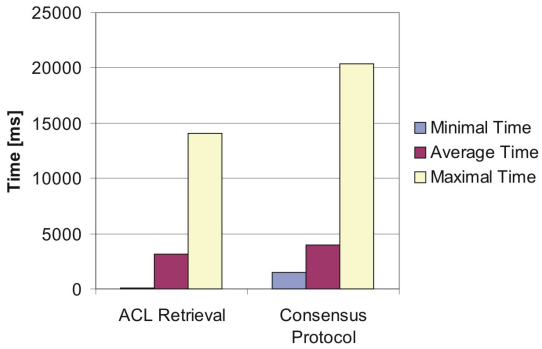

Figure 7.8: The costs for gatekeepers are dominated by the consensus protocol and the retrieval of the writers' access control list.

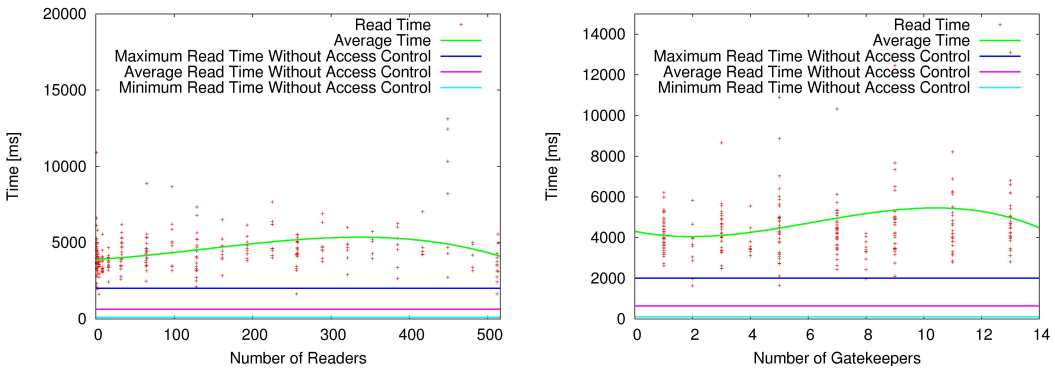

Figure 7.9: Measured and average time for a read operation given a certain number of readers (left) and gatekeepers (right).

7.4 Operations on the Tree Data Structure

The previous sections described the measurements with Celeste as the underlying object store. Although the measured times give information about the general behavior in a real-world setting, all of them are influenced by the implementation of the object store. Hence, it is desirable to measure the time on the key tree object as a data structure independent from an object store as well. The goal is to determine whether the theoretical estimates in Section 5.11 are comparable to concrete measurements.

For the tree data structure, there are three operations which are of particular interest: Creation of the tree for a given number of readers, access to the tree's root private key and removal of a constant number of keys.

The number of public keys which form the leaves of the tree was iteratively

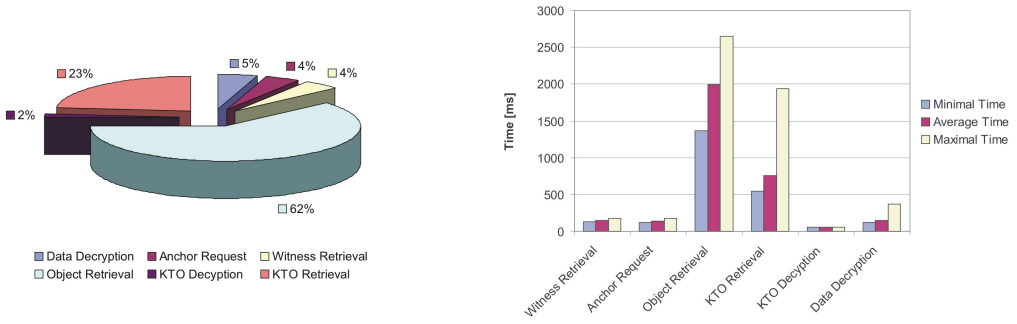

Figure 7.10: Retrieval of the KTO and the object needs most of the time of a read operation.

incremented, starting with four keys up to $2^{14} = 16'384$. In every iteration, the number of public keys was incremented by four. A fresh tree was created with the respective number of leaves and then a full encryption on the tree was performed. The creation time measured both, the time to create the full tree including all keys and the time to encrypt each node of the tree. The generation of the reader group private key was also added to the tree creation time, but not the creation of the public keys of the leaves. In a practical system, those public keys can be assumed to be given. All public/private-key pairs were generated using RSA and a key length of 1024 bit. For the symmetric keys, AES was chosen with a key length of 128 bit.

The expected time resulted from a simple extrapolation. In an experiment with 100'000 iterations, the time for single cryptographic operations was measured. Their average value is depicted in table 7.1.

	PK_{Gen}	PK_{Enc}	PK_{Dec}	SK_{Gen}	SK_{Enc}	SK_{Dec}
Time [ms]:	386.862	0.6791	8.82806	0.11733	0.12825	0.0901

Table 7.1: Average time for one cryptographic operation of the given types.

Section 5.11.3 explains that the number of asymmetric encryptions is m, the number of symmetric-key generations is $2m - 1$ and the number of symmetric encryptions is $3m - 1$, where m denotes the number of readers that are added. With those formulas, an expected time can be computed which is linear in the number of leaves or public keys respectively. Figure 7.11 depicts the measured and expected times for the creation of a tree. The implementation is efficient and is below the theoretically expected time. But more important is that the statement for linearity of the tree creation could be underlined with this experiment.

The time needed to access the tree is as expected. Because of the logarithmic characteristic of the tree, the number of secret key operations is of no significance.

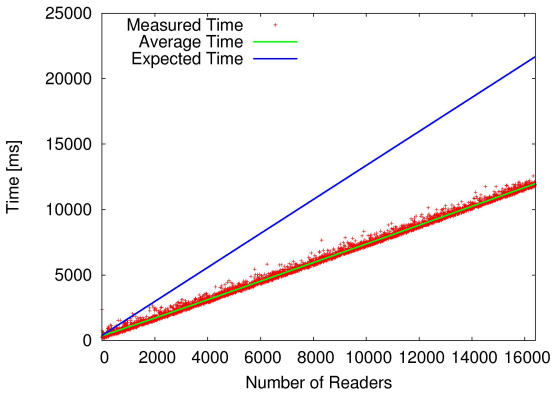

Figure 7.11: Measured and expected time to create the tree data structure.

Once more, this emphasizes the approximation from Section 5.11.5 which stated that $O(k + \log m) \approx O(k)$. The result is depicted in Figure 7.12.

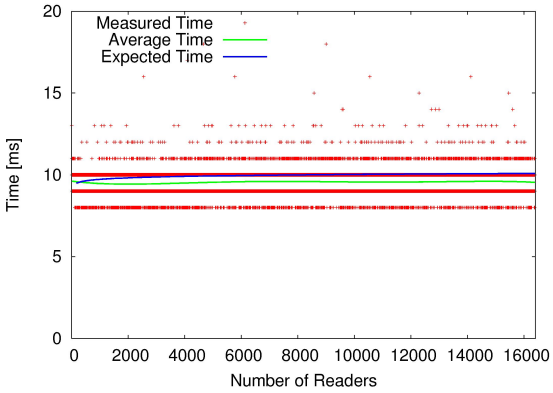

Figure 7.12: Measured and expected time to access the tree data structure.

At last but not least, the removal of a constant number of keys from a given tree is as estimated. The concrete performance test always used four as the constant number independent of the size of the tree. To compute the expected time, the precise formula from Section 5.11.3 was used. Figure 7.13 shows the results for the removal test.

The reason for the strong variance in Figure 7.13 is probably the generation of the new keys in the tree. The stand-alone measurements for the cryptographic operations that resulted in table 7.1 revealed significant differences for both, the symmetric and asymmetric key generation algorithms.

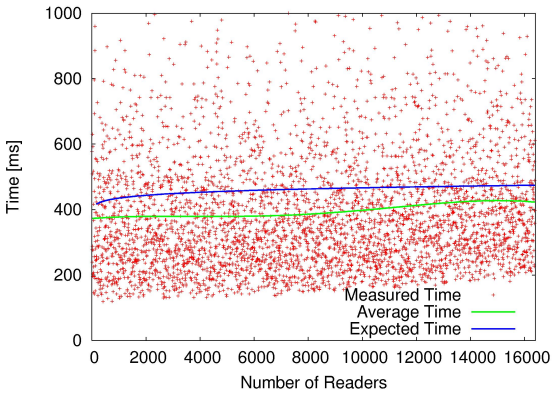

Figure 7.13: Measured and expected time to remove four keys from the tree data structure.

7.5 Conclusion

Deterioration of performance by constant factor

It is not surprising that the measured times for an object store with access control are worse than for a system without access control. Fortunately, the needed time is only worse by a constant or linear factor, which is suitable for practical systems.

The measured time to add new group members was linear in the number of entities because their profiles needed to be retrieved from the object store. Membership updates require updates on data structures and gatekeepers. In the implementation, the gatekeepers are all notified in parallel, which might be the reason why the number of gatekeepers did not have any significant impact on the measured times.

Removal of entities is constant since profiles do not need to be fetched from the object store. Entries can be directly removed by using the profile GUID.

It turned out that write operations are the most expensive form of interaction in the system. Although the number of writers has no impact on the performance, the time grows linearly in the number of gatekeepers. The reason is that the gatekeepers need to perform a consensus protocol to agree on the validity of the write request. A more efficient implementation might reduce the gradient of the curve, but the general behavior can be expected to remain linear.

Since on a read request, gatekeepers do not need to communicate with each other, read access time is constant or linear.

The time needed for operations on the tree data structure meets the theoretical estimations of Section 5.11. Creation of the tree is linear while reader removal and tree access turned out to be nearly constant even for a tree with 2^{14} public keys. From the measurements, it seems justified to conclude that the tree as a data structure is adequate for large groups.

Due to the constant or linear behavior of the system, one can conclude that gatekeepers and the tree data structure are suitable for access control in a P2P storage system.

8 Conclusion and Future Work

In the preceding chapters, techniques and mechanisms to realize access control in a P2P storage system were presented. This chapter concludes the thesis and summarizes the main contributions including their limitations and relevance. Moreover, it mentions future work to be done in the area of P2P access control.

8.1 Conclusion

Security in general and P2P access control in particular are difficult topics. The task of the central authority that controls access to objects must be replaced by a distributed reference monitor called gatekeepers.

8.1.1 Contribution

This thesis described in-depth how gatekeepers communicate among themselves and interact with other types of entities.

The group of gatekeepers can be changed by the owner or by the gatekeepers themselves if they are self-organizing. A central aspect is that they must be found by the members of the group since both readers and writers need the gatekeepers for their operations. One important contribution of this thesis is a novel method to determine the current set of gatekeepers without the need for a dedicated authority by using a secure version identifier (SVID). The way how the SVID is constructed is the key to determine subsequent versions of the witness object only by local

computations.

The thesis explained in detail how write operations are processed. In principle, gatekeepers have to perform consensus protocol rounds to ensure that all honest gatekeepers agree on the same written object.

Read access has to be handled differently because it is implicitly defined via the knowledge of a key. Two of the presented schemes, the list- and tree-based approaches, only involved gatekeepers to get the latest version of an object. Freshness of objects is an issue that was never addressed by other existing schemes. It is important because the semantic of a replay of an old object version is equal to an improper write operation. As an orthogonal approach, a scheme based on secret sharing was explained that shared the decryption keys among the gatekeepers instead of storing them in some data structure.

The implementation in Celeste allows evaluate the mechanisms under realistic conditions.

8.1.2 Evaluation

The theoretical asymptotic runtime analysis revealed interesting properties of all schemes. The list-based scheme is efficient for adding readers as well as read access, but expensive with respect to the removal of members. The tree-based approach is in the order of the list-based scheme, but is significantly more efficient when considering leaving readers in the average case. The benefit of the sharing-based scheme is that changes on the membership are in any case constant.

The implementation and the subsequent performance tests underlined the estimations of the theoretical complexity analysis for the tree data structure. Measurements revealed that the consensus protocol and the retrieval of the required objects are time-consuming while cryptographic operations do not preponderate. This might be different for other object store implementations. If the retrieval of objects is the bottleneck of the system, it might be worth considering the sharing-based approach because read and write attempts do not require the retrieval of objects[56] for a writing or reading entity[57].

8.2 Limitations and Future Work

Although this thesis covered a wide range of topics related to P2P access control, not all issues could be solved and considered. First of all, only two types of

[56] Except for the witness object that is always required.

[57] The gatekeepers need to retrieve the access control lists and the key shares. If there is enough disk space, gatekeepers can store this information locally to speed up the time for an access attempt significantly.

permissions were explicitly discussed, namely read and write permissions. But there are other permissions that need to be examined such as append, truncate or delete. For instance, append permissions require gatekeepers to verify that only blocks are added to the object and none of the existing ones are deleted or overwritten. Using king consensus of appendix A on only the header object's ID is insufficient. A similar problem occurred for version capabilities (cf. Section 5.9.2). Of course, the king can send the header object along with his messages. But this has a negative impact on the efficiency of the consensus protocol with growing size of the header object.

Another issue is the bootstrap problem for joining entities. How do they know that they were added to the group? How is the group information transmitted to the joining entities in a secure way? In general, one can send such information by email, publish it on some website or use a well-known Internet service. However, an initialization of joining members that uses other mechanisms than the ones supplied by the object store implies additional complexity. The desirable solution is that such kind of notifications can be handled by the object store itself. If append permission is implemented, one can sketch out a scheme that follows closely normal read/write access control: Each user initializes a group of gatekeepers to guard access to messaging objects. To find the witness object of those gatekeepers, the owner's name can be used as the group name. Each entity can then send messages to the gatekeepers that are appended to some object depending on the type of message.

A problem that was not addressed is how to determine which objects are contained within a group. The text implicitly assumed that the mapping of objects to groups is known. In general, such a knowledge cannot be presumed. A straightforward solution is to request all anchors from the gatekeepers, which allows denial-of-service attacks quite easily. Alternatively, the owner could maintain an object that lists all IDs of the objects contained in a group and pass a reference to the gatekeepers. Whenever the owner adds or removes objects, he updates the dedicated object and informs all gatekeepers. A request for all anchors in the group can then be reduced to simply one identifier that must be requested. Alternatively, one could use a similar mechanism for the object containing all references as for the witness object. Readers and writers can then find it on their own.

Finding the witness object was one of the major challenges. It is similar to the bootstrap problem mentioned above. Members of a group must be able to find and identify the current gatekeepers definitely. Since there is no authority that can refer to the latest gatekeepers, group members must compute the witness object's identifier on their own. The current proposal involves cryptographic operations like digital signatures. It is a challenge to find another mechanism that minimizes the cryptographic operations further. Moreover, the secure version identifier only works if the system can guarantee availability as assumed in this thesis. However, if a witness object can temporarily not be found, operations are carried out on the wrong group of gatekeepers.

Former readers turned out to be a severe and hard to solve problem. Because of efficiency reasons, re-encryption of all affected objects is not feasible. The presented approaches described methods which are practically feasible, but nevertheless allow readers to access old objects in some cases. The secret sharing-based approach turned out to be very robust against former readers since gatekeepers control the key shares. There might be several other approaches to tackle the problem of former readers, for example an adequate encryption scheme or schemes using erasure codes or secret sharing.

It has been shown that hierarchies of groups are very expensive. For many applications, hierarchies are not necessary. But for those applications that require hierarchies of groups, it is important to find a scheme that can handle operations on hierarchies efficiently.

Last but not least, the mechanism for self-organizing gatekeepers used two black-box protocols: One that determines which gatekeepers are excluded and one that defines the entities which are added. Both algorithms imply consensus among the gatekeepers. Therefore, a simple algorithm would perform separate consensus protocol rounds for each joining and leaving entity. As previously mentioned, a consensus protocol is expensive and its use should therefore be minimized. It is a challenge to describe two efficient algorithms that meet the required specification. Another problem that occurred was the re-sharing of the signature key. The secure approach requires three-dimensional polynomials and is not efficient since many broadcast rounds are required. The approach that was chosen in this thesis is not secure in every case, but is a good and efficient approximation to the problem. The next step is to further improve the re-sharing scheme regarding security while retaining efficiency.

A King Consensus

In this section, the king consensus protocol is described [54, 55, 56] that will be used by the gatekeepers to agree on a decision of a write operation. A modified version of the protocol is used [57, 58] that allows an arbitrary value domain $\{0,1\}^k, k > 0$ since the standard protocols only foresee binary input values, but requires that $t < n/4$ instead of $t < n/3$, where n is the number of players P. For the following considerations, the standard communication model with a complete (fully connected) synchronous network of pairwise authentic channels among the players is assumed.

A.1 King Consensus Protocol

The definition of king consensus is as follows:

Definition A.1.1 *A protocol achieves king consensus (with respect to p_k) if it satisfies the following conditions.*

Termination: *All correct players terminate the protocol after a finite number of communication rounds.*

Consistency: *If player p_k is correct then all correct players agree on the same value $v' \in \{0,1\}^k$ at the end of the protocol.*

Persistency: *If all correct players enter the protocol with the same input $v \in \{0,1\}^k$ then $v'_p = v$ for every correct player p.*

The goal of the king is to affect the decision of the other players. Before the consensus protocol is started, it is determined[58] which $t+1$ players out of n play the role of the king. In the first phase of a round, all players broadcast their values. In the second phase, the king of the corresponding round additionally sends his value to all players. Players which do not yet have a solid majority of values will then decide for the king's value. Player p executes phase k of the king consensus protocol as follows:

Protocol A.1.1 $KingConsensus_{p_k}(v_p)$

1. *SendToAll*(v_p); $\forall q \in P$ *do Receive*(v_q) *od;*
2. $V_x := \{q \in P : v_q = x \wedge \nexists y : y \neq x \wedge |V_y| > |V_x|\}$;
3. $v'_p := x$;
4. *if* $p = p_k$ *then SendToAll*(v'_p);
5. *else Receive*(v'_k);
6. *if* $|V_x| < n/2 + f$ *then* $v'_p := v_k$ *fi;*
7. *fi;*
8. *return* v'_p

Based on the king consensus protocol, one can derive the consensus protocol which conforms to the following definition:

Definition A.1.2 *A protocol achieves consensus if it satisfies the following conditions.*

Termination: *All correct players terminate the protocol after a finite number of communication rounds.*

Consistency: *All correct players decide on the same output value.*

Persistency: *If all (correct) players initially hold the same input value v then all correct players decide on v.*

The idea of king consensus is that there are $t+1$ different kings in each phase of the protocol. If the honest players do not hold the same value, the king can break ties by broadcasting his value. Since $t+1$ rounds are executed, it is ensured that there is at least one honest king. The full consensus protocol then proceeds as follows:

[58] In the context of this thesis, one possible solution is to choose the first $t+1$ gatekeepers in the witness object list that are available.

Protocol A.1.2 $Consensus(v_p)$

1. *for* $k := 0$ *to* t *do* $v_p := KingConsensus_{p_k}(v_p)$ *od;*
2. *return* v_p

The message complexity of the consensus protocol is $O(tn^2)$. See also [58] for correctness proofs of those consensus protocols.

List of Figures

List of Tables

Bibliography

[1] http://isb.wa.gov/policies/definitions.aspx#IndexA

[2] http://web.mit.edu/kerberos/www/

[3] D. Kahn, "The Codebreakers", Macmillian, New York, 1967

[4] C. E. Shannon, "Communication theory of secrecy systems", Bell Sys. Tech. J., 28:657-715, 1949

[5] M. E. Hellman, "An extension of the Shannon theory approach to cryptography", IEEE Trans. Inform. Theory, IT-23:289-294, 1977

[6] R. L. Rivest, "Cryptology", MIT Laboratory for Computer Science

[7] U. Maurer, D. Basin, Lecture Notes from "Information Security", Swiss Federal Institute of Technology (ETH), 2005-2006

[8] U. Maurer, Lecture Notes from "Cryptographical Protocols", Swiss Federal Institute of Technology (ETH), 2004

[9] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", In Proc. of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, NM, Nov. 2022, 1994, IEEE Computer Society Press, pp. 124134.

[10] D. Wells, "The Penguin Book of Curious and Interesting Puzzles", Penguin Books, 1992

[11] Oystein Ore, "Number Theory and Its History", Dover Publications, 1976

[12] K. Ireland and M. Rosen, "The Chinese Remainder Theorem." chapter 3.4 in "A Classical Introduction to Modern Number Theory", 2nd ed. New York: Springer-Verlag, pp. 34-38, 1990

[13] K.S. McCurley, "The discrete logarithm problem", Proc. Symp. Appl. Math. 42 (1990) 49-74.

[14] R. P. Brent, "Recent Progress and Prospects for Integer Factorisation Algorithms", Computing and Combinatorics, 2000, pp.3-22

[15] M. Blum and S. Micali, "How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits", SIAM J. on Computing, Vol. 13, 1984, pp. 850-864, FOCS 1982.

[16] O. Goldreich, H. Krawczyk, and M. Luby, "On the Existence of Pseudorandom Generators", 29th FOCS, 19SS, pp. 12-24.

[17] L.A. Levin, "One-Way Function and Pseudorandom Generators", Combinatorics, Vol. 7, No. 4, 1987, pp. 357-363, STOC 1.985.

[18] A. Shamir, "How to share a secret", Communications of the ACM 22 (1979), 612-613.

[19] G. Blakley, "Safeguarding cryptographic keys", Proceedings of AFIPS 1979 National Computer Conference, vol. 48, 1979, pp. 313–317

[20] G.J. Simmons, editor, "Contemporary Cryptology - The Science of Information Integrity", IEEE Press, 1992.

[21] C. A. Asmuth and J. Bloom, "A modular approach to key safeguarding", IEEE Transactions on Information Theory, IT-29(2):208–210, 1983.

[22] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game - a completeness theorem for protocols with honest majority", In Proc. 19th ACM Symposium on the Theory of Computing (STOC), pages 218-229, 1987

[23] U. Maurer, Lecture notes "Cryptographic Protocols", ETH Zurich

[24] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation", In 20th annual Symposium on the Theory of Computing (STOC), pages 1-10, 1988

[25] T. Rabin and M. Ben-Or, "Verifiable Secret Sharing and Multiparty Protocols with Honest Majority", in Proc. 21st STOC, pages 73–85. ACM, 1989.

[26] http://www.openssl.org/docs/crypto/des_modes.html

[27] R.L. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, pages 120-126, 1978

[28] B. Lampson. "Protection". Proc. 5th Princeton Conf. on Information Sciences and Systems, Princeton, 1971. Reprinted in ACM Operating Systems Rev. 8, 1 (Jan. 1974), pp 18-24.

[29] http://csrc.nist.gov/rbac/

[30] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, "Role-based access control models", IEEE Computer, 29(2), 38-47, 1996

[31] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

[32] D.E. Bell and L.J. LaPadula, "Secure Computer Systems: Mathematical Foundations and Model", Mitre Corp. Report No. M74-244, Bedford, Mass., 1975

[33] K.J. Biba, "Integrity Considerations for Secure Computer Systems", Mitre Corp. Report TR-3153, Bedford, Mass., 1977

[34] Yair Frankel, Philip D. MacKenzie and Moti Yung, "Robust Efficient Distributed RSA-Key Generation", The Thirtieth Annual ACM Symposium on Theory of Computing, STOC '98

[35] Dan Boneh and Matthew Franklin, "Efficient Generation of Shared RSA keys (1997)", Lecture Notes in Computer Science

[36] J. McHugh, "Covert Channel Analysis", Technical Memorandum 5540:080A, Naval Research Laboratory, Washington D.C., 1995. A Chapter of the Handbook for the Computer Security Certification of Trusted Systems.

[37] D.F.C. Brewer and M.J. Nash, "The Chinese Wall Security Policy", Proc. IEEE Symp. Research in Security and Privacy, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 215-228

[38] R.S. Sandhu, G. Mason, "Lattice-Based Access Control Models", IEEE Computer, 9-19, 1993

[39] M. Waldvogel, G. Caronni, D. Sun, N. Weiler, B. Plattner, "The VersaKey Framework: Versatile Group Key Management", IEEE journal on selected areas in communications, Vol. 17, no. 8, August 1999

[40] R. Gennaro, T. Rabin et al., "Robust and efficient sharing of RSA functions", J. Cryptology (2000) 13: 273300, DOI: 10.1007=s001459910011

[41] C. Attiya, D. Dolev, J. Gil, "Asynchronous Byzantine Consensus", Proceedings of the third annual ACM symposium on Principles of distributed computing, Pages: 119 - 133, 1984, ISBN:0-89791-143-1

[42] D. Bindel, Y. Chen et al., "Oceanstore: An Extremely Wide-Area Storage System", Technical Report UCB/CSD-00-1102

[43] J. Kubiatowicz, D. Bindel et al., "OceanStore: An Architecture for Global-Scale Persistent Storage", In Proc. of ASPLOS 2000

[44] A. Adya, W. Bolosky, "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment", In Proc. of OSDI 2002, Boston, MA, December 2002

[45] R. Merkle, "Protocols for Public Key Cryptosystems", IEEE Symposium on Security and Privacy, 1980

[46] R. Merkle, "A digital signature based on a conventional encryption function", In proc. of Crypto 1987, volume 293 of LNCS, pages 369-378, Springer-Verlag, August 1987

[47] A.J. Menezes, P. C. van Oorschot, S. A. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997

[48] A. Muthitacharoen, R. Morris et al., "Ivy: A Read/Write Peer-to-Peer File System", 5th Symposium on Operating Systems Design and Implementation, Boston, MA. December 2002

[49] F. Dabek, M. Frans Kaashoek et al., "Wide-area cooperative storage with CFS", in proc. of the ACM Symposium on Operation System Principles, October 2001

[50] FIPS 180-1, "Secure Hash Standard", U.S. Departemnt of Commerce/N.I.S.T, National Technical Information Service, April 1995

[51] E. Goh, H. Shacham et al., "SiRiUS: Securing Remote Untrusted Storage", In proceedings of the Internet Society (ISOC) Network and Distributed Systems Security (NDSS) Symposium 2003, pp. 131-145.

[52] https://www.trustedcomputinggroup.org/faq/TPMFAQ/

[53] M. Pease, R. Shostak, L. Lamport, "Reaching agreement in the presence of faults", J. ACM 27,2 (Apr. 1980), 228-234

[54] P. Berman, J. A. Garay, and K. J. Perry. "Towards optimal distributed consensus (extended abstract)", In 30th Annual Symposium on Foundations of Computer Science, pages 410-415, Research Triangle Park, North Carolina, 30 October-1 November 1989. IEEE.

[55] P. Berman, J. Garay and K. Perry, "Bit Optimal Distributed Consensus", Computer Science Research, Plenum Publishing Corporation, NY, NY, 1992

[56] M. Fitzi, U. Maurer, "From partial consistency to global broadcast", Proceedings of the thirty-second annual ACM symposium on Theory of computing, Portland, Oregon, United States, Pages: 494 - 503, 2000

[57] R. Wattenhofer, Lecture Notes "Distributed Systems", http://vs.inf.ethz.ch/edu/WS0304/VS/

[58] H. Attiya, J. Welch, "Distributed Computing: Fundamentals, Simulations, and Advanced Topics", Second Edition, John Wiley and Sons, Inc., ISBN 0-471-45324-2

[59] D. Bleichenbacher, "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1", in Advances in Cryptology – CRYPTO'98, LNCS vol. 1462, pages: 1–12, 1998

[60] M. Hirt, U. Maurer, and B. Przydatek, "Efficient Secure Multi-Party Computation", Advances in Cryptology - ASIACRYPT '00, Lecture Notes in Computer Science, Springer-Verlag, vol. 1976, pp. 143-161, Dec 2000

[61] G. Caronni, R. Rom, G. Scott, "Celeste: An Automatic Storage System", White Paper, 2004

[62] A. Rowstron and P. Druschel,"Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility", 18th ACM SOSP'01, Lake Louise, Alberta, Canada, October 2001

[63] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility", HotOS VIII, Schoss Elmau, Germany, May 2001

[64] www.caleido.com/kangoo

[65] Dominik Grolimund, Luzius Meisser et al., "Cryptree: A Folder Tree Structure for Cryptographic File Systems", 25th IEEE Symposium on Reliable Distributed Systems (SRDS), Leeds, United Kingdom, October 2006.

[66] Private conversation with Dominik Grolimund, Luzius Meisser and Stefan Schmid, August, 15. 2006.

[67] Hari Balakrishnan, M. Frans Kaashoek et al. "Looking up data in P2P systems", In Communications of the ACM, February 2003

[68] P. Druschel and A. Rowstron, "Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems", In Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Nov. 2001.

[69] S. Ratnasamy, P. Francis et al., "A scalable content-addressable network", In Proc. ACM SIGCOMM, San Diego, CA, Aug. 2001, pp. 161172.

[70] I. Stoica, R. Morris et al., "Chord: A scalable peer-to-peer lookup service for Internet applications", In Proceedings of the ACM SIGCOMM 01 Conference, San Diego, CA, USA, Aug. 2001.

[71] B. Y. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing", Tech. Rep. UCB/CSD-01-1141, University of California at Berkeley, EECS Department, 2001.

[72] G. Caronni, R. Rom, G. Scott, "Maintaining Object Ordering in a Shared P2P Storage Environment", Technical Report, Sun Microsystems Laboratories, Report Number: TR-2004-137, 2004

[73] F. Dabek, B. Zhao et al., "Towards a common API for structured peer-to-peer overlays", in Pro. of 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03), Feb. 2003

[74] Extreme programming, http://www.extremeprogramming.org/rules/testfirst.html

[75] JUnit, http://www.junit.org/index.htm

[76] Private conversation with Germano Caronni, 13.09.2006