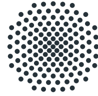 Universität Stuttgart

Deletion of Content in Large Cloud Storage Systems

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Tim Waizenegger

aus Stuttgart

Hauptberichter: Prof. Dr.-Ing. habil. Bernhard Mitschang

Mitberichter: Prof. Dr.-Ing. Dr. h. c. Theo Härder

Tag der mündlichen Prüfung: 10.11.2017

Institut für Parallele und Verteilte Systeme
der Universität Stuttgart

2017

Contents

1 Introduction 13
- 1.1 Contributions and Outline of the Thesis 13
- 1.2 Motivation, Standards, and Regulations 18
- 1.3 The Concept of Cryptographic Deletion 20
- 1.4 Definition of Secure Deletion 21
- 1.5 The Key Management Problem 24
- 1.6 Problem Statement and Requirements 28

2 Background and Related Work 31
- 2.1 Secure Deletion 32
  - 2.1.1 Classification of Approaches 32
- 2.2 Wrapped Keys and Chains of Keys 35
  - 2.2.1 Example: Practical Disk Encryption 36
  - 2.2.2 Authenticated Key Wrapping and Validation 36
- 2.3 Cryptographic Hardware 38
  - 2.3.1 Improving Security with Cryptographic Hardware . . 39
- 2.4 Related Work 42
  - 2.4.1 Secure Deletion 42
  - 2.4.2 Cryptographic Deletion 44
  - 2.4.3 Encryption in Database Systems 50

3 Key-Cascade Method 55
3.1 Key-Cascade: Abstract Idea 57
3.2 Data Structure and Key Organization 59
3.2.1 Static Tree Benefits 61
3.3 Data Structures and Storage Locations 62
3.4 Complexity and Efficiency Consideration 63
3.5 Key-Cascade Geometry Examples 65
3.6 ID Calculations 67
3.7 Secure Deletion Through Cascaded Re-Keying 69
3.8 Operations: Key Creation, Retrieval, and Deletion 71
3.9 Related Publications 72

4 Extensions to the Key-Cascade Concept 75
4.1 Resizing the Key-Cascade 76
4.2 Deletable Key 79
4.3 Deletable Key Sources 81
4.3.1 Key Source: Static Key 84
4.3.2 Key Source: Password-Derived Key 85
4.3.3 Key Source: Trusted Platform Module 85
4.4 Object ID Index 86
4.5 Free List 87
4.6 Node Cache 88
4.7 Batch-Delete Log 88

5 SDOS: The Key-Cascade Implementation 91
5.1 System Architecture 93
5.2 Swift Integration 96
5.2.1 Swift Storage 97
5.2.2 Swift Operations 99
5.3 Authorization and Multi-Client Support 101
5.4 Multi-threading in SDOS 103
5.4.1 Parallel Request Processing 103
5.4.2 Implementation of Parallel Processing 107

5.5 Pseudo-Object API . . . 109
5.6 Performance Evaluation . . . 112
5.6.1 Q1: Proxy Impact . . . 115
5.6.2 Q2: Encryption Impact . . . 119
5.6.3 Q3: Key-Cascade Geometry . . . 121
5.6.4 Conclusion and Areas of Improvement . . . 123
5.7 Related Publications . . . 124

6 MCM: a Demonstrator Application for SDOS 125
6.1 Design Goals . . . 126
6.2 System Architecture and Components . . . 128
6.2.1 SDOS . . . 129
6.2.2 Bluebox UI . . . 129
6.2.3 Task Runner . . . 132
6.3 Use Cases and Functionality . . . 135
6.3.1 Container Creation, SDOS Configuration, and Content Management . . . 135
6.3.2 Metadata Management and Analytics . . . 138
6.3.3 Content Disposal and Batch Deletion . . . 143
6.4 Authentication and Authorization . . . 144
6.5 Cloud Modeling and Deployment Automation . . . 146
6.5.1 Trusted Enclave . . . 148
6.5.2 Executable Model in Docker Compose . . . 149
6.6 Related Publications . . . 149

7 Conclusions and Future Work 151

Bibliography 155

List of Figures 165

List of Tables 169

Definitions 171

Zusammenfassung

Diese Arbeit behandelt die Herausforderungen hinter und Lösungen für das sichere Löschen von Daten in Cloudspeichersystemen. Sicheres Löschen ist eine wünschenswerte Funktionalität für manche Nutzer, aber auch eine gesetzliche Anforderung für andere. Der Begriff sicheres Löschen beschreibt die Praktik, Daten auf eine Art und Weise zu löschen, sodass sie selbst mit forensischen Mitteln nicht wiederhergestellt werden können. Eine Betrachtung der bisherigen Methoden zeigt, dass diese für lokale Speichersysteme ausgelegt sind, und sich nicht auf moderne Cloudspeichersysteme übertragen lassen. Daher stellt diese Arbeit das Konzept des kryptografischen Löschens vor und zeigt welche Herausforderungen der praktischen Anwendbarkeit entgegenstehen. Eine Diskussion der vorangegangenen Arbeiten im Bereich kryptografisches Löschen zeigt, dass eine Forschungslücke besteht, die bisherigen Ergebnisse auf Cloudspeichersysteme anzuwenden.

Der Hauptbeitrag dieser Arbeit, die Key-Cascade-Methode, bedient diese Lücke, indem sie eine effiziente Schlüsselverwaltung für kryptografisches Löschen bietet, die auf Cloudspeichersysteme zugeschnitten ist.

Sicheres Löschen wird überall dort angewendet, wo vertrauliche Daten nach ihrer Lebenszeit vor zukünftigem Zugriff geschützt werden müssen. Das Konzept wird bei lokal gespeicherten Daten auf Laptops und PCs angewendet, sowie auf großen Netzwerk- und Cloudspeichersystemen. Diese

Arbeit fokussiert sich auf den letzteren Anwendungsfall. Unternehmen und Organisationen aus dem Gesundheits-, Verwaltungs- und Finanzsektor sind gesetzlich verpflichtet, sicheres Löschen für bestimmte Daten zu praktizieren. Einige dieser Gesetzte werden im Abschnitt “Motivation, Standards, and Regulations” dieser Arbeit diskutiert. Um dieser Anforderung zu genügen, setzen Unternehmen heute lokale Rechenzentren ein, in denen sie das sichere Löschen durch regelmäßige, physische Zerstörung alter Speichermedien erreichen. Physische Zerstörung ist eine Kernmethode des traditionellen sicheren Löschens, die sich nicht sinnvoll auf Cloudspeichersysteme übertragen lässt.

Kryptografisches Löschen beschreibt die Methode, Daten verschlüsselt auf einem nicht vertrauenswürdigen Medium zu speichern, während der Schlüssel in einer kontrollierten, vertrauenswürdigen Umgebung bleibt. Sofern Vertrauen in die Wirksamkeit der Verschlüsselung besteht, kann sicheres Löschen der Daten nun durch sicheres Löschen des Schlüssels erreicht werden. Die Verwendung von Verschlüsselung muss dabei entweder durch den Gesetzgeber, oder den Kunden erlaubt sein. Dies hängt davon ab, ob sicheres Löschen aufgrund rechtlicher oder eigener Motivation geschieht.

Die Methode des kryptografischen Löschens ist bekannt, wird jedoch noch wenig eingesetzt. Die Methode kann überall dort eingesetzt werden, wo Datenverschlüsselung betrieben wird. Einige Hersteller verschlüsselter Speichersysteme bieten kryptografisches Löschen an, implementieren diese Funktion jedoch analog zur physischen Zerstörung: Nur das sichere Löschen eines Hauptschlüssels ist möglich, womit alle gespeicherten Daten verloren gehen. Ist eine feinere Granularität gewünscht, so sind teure Kopier- und Neuverschlüsselungsoperationen nötig.

Um kryptografisches Löschen in einem Cloudspeichersystem sinnvoll einzusetzen, ist eine feine Granularität der Löschoperation nötig. Nur so können Kunden das System kontinuierlich benutzen und regelmäßig alte Objekte sicher löschen, zum Beispiel nach Ablauf von Rückhaltefristen. Diese feine Granularität kann durch die Wahl einer geeigneten Methode zur Erzeugung und Verwaltung der Schlüssel erreicht werden. In der Literatur finden sich einige wenige Konstruktionen, die dies erreichen, welche im Abschnitt “Re-

lated Work” dieser Arbeit diskutiert werden. Als Weiterentwicklung dieser Ergebnisse stellt diese Arbeit die Key-Cascade-Methode vor, welche Datenstrukturen und Prozesse beschreibt, die kryptografisches Löschen in einem Cloudspeichersystem effizient ermöglichen. Sie erlauben eine kontinuierliche Nutzung des Systems sowie effizientes, selektives Löschen.

In dieser Arbeit präsentiere ich die konzeptionellen und mathematischen Eigenschaften der Key-Cascade-Methode. Um eine praktische Anwendung in einem Cloudspeichersystem zu ermöglichen, präsentiere ich Erweiterungen zu dem Kernkonzept, welche die Nutzbarkeit verbessern und eine reibungslose Integration in bestehende Anwendungen erlauben. Mit SDOS liefert diese Arbeit eine vollständige Implementierung der gezeigten Konzepte und Erweiterungen. Die Gestaltung von SDOS als API Proxy ist ein Alleinstellungsmerkmal unter den bestehenden Lösungen und erlaubt die modifikationsfreie Anwendungsintegration. Basierend auf SDOS zeige ich Ergebnisse von Leistungsmessungen, die belegen, dass die vorgestellten Konzepte praktisch nutzbar sind.

Abschließend präsentiere ich mit MCM eine Demonstratoranwendung für SDOS, die interessierten Lesern einen schnellen Einstieg in das System erlaubt und als Beispiel dient, wie SDOS zusammen mit einer Cloudanwendung eingesetzt werden kann.

Abstract

This thesis discusses the practical implications and challenges of providing secure deletion of data in cloud storage systems. Secure deletion is a desirable functionality to some users, but a requirement to others. The term secure deletion describes the practice of deleting data in such a way, that it can not be reconstructed later, even by forensic means. This work discuss the practice of secure deletion as well as existing methods that are used today. When moving from traditional on-site data storage to cloud services, these existing methods are not applicable anymore. For this reason, it presents the concept of cryptographic deletion and points out the challenge behind implementing it in a practical way. A discussion of related work in the areas of data encryption and cryptographic deletion shows that a research gap exists in applying cryptographic deletion in an efficient, practical way to cloud storage systems.

The main contribution of this thesis, the Key-Cascade method, solves this issue by providing an efficient data structure for managing large numbers of encryption keys.

Secure deletion is practiced today by individuals and organizations, who need to protect the confidentiality of data, after it has been deleted. It is mostly achieved by means of physical destruction or overwriting in local hard disks or large storage systems. However, these traditional methods of

overwriting data or destroying media are not suited to large, distributed, and shared cloud storage systems.

The known concept of cryptographic deletion describes storing encrypted data in an untrusted storage system, while keeping the key in a trusted location. Given that the encryption is effective, secure deletion of the data can now be achieved by securely deleting the key. Whether encryption is an acceptable protection mechanism, must be decided either by legislature or the customers themselves. This depends on whether cryptographic deletion is done to satisfy legal requirements or customer requirements. The main challenge in implementing cryptographic deletion lies in the granularity of the delete operation. Storage encryption providers today either require deleting the master key, which deletes all stored data, or require expensive copy and re-encryption operations. In the literature, a few constructions can be found that provide an optimized key management. The contributions of this thesis, found in the Key-Cascade method, expand on those findings and describe data structures and operations for implementing efficient cryptographic deletion in a cloud object store.

This thesis discusses the conceptual aspects of the Key-Cascade method as well as its mathematical properties. In order to enable production use of a Key-Cascade implementation, it presents multiple extensions to the concept. These extensions improve the performance and usability and also enable frictionless integration into existing applications. With SDOS, the Secure Delete Object Store, a working implementation of the concepts and extensions is given. Its design as an API proxy is unique among the existing cryptographic deletion systems and allows integration into existing applications, without the need to modify them. The results of performance evaluations, conducted with SDOS, show that cryptographic deletion is feasible in practice.

With MCM, the Micro Content Management system, this thesis also presents a larger demonstrator system for SDOS. MCM provides insight into how SDOS can be integrated into and deployed as part of a cloud data management application.

Introduction

The following chapter contains two main parts: The first gives an overview of the contributions and content in this thesis. The Second part introduces the background and motivation for this work as well as the research question and requirements.

1.1 Contributions and Outline of the Thesis

In the following, I give an overview of the contents of this thesis by firstly stating the three main contributions of this work. I then briefly summarize the content of each chapter and point out where each contribution is presented.

The three main contributions of this work are:

Contribution C1: The Key-Cascade method. The Key-Cascade describes data structures and operations for managing large numbers of encryption keys for cryptographic deletion. Using a tree structure, it establishes a hierarchy of encryption keys so that secure deletion of a root key translates to cryptographic deletion of selected leaf keys. This Key-Cascade method solves the same key management problem as the

two preliminary approaches presented in Chapter 1, but does so in a manner that satisfies performance and storage capacity requirements.

Contribution C2: SDOS. The Secure Delete Object Store is an implementation of the Key-Cascade method. SDOS is based on the object store Swift by the OpenStack project and is designed as an API proxy for the Swift REST protocol. With this design aspect, it is possible to transparently use SDOS with any unmodified Swift server, or cloud service, as well as any existing Swift client or application.

I implemented this prototype in order to conduct performance evaluations of the Key-Cascade method but also as a first step towards a productively usable implementation of cryptographic deletion. For this reason, SDOS contains extensions that provide multi-threaded execution, multi-client and multi-tenant support, as well as integration with native Swift and OpenStack authentication providers. Finally, SDOS contains a batch deletion function and integration with hardware key stores that make production use feasible.

Contribution C3: MCM. The Micro Content Management System is a demonstrator application for SDOS. MCM contains a Web-based user interface and services for extracting and visualizing object metadata. The user interface supports uploading, downloading, and listing of objects and also allows configuring the Key-Cascade parameters for SDOS. It contains an interactive visualization of the Key-Cascade data structures and allows the user to interact with the hardware key store, if used.

I included the extraction and visualization of object metadata in order to show how cloud providers can still offer advanced services even when customers encrypt their data. In MCM, users can decide which metadata they want to extract and store unencrypted, so that further processing of this data by the cloud provider is possible.

MCM is designed as a multi-component cloud application that can be deployed in multiple topologies. As part of MCM, I provide a fully automated deployment of all the components, including SDOS and

the Swift Object Store.

This thesis is structured as follows:

In Chapter 1, I first present the contributions and structure of this thesis. I motivate the topic of secure deletion by showing regulations and standards as well as existing practices and use cases. I then introduce the concept of cryptographic deletion and show how it can be applied in practice. This shows that cryptographic deletion is an application of data encryption that uses a special key management. Two preliminary approaches to managing the encryption keys highlight the challenges and help derive requirements for a satisfactory solution. Based on these circumstances, I then state my research question together with requirements for a satisfactory solution. Finally, I present an adversary model which describes properties of an adversary against which the solution should hold.

In Chapter 2, I present background information on the practice of secure deletion, technologies and methods that I use in my contributions, as well as related technologies and related work. I first introduce the topic of secure deletion by discussing its application in practice and the methods currently used. This covers applied methods for storing and deleting sensitive information such as physical destruction of storage media and overwriting of data. I then introduce technologies and methods that are used in the contributions of this thesis. This includes chains of perpetually encrypted keys and specialized cryptographic hardware. Trusted Platform Modules (TPM) are an example of such hardware devices that are found in most modern personal computers and servers. In order to point out possible areas where cryptographic deletion could be applied, I show candidate applications where data encryption is used. This includes file and disk encryption, cloud data storage services that use encryption as well as encryption in database systems. Finally, I discuss related work in the area of cryptographic deletion and present the state of the art on which this thesis builds.

In Chapter 3, I present the first contribution **(C1)** of this thesis: The concepts behind the Key-Cascade method. The Key-Cascade method describes data structures and processes for managing large numbers of encryption

keys. The description of the Key-Cascade method is started by introducing the data structure and relations between the encryption keys. Following this, I discuss the operations on this data structure and how these operations relate to storing, retrieving, and deleting encrypted data. I then show the underlying mathematical properties and provide complexity considerations.

In Chapter 4, I present extensions to the core Key-Cascade concept. These extensions are not strictly necessary for providing a working Key-Cascade implementation, as they cover operational, performance, and integration aspects. They are, however, useful extensions that help build a practically usable implementation. In this chapter, I first discuss the problem of resizing an existing Key-Cascade in order to increase or decrease the object key capacity. Because the Key-Cascade is based on a static tree, it has a fixed capacity and does not expand and shrink dynamically. For this reason, it is useful to have methods for increasing or decreasing its size. This extension is only discussed conceptually and not implemented in the prototype since the need to resize can be avoided in practice. The deletable key is the topmost key in a Key-Cascade data structure and allows decryption of the data. It is also the only key which must be securely deletable, because the Key-Cascade then transfers this property to all further keys and encrypted objects. For this reason, I discuss how a deletable key can be realized in practice and show three different sources (or types) of deletable keys. An index for object identifiers is another extension I present. It allows using arbitrary names or identifiers for the objects whose keys are stored in the Key-Cascade. Without this extension, only sequential numbers can be used as object identifiers. The last three extensions cover performance aspects in a practical implementation. I introduce a free-list that is dynamically created and used during run time. This list allows the system to quickly find a free leaf position for inserting new object keys. This is followed by a node cache which improves performance by providing faster access to Key-Cascade nodes. The last extension, a batch delete log, adds a new mode of operation to the Key-Cascade implementation. It allows deferring operations on the data structure to a later time, so that multiple deletions can be processed together. Secure delete operations for different objects often have overlap

since different paths from the root of a tree to its leaves always share some nodes and edges. This extension provides a large performance improvement and also matches the common use case where cryptographic deletion is not done immediately, but periodically.

In Chapter 5, I present the second contribution **(C2)** of this thesis: a practically usable implementation of Contribution C1, the Key-Cascade method. This system can be used with an object store, specifically with the Swift Object Store by the OpenStack project. The prototype is called SDOS (the Secure Delete Object Store) for this reason. In this chapter, I cover the implementation of the core Key-Cascade concept. Implementation aspects of the extensions of Chapter 4 are covered in that chapter. I first present the high-level architecture of the system and show how it is integrated with the object store and its clients. This is followed by a walk-through of the operations it supports. I discuss each operation provided by the object store interface and elaborate on how they relate to operations on the Key-Cascade data structure. After the basic structure and operations are covered, I discuss implementation aspects that are relevant for integrating a Key-Cascade into a production environment. The example of SDOS shows that the Key-Cascade concept does not prohibit a practically usable implementation. These implementation aspects cover the parallel execution of operations as well as integration with an authorization provider and support for multiple clients, or tenants in a single SDOS instance. Finally, I discuss the setup and results of performance evaluations conducted with the SDOS prototype. These results show both the performance of the prototypical implementation as well as the overhead implied by the Key-Cascade operations.

In Chapter 6, I present the third contribution **(C3)** of this thesis: MCM (the Micro Content Management System), a demonstrator application for SDOS and cryptographic deletion. I first discuss the design goals for MCM as well as its functionality and use cases. This is followed by a detailed description of the individual components of MCM. I then present different possible scenarios how the MCM components, including SDOS, can be integrated with a cloud object store and its clients. Finally, I present how MCM can be realized with a cloud application modeling approaches: An executable,

imperative model in Docker Compose. I also briefly discuss a conceptual, declarative model in TOSCA.

Finally, I draw conclusions and summarize the work in Chapter 7.

1.2 Motivation, Standards, and Regulations

The secure, irrevocable deletion of data is a useful feature for some users and mandatory for others. I discuss the current approaches and methods for secure deletion in Chapter 2 and, in the following, I point out the motivation for secure deletion in three different groups of users. These groups are: governments and institutions, corporations and enterprises, and individuals.

Governments and institutions frequently handle sensitive personal information about their citizens. For this reason, they often impose strict rules onto themselves about how such information has to be stored and managed. Rules like the following can be found in most jurisdictions and they mandate that some effort has to be made in order to assure the secure deletion of data:

The United Kingdom National Police has strict rules as to when personal records are to be assuredly deleted from national police systems [Ber15].

The Recordkeeping Support Unit of the Archives of Ontario has rules in place that "...records that may contain personal information or other sensitive or confidential information must be disposed of in such a manner that the information cannot be reconstructed ..." [oOnt08]. Their *Archives and Recordkeeping Act* from 2006 further requires that "all public records be disposed of by following the conditions set out in approved records schedules".

Corporations and enterprises not only have an interest in being compliant with rules and regulations, but also in the secure deletion of content no longer required in order to avoid "smoking guns" in their data stores [Con09].

The EU data protection reform *General Data Protection Regulation (GDPR)* came into law on May 25th 2016 and strengthens the individual person's "right to erasure" [PC16]. Prosecution of GDPR violations is however sus-

pended until May the 28th of 2018, to allow data processing enterprises more time to become compliant.

European enterprises are particularly anxious about the right to erasure (formerly “right to be forgotten”) clause in the new GDPR and will have to seek out storage solutions that allow secure deletion in order to be compliant. Cloud service corporations, aiming to sell their services to European businesses, are especially sensitized in light of the GDPR. They bear responsibility for properly managing their customers’ data. Noncompliance or negligence leads to fines. GDPR violators will be fined EUR 20 million or 4% of annual global turnover, whichever is greater. Additionally, each customer of the affected product is eligible for a EUR 2,500 compensation, independent whether harm was caused, or not. The scale and severity of these fines for noncompliance with the GDPR, as well as the ensuing reputational damage, present a risk that will influence data storage selection in the future [BAV15].

In the US, transitory records (any data collected during a business transaction) are public records (i.e., the government may access them) and may have to be reviewed and disclosed in response to a formal request for information, even if companies kept them when they could have destroyed them [oOnt08].

By not keeping more than required, companies will have less information to search and review if served with a document request during litigation. This will save a company time and money during litigation since there will be fewer materials for its attorney to review and produce.

The adoption of cloud services by corporations has put a focus on security protocols. Financial services firms, in general, are ready to adopt cloud services, as long as those services adhere to necessary security protocols and other requirements [Day14]. Requirements such as data leakage prevention are now topping the information security priorities of organizations [BAV15].

According to Schafer, even organizations that do not require secure deletion should, at least, use data encryption to minimize chain-of-custody security risks [Sch14].

To summarize, corporations who are not compliant with regulations, or negligent with data security, risk large fines, reputational damage and

disgruntled customers [Sch09; Sch11; Pid11; Ost09].

Individuals are mostly concerned with privacy of their own personal information and are not bound by regulations like the other two user groups. Additionally, the ability to securely delete data can increase trust in cloud services and increase adoption, especially among customers who are aware of security issues.

Individuals already rely heavily on cloud backup and synchronization services for mobile devices. The risk behind outsourcing the storage of personal data became obvious to many customers of Apples iCloud backup service in the prominent data leak of August 2014, where much of the leaked data was presumably deleted by its owners [McC14].

1.3 The Concept of Cryptographic Deletion

Cryptographic deletion refers to the practice of using encryption in order to provide secure deletion in a storage location, that does not offer secure deletion on its own. The general principle is shown in Figure 1.1: Data are encrypted and stored separately from their encryption key. Two different storage locations are used. A trusted storage location as well as a larger, untrusted location. The trusted storage location has the capability to provide secure deletion so that any data deleted from the trusted location can not be recovered later. This *secure delete* property of the trusted storage is then transferred to the stored data by means of encryption: If the key can not be recovered, neither can the encrypted data.

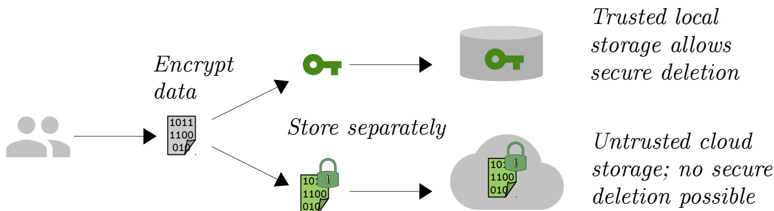

Figure 1.1: General principle of cryptographic deletion.

At least two conditions must be met in order for cryptographic deletion to work:

1. The cryptographic algorithm must be effective. Cryptographic deletion assumes that any data deleted from the untrusted storage location can be recovered. An adversary can therefore recover this encrypted data and try to break the encryption. For this reason, a cryptographic algorithm must be chosen that is expected to have strong security in the foreseeable future. This condition is shared with any solution that uses cryptography. Industry standards like FIPS specify certain algorithms, key sizes, and implementation procedures that allow implementing certifiable encryption systems. Whether encryption is an acceptable protection mechanism, must be decided either by legislature or the customers themselves. This depends on whether cryptographic deletion is done to satisfy legal requirements or customer requirements.
2. Secure deletion on the trusted storage must be effective. Secure deletion of the encrypted data is derived from secure deletion of the encryption key. If a deleted key can be restored, data, assumed to have been deleted, can be decrypted. I present possible implementations of such a trusted storage in Section 4.3.

1.4 Definition of Secure Deletion

In the following, I define what the term *secure deletion* means in the context of cryptographic deletion in a cloud storage system. In general, data are securely deleted if they can not be reconstructed by an adversary. For this reason, I discuss the capabilities of such an adversary in the following.

An adversary model is an abstract way of describing security properties of a system. Instead of directly describing those properties, an adversary model describes the capabilities of the most powerful adversary (or attacker) against whom the system holds. In other words, it describes which capabilities are not sufficient to circumvent the security measures.

The cloud storage scenario I use in this work involves the following three parties: A user of the system, a cloud provider offering storage, and an adversary aiming to recover the user's deleted data. They have the following capabilities and limitations in this scenario:

The user has a small trusted storage location that provides secure deletion. Any data deleted from this location is assumed to be unrecoverable. Such a location can be provided by Hardware Security Modules (HSM) or Trusted Platform Modules (TPM). The user stores the encryption keys for cryptographic deletion in this location. Therefore, deleted keys can not be reconstructed by an adversary who gains access to the trusted storage.

Finally, the user wants to store a large volume of sensitive data that exceeds the capacity of the trusted storage location. This data consists of small enumerable units (e.g. files) which need to be stored, retrieved, and securely deleted on an individual basis. The cloud provider offers a data storage service that fits the user's capacity requirements. But neither does the provider offer the possibility to securely delete said data, nor does the user trust the provider to offer such a service. The user therefore uses this cloud storage only to store encrypted data.

Adversary limitation: The adversary is computationally bound. This limitation must be imposed on any adversary to a cryptographic solution. The reason is that encryption relies on the limited computational resources in the real world. A theoretical adversary with unlimited resources could determine the encryption key just by brute force.

Adversary capability: The adversary may have unlimited access to the cloud provider's systems at all times. This capability is another way of stating that the cloud provider is untrusted. It means that any data that is or was stored on the cloud may be available to the adversary.

Adversary capability: The adversary gains access to the user's trusted systems at a certain time. Any data that is currently stored on the trusted location is therefore available to the adversary. However, because of the secure delete property of the trusted storage, no previously deleted data can be restored by the adversary. This is an important

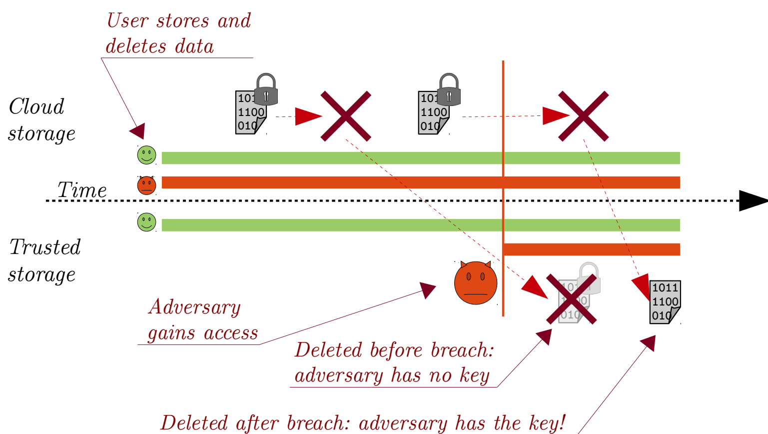

Figure 1.2: Representation of the adversary model.

aspect for cryptographic deletion; the adversary may have access to all currently stored encryption keys. The cryptographic deletion method must therefore prevent that accessing the current keys allows decrypting previously deleted data.

In Figure 1.2, I illustrate the effects of these capabilities and limitations. Above the time-line, it is shown that both the user and the adversary have access to the cloud resources. The user stores two encrypted files on the cloud and the two encryption keys in the trusted storage. The user then deletes the first file from the cloud and the respective key for that file from the trusted storage.

Below the time-line, the access to the user's trusted storage is shown. The user had always access. They stored two encryption keys and deleted the first one as described above. At a certain time, the adversary gains access to the trusted storage. This is where encryption with cryptographic deletion and regular data encryption differ: This access must not enable the adversary to recover any data that was previously deleted. In regular encryption solutions, access to the current master key allows decrypting previously deleted data.

It is shown that the adversary is able to recover both encrypted files from

the cloud. It is not even necessary to assume that recovery of the files is possible for the adversary. They may also have copies of the data from the past. The adversary is however unable to decrypt the first file because the encryption key was securely deleted from the trusted location. This is not the case for the second file. The key for that file was present in the trusted storage when the adversary gained access. Therefore, the second file can be decrypted even if the user deletes the key after the fact.

The goal of secure deletion is purely to prevent the adversary from recovering previously deleted data in the aforementioned scenario. Other security aspects (e.g. confidentiality of stored data, access protection) might be achieved, but are not the focus of this work. The important difference is that systems without secure deletion do not have this property. I discuss the details in Section 2.4.3, but the general reason is that, in regular encryption, a current master key is also valid for previously deleted files. Even generating a new master key yields a key that is valid for deleted data.

To summarize: cryptographic deletion must manage the encryption keys in such a way that access to the current keys does not allow decryption of previously deleted encrypted data.

1.5 The Key Management Problem

Cryptographic deletion is an application of data encryption that employs a specialized key management method. In the following, I discuss this key management problem. It shows why a specialized key management method is needed and why cryptographic deletion in regular data encryption solutions is not feasible.

For this, I present two preliminary approaches to applying cryptographic deletion to a cloud storage system. They are straight-forward ways of doing cryptographic deletion in an encrypted storage system and both have prohibitive downsides. These approaches work in most encrypted storage systems, albeit only as a last measure due to their downsides. Still, some providers of encrypted storage solutions recommend them (see Sec-

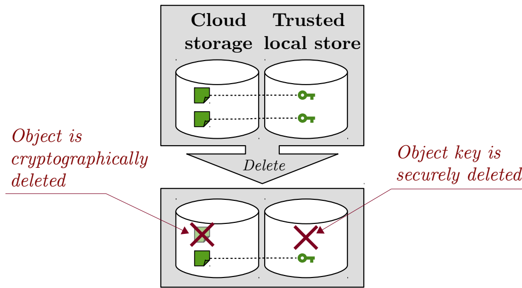

Figure 1.3: The individual key approach: one separate key per object.

tion 2.4.3). The shortcomings of the preliminary approaches also help specify the requirements in Section 1.6.

I design and implement the contributions in this thesis around the data models and interfaces of an object store. For this reason, I use the term *object* as the unit of data that gets stored, retrieved, and deleted in the following examples. Objects are comparable to files in a file system, they are addressable by an object identifier. The actual data model of an object stored is discussed in detail in Chapter 5. For the following conceptual discussion, it is enough to assume a single, flat hierarchy, i.e., a list, of objects with consecutive numbers as identifiers.

As soon as we want to store multiple objects and provide secure deletion on the granularity of individual objects, cryptographic deletion faces the key management problem:

The **individual key approach** is the first preliminary approach. It uses an individual encryption key for each data object, as shown in Figure 1.3. In this scenario, the user generates a new encryption key for each new object and stores this key locally in the trusted storage location prior to storing the encrypted object in the cloud. Therefore, the overhead for writing a new object consists of generating and storing the encryption key as well as encrypting and storing the object. The overhead for reading an object consists of retrieving the key and the object and the decryption of the object. The overhead for the delete operation consists of only deleting the locally

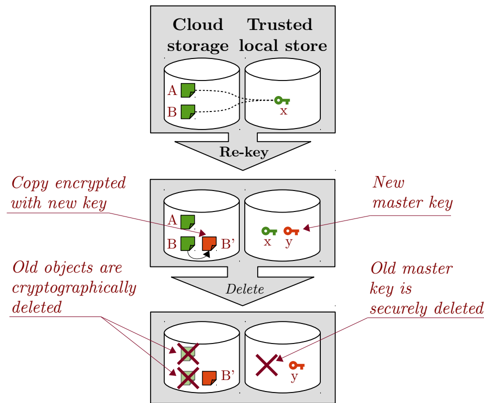

Figure 1.4: The master key approach: a single key for all objects. The re-keying operation allows selective deletion.

stored encryption key.

The individual key approach has minimal computational overhead for all involved operations, but a prohibitively large storage overhead on the trusted storage, since it has to store one encryption key locally for each object stored remotely on the cloud. Especially in scenarios with many small objects, the required local storage volume reaches the size of the cloud storage. This defeats the purpose of using a cloud-based storage system.

The second preliminary approach, the **master key approach**, is illustrated in Figure 1.4. It is aimed at minimizing the required storage capacity for the encryption keys. In this approach, I only use a single master key that needs to be stored in the trusted storage. This key is used for encrypting all stored objects. In order to delete individual objects while keeping others, I introduce the following re-keying operation:

Definition 1.1 (Re-keying operation)
Objects A and B are encrypted with Key x. The objects are stored in an untrusted location, the key is stored in a trusted location which can provide secure deletion. I want to delete Object A, but keep Object B. This is achieved by re-keying B: generate and store a new encryption key y, retrieve and decrypt B with x,

encrypt and store B' with y, securely delete x.

This operation allows using a single master key and still offers a deletion granularity of the object level. When an object gets deleted, I re-key the remaining objects with a new master key. Now, writing a new object only has the overhead of encrypting the object (assuming the master key is already present). Reading an object has the same overhead as before, i.e., retrieving the key and decrypting the object. But the delete operation now has the overhead of generating and storing a new master key, deleting the old master key, and re-keying all remaining objects. The overhead on the trusted storage is low with this approach, only a maximum of two encryption keys needs to be stored at any time.

This approach drastically reduces the required trusted storage capacity, at the expense of executing the re-keying operation for all remaining objects every time I want to securely delete a single object. This second approach therefore has a prohibitively expensive delete operation.

To summarize, the individual key approach has a prohibitively large storage requirement on the trusted storage, while the master key approach has a prohibitively expensive delete operation. In Table 1.1, I compare the numbers of both approaches. In this example, the object store has 16,777,216 objects encrypted with the AES256 algorithm that uses a key size of 32 bytes. The additional operations are calculated for deleting only a single object. The Key-Cascade method presented in this thesis solves these issues and provides a balanced approach.

	Individual key	Master key
Volume of trusted storage	**512 MB**	64 bytes
Volume of keys stored on cloud	0	0
Add. operations for deleting	1	**16,777,215**

Table 1.1: Storage and computational overhead comparison for preliminary approaches. Using an object store with 16,777,216 objects.

1.6 Problem Statement and Requirements

In the following, I give the application context, the research problem, as well as the requirements for a satisfactory solution.

The application context, to which the proposed solution is applied, is illustrated in Figure 1.5. First, two trust zones are defined. A trusted environment is present on the user side so that a trusted storage location is provided. The untrusted zone is given by a cloud storage service. Assuming this zone to be untrusted renders the requirement of trusting the cloud provider unnecessary.

In the trusted zone, I distinguish between a trusted gateway and multiple clients. A simplified solution consists of only a single client that contains the trusted gateway's components. However, separating the client from the trusted gateway yields certain benefits that make the solution more flexible and applicable in practice:

- Clients need not be aware of the data encryption and key management for cryptographic deletion. This enables the transparent use of the solution with existing clients or existing applications.
- Multiple clients can be supported without the need for a distributed key management approach.

I consider the following three requirements necessary for a satisfactory

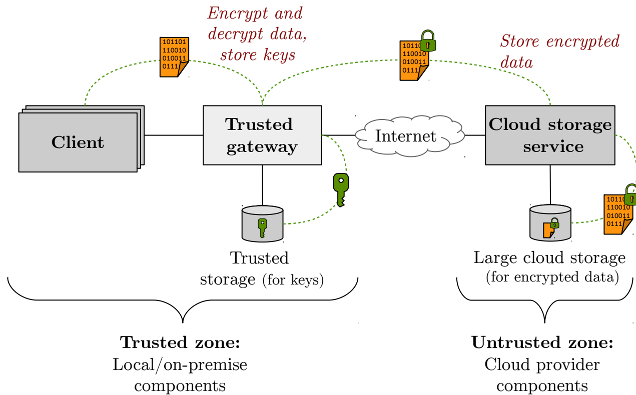

Figure 1.5: System overview showing local and cloud components.

solution in the above application context:

Requirement 1.1 (Small trusted storage volume)
The amount of data stored on the trusted gateway has to be small.

This requirement is derived from the practical implementation of the trusted storage location. Specialized cryptographic hardware with integrated hardware key stores provide the highest level of security, as they offer assured deletion of stored keys. These devices can only store small numbers of encryption keys (refer to Section 2.3). In general, the smaller the amount of data, the easier it is to store and delete in a secure way.

Requirement 1.2 (Acceptable overhead)
The computational overhead for the secure delete operation has to be small.

There is always a certain overhead implied by performing cryptographic deletion because it adds additional steps to the process of deleting data. A low overhead is therefore a desirable property of any solution.

Requirement 1.3 (Object-level granularity)
The granularity of the secure delete operation should be on the object level.

Without requiring a fine granularity, a naive approach like the above master key approach can always be used. Therefore, a research problem is only given when requiring such a fine granularity. I also consider this a necessary requirement for a practically usable system. Continued use of the system is only feasible if individual objects can be deleted, while others are kept.

Chapter 2

Background and Related Work

In this chapter, I present background information on the practice of secure deletion, technologies and methods that I use in my contributions, as well as related technologies and related work.

I first introduce the topic of secure deletion by discussing its application in practice and the methods currently used. These cover applied methods for storing and deleting sensitive information such as physical destruction of storage media and overwriting of data. I then introduce technologies and methods that are used in the contributions of this thesis. This includes chains of perpetually encrypted keys and specialized cryptographic hardware. Trusted Platform Modules (TPM) are an example of such hardware devices that are found in most modern personal computers and servers.

In order to point out possible areas where cryptographic deletion could be applied, I show candidate applications where data encryption is used. This includes file and disk encryption, cloud data storage services that use encryption as well as encryption in database systems. Finally, I discuss related

work in the area of cryptographic deletion and present the state of the art on which this thesis builds.

2.1 Secure Deletion

Secure deletion describes the idea that deleted data can not be reconstructed later, even by forensic means. This property is usually not present in storage systems for two reasons:

- For performance reasons, the storage system only marks deleted blocks or segments and reuses them later to store new data. Data recovery processes and forensic tools allow finding and reconstructing these presumably deleted data in many cases.
- Many storage systems use logs and journals in order to maintain performance and data consistency. This creates secondary copies of the data that are not considered during deletion.
- Especially large storage systems replicate and duplicate data for performance, availability, and reliability reasons. These copies must be considered in the process of securely deleting data. This is often difficult or impossible, especially with outsourced, shared storage solutions.

There are however solutions to the problem since secure deletion is a requirement in some cases. I discuss some of these cases in Section 1.2. In the following, I give an overview of current approaches to secure deletion and also show the reasons why they are not applicable in the outsourced cloud storage scenario.

2.1.1 Classification of Approaches

The currently practiced approaches to secure data deletion can first be grouped into physical destruction and logical deletion of data, as shown in Figure 2.1. Physical destruction as well as logical overwriting focus on the destruction of specific bits of stored information and therefore rely on physical or block-level access to the corresponding storage device.

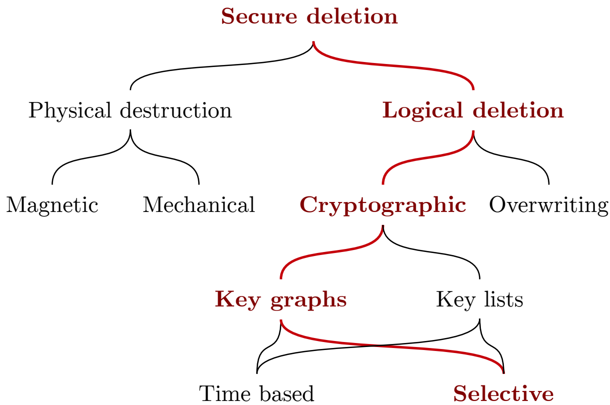

Figure 2.1: Methods to secure data deletion. The cryptographic deletion method presented in this thesis follows the highlighted path.

Physical data destruction is a straight-forward approach to secure data deletion. The operating principle is based on physically destroying the storage media containing the data. Common destruction mechanisms include mechanical (e.g. shredding) and magnetic destruction. They require the destruction of entire storage media and thus have a coarse granularity. They also do not take into account any copies of the data on other media. Due to their operating principle, they require direct physical access to the storage media that holds the desired data objects and render the media unusable. Because of these limitations, physical destruction as a secure deletion mechanism is undesirable.

Logical data destruction refers to a group of mechanisms that leave the storage medium intact and do not require physical access. It can be further separated into cryptographic deletion and overwriting. The principle of overwriting is based on block-level access to the storage medium and simple overwriting or erasing of data records using regular read/write operations [Gut96; GS03].

Both physical destruction and logical overwriting are not well suited for cloud storage scenarios. Disks are typically shared between multiple cus-

tomers and even applications, which makes physical destruction impractical. Multiple layers of abstraction and virtualization separate the application from the actual disk, which is problematic for both physical destruction and logical overwriting. Identifying the disk that should be destroyed, or the blocks that should be overwritten, becomes difficult to impossible.

Cryptographic deletion describes the concept of deleting data by keeping it in an encrypted form and removing any access to the encryption keys (see Section 1.3). Cryptographic deletion methods mainly differ in two aspects: Firstly how they store and manage the encryption keys and secondly how the deletion ob objects is decided upon.

Deletion on the granularity of groups of objects, or even individual objects, is usually required in any practical system. This means that multiple encryption keys are necessary in order to provide this granularity. A mechanism for managing these keys for cryptographic deletion is necessary in order to support any significant amount of keys. Such mechanisms commonly use either list-like data structures or graphs. Examples for list-like structures are given with the preliminary “individual key” approach (Section 1.5) as well as in the discussion of the related work (Section 2.4). The preliminary “master key” approach (Section 1.5) is not considered using a graph, since the master key directly encrypts the objects. The main contribution of this thesis, the Key-Cascade method, uses a graph of keys and allows selective deletion of individual objects.

Time-based vs. selective deletion is the last aspect of the classification in Figure 2.1. Selective deletion allows users to securely delete individual objects (or groups of objects) without affecting the rest of the managed data. Time-based approaches implicitly delete old objects after a certain period. Examples for such approaches are given in Section 2.4. Selective approaches, like the Key-Cascade method, can always emulate the time-based behavior, but may be less efficient in doing so.

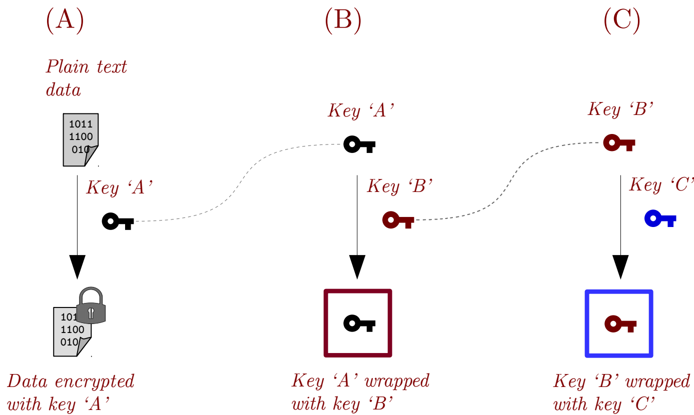

Figure 2.2: Simplified wrapped key and chain of wrapped keys.

2.2 Wrapped Keys and Chains of Keys

Key wrapping refers to the process of encrypting encryption keys in order to secure them for storage or transport [Nat01]. Multiple stages of wrapped keys form a chain of key dependencies which enables properties beyond just securing the keys. A simplified version of key wrapping is shown in Figure 2.2. The arrows represent the application of a symmetric encryption algorithm (i.e., AES). Key A is first used to encrypt data. By encrypting Key A with Key B, A gets wrapped (i.e., encrypted) by B. The wrapped key can now only be read with knowledge of the wrapping Key B. Now the encrypted data and wrapped Key A can be stored on untrusted storage. Any plain-text copy of Key A gets discarded and then only Key B allows decrypting A and therefore decrypting the data.

Wrapping B as well with a new Key C forms a chain: $C \mapsto B \mapsto A$. Key C now allows access to the stored wrapped Keys B and A as well as the encrypted data. Chains establish dependencies between keys and are used to create encryption schemas. The Key-Cascade is such an encryption schema based on branching chains that form a tree. In the following, I give an example of a simple encryption schema in order to highlight the properties of chains.

2.2.1 Example: Practical Disk Encryption

In this scenario, I want to encrypt many individual files stored on disk and use a single key (master key) for accessing them. A trivial encryption schema would use the master key (MK) to encrypt each file, as exemplified by the preliminary “master key” approach (Section 1.5). But I want the schema to have the following properties as well:

- The master key must be exchangeable with little effort.
- It must be possible to have secondary (backup) master keys.
- Each file must be encrypted with an individual key in order to minimize key fatigue and leaking[1].

Chains allow achieving these properties with the following encryption schema: Create an individual file encryption key (FEK) for each file; this creates n FEKs for n files. Wrap the n FEKs with a key encryption key (KEK). Finally, wrap the KEK with the master key. This schema implements a chain as shown in Figure 2.2 where multiple Keys A exist (the FEKs), where Key B is the KEK and Key C is the master key.

It is now possible to change the master key by unwrapping (decrypting) the KEK and wrapping it with a new master key. Secondary master keys can be generated by creating a copy of the KEK wrapped with the secondary MK. And since each file has its own FEK, the risks of key fatigue and leaking are minimized.

Wrapped keys and chains are the basic building blocks for creating encryption schemas with specific properties. Chains can be combined and form trees or graphs. This enables properties like permission management through cryptography or secure deletion as realized in the Key-Cascade.

2.2.2 Authenticated Key Wrapping and Validation

Authenticated wrapping is an extension of the simplified key wrapping explained above. It adds the capability to verify (authenticate) the key after

[1] The practical security of keys decreases with extended exposure and use [Ken05, p. 51]

unwrapping.

Unwrapping, or decrypting data, with a wrong key only leads to incorrect output of the encryption algorithm. Detecting if the correct key was used is only possible by examining the output, the decryption process itself does not express the difference. But good encryption keys are indistinguishable from random bytes; it can not easily be verified whether an unwrapped key is the correct encryption key. In other words, it is not obvious after decryption whether the correct key was used.

Message authentication is a basic concept in cryptography that can be realized in different ways [FS03, p. 25]. Its purpose is to add content integrity validation to encryption in order to achieve content authentication. If a ciphertext can be decrypted and the integrity of the result validated, it is assumed to be authenticated because only the holder of the encryption key is capable of creating such a ciphertext. Hashing is often used to add integrity validation to encryption.

Authenticated key wrapping[1] is possible for example by: calculating a hash of the key that should be wrapped, appending that hash to the key and then wrapping key and hash. After unwrapping, the resulting key is hashed again and then the hashes are compared. A hash mismatch indicates that either the unwrapping key was incorrect or that the ciphertext was tampered with. Note: in such applications, known cryptographic hash functions (e.g. MD5, SHA) are used. For this reason, the hash function does not need to be stored. The two parties must only know which of the known cryptographic hash functions they use.

If authentication is not required but only validation that the correct unwrapping key was used, then a different approach can be taken. In other words: If the user only wants to know if the correct decryption key was used, but does not care if the message was tampered with or forged, they are only interested in validation and not authentication. Validation can be achieved by simpler means than authentication.

Instead of appending a hash to the key before wrapping, a sequence of

[1] Specialized wrapping algorithms for encryption keys exist as well [Nat01].

known bytes is used. These bytes constitute a known header whose presence can be checked after unwrapping. Such a header can be a string of letters or numbers specified by the author of the encryption software. As stated above, the unwrapped encryption key can not be identified, but the presence of the header before or after the key's bytes asserts that decryption was successful. In SDOS, I use this header approach for all wrapped keys and encrypted nodes: before encrypting a key or node, the string "`SDOS_ENC`" is prefixed to the key's bytes. After decryption, SDOS checks the presence of this string and so learns whether decryption was successful.

2.3 Cryptographic Hardware

Specialized cryptography hardware exists for two purposes today: For improving performance and for improving security.

Cryptography accelerators are found in many computing platforms today due to the ubiquity of encryption. Cryptography heavily relies on a very small number of complex algorithms, so that in-hardware implementations of those algorithms can accelerate numerous cryptographic tasks. The most widely used algorithm as of 2017 is the Advanced Encryption Standard (AES) symmetric cipher.

Symmetric ciphers are used for actual (storage or transmission) data encryption, compared to asymmetric ciphers, which are mostly used for signing, verifying, and key exchange. This is because symmetric ciphers are orders of magnitude faster to compute than asymmetric ones. For this reason, the most common cryptographic task today is encrypting and decrypting data with the AES algorithm.

In 2008, Intel published a specification for extending the x86 CPU instruction set with instructions for hardware accelerated AES. Both Intel and AMD have been shipping all their processors since 2010 with included hardware AES functionality [Gue12]. The specification was later also implemented in non-x86 architectures and is now available in Sun SPARC, IBM Power7 and the ARM architecture [Yod13].

Figure 2.3: IBM 4767 PCIe HSM with secure enclosure [IBM].

2.3.1 Improving Security with Cryptographic Hardware

A different type of cryptography hardware comprises devices that extend the security or add new security features to a computing platform. These devices contain a small secure storage area, usually for cryptographic keys, as well as a processor.

Hardware Security Modules (HSM) and Trusted Platform Modules (TPM) represent the most widely used devices in this category. HSMs are general purpose cryptography modules that are available in a variety of configurations. These devices are mostly used in banking and payment processing applications and are often required by standards and regulations in these industries [IBM]. HSMs have a tamper-resistant hardware construction so that they securely delete or destroy their sensitive data when they detect physical manipulation, as seen in the IBM 4767[1] in Figure 2.3.

Hardware Security Modules create a trusted enclave inside an otherwise untrusted computing platform. They achieve this by their physical secu-

[1] http://www-03.ibm.com/security/cryptocards/

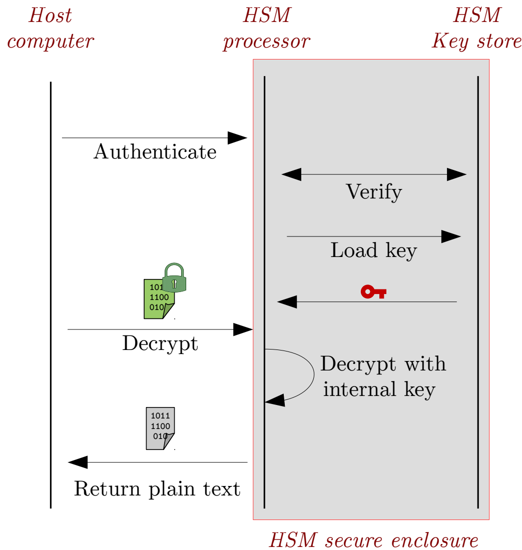

Figure 2.4: General HSM/TPM operation. Keys never leave the secure enclosure.

rity combined with the fact that master keys never leave the HSM. They contain master keys programmed by the manufacturer and can generate additional keys inside the device. These keys can only be used to perform cryptographic functions on the HSM processor inside the HSM. The HSM requires authentication before allowing any operation with the stored keys (see Figure 2.4). This creates a multi-factor security because decryption of data, or other cryptographic operations, are only possible with knowledge of the authentication key as well as physical possession of the working HSM.

Hardware Security Modules are programmable, allowing the execution of custom code inside the secure container. This functionality is used in order to prevent sensitive decrypted data from leaving the HSM. Especially wrapped-key applications (see Section 2.2) make use of this in order to load encrypted external keys into the HSM, then decrypt and use them only inside the HSM. Combined with the capability to generate and encrypt new

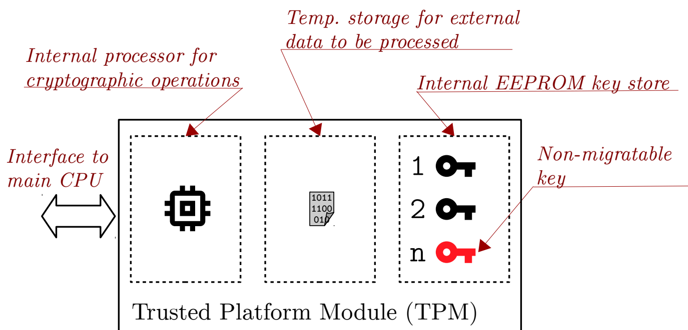

Figure 2.5: TPM internal components. Non-migratable keys can only be used inside the TPM.

keys inside the HSM, the use of custom cryptographic schemas with large numbers of keys is enabled.

Trusted Platform Modules (TPMs) are a specialized type of HSM that are used in personal computers and servers. They are already ubiquitous in those devices today and will achieve even higher distribution in the future, since Microsoft added a TPM as a mandatory hardware requirement for its Windows 10 operating system[1]. TPMs implement a specification by the Trusted Platform Group that defines core capabilities and security requirements [Gro16]. Their intended purpose is to support local disk and data encryption, as well as verified device identification.

A block-level diagram of the typical TPM hardware is given in Figure 2.5. Similar to HSMs, they contain their own processor for cryptographic operations. TPMs contain two logical storage areas on an EEPROM[2]. One is used for data, transmitted to or from the main CPU, to be processed (en/decrypted) by the TPM. The other is used as an isolated storage area for encryption keys. Keys can be specified to be "non-migratable". Such a key can never be transmitted to the main CPU, but only be used inside the TPM.

For typical data encryption with a TPM, an encryption key is stored on

[1] https://msdn.microsoft.com/en-us/library/windows/hardware/dn915086(v=vs.85).aspx

[2] Electronically erasable and programmable read only memory.

disk but wrapped (see Section 2.2) with the TPM master key. This encryption key can only be used after it was unwrapped by an authenticated TPM. The actual data de- and encryption is then done by the main CPU. For device identification, they contain an "endorsement key" that was signed by a trusted manufacturer master key. Remote services can challenge the TPM and verify the endorsement key in order to identify a certain machine. This is used for enterprise asset tracking as well as license management for digital media (digital rights management).

Just like general purpose HSMs, TPMs can be used in custom applications with the limitation that no custom code can be run inside the TPM. Only the basic cryptographic operations are supported by the processor inside the TPM [ZDB09]. The physical security and tamper resistance of TPMs is also lower than that of most enterprise HSMs.

2.4 Related Work

The topic of cryptographic deletion and its applicability to cloud storage systems, which was the focus of my research, covers three areas of interest: secure deletion, cryptographic deletion, as well as storage and data encryption systems. In the following, I discuss the related work from research and industry in these three areas.

2.4.1 Secure Deletion

As discussed in Section 2.1, cryptographic deletion is only one method to achieve secure deletion. In this section, I discuss standards and recommendations concerning applied secure deletion. I also discuss the other two approaches: physical destruction and logical overwriting.

In 1996, Gutmann published a comprehensive and widely cited report on secure deletion through overwriting [Gut96]. He evaluated the effectiveness of different methods by attempting forensic data recovery afterwards. At the time of his experiments, he found that up to 35 overwrite cycles with specialized data patterns were necessary on some media. His results became

popular as the “Gutmann method” and the 35 cycles and his data patterns are implemented in many overwriting tools[1] today.

The US Department of Defense (DoD) publication “DoD 5220.22-M” states internal procedures used for secure data deletion. It mentions physical destruction through strong electromagnets or shredders and also specifies a procedure for data overwriting. This document is often cited as authoritative and the specified procedure of overwriting data three times with zeroes has become a popular practice [BL05]. However, the DoD discontinued publishing such procedures and recommends against referring to the above mentioned document. They state that it is not their responsibility to specify or certify such procedures for the civilian public.

In 2006, the US National Institute for Standards and Technology (NIST) published a special report on secure deletion [KSSL06]. They define different levels of deletion based on the sensitivity of the contained data. These include overwriting, magnetic destruction, and physical destruction. They state that “Studies have shown that most of today’s media can be effectively cleared by one overwrite”. This is an interesting development considering the older Gutmann method and DoD standard recommend multiple overwrites. According to NIST, modern magnetic storage devices store data with such a high density that recovery is already impossible after a single overwrite.

The above mentioned NIST publication exists in multiple, revised editions; the first being the 2006 version. In this initial publication, they state that “Encryption is not a generally accepted means of sanitization.”, effectively prohibiting the use of cryptographic deletion.

However, in the 2012 revision, they state that “If strong cryptography is used, sanitization of the target data is reduced to sanitization of the encryption key(s) used to encrypt the target data. Thus, with cryptographic erase, sanitization may be performed.”. They further state “…selective sanitization, has potential applications in cloud computing and mobile devices…”, effectively recommending cryptographic deletion in cloud storage systems. They point out that it is necessary to store the encryption keys in a separate

[1] E.g. Linux wipe:
http://manpages.ubuntu.com/manpages/wily/man1/wipe.1.html

storage location that provides conventional secure deletion. This fits the use of a Trusted Platform Module in this thesis.

2.4.2 Cryptographic Deletion

The area of cryptographic deletion is not an actively researched topic at the time of writing this thesis. I can only assume that this is coincidence and that the topic is not widely known. For this reason, the following discussions on related work contain almost the entire published academic literature on the topic. Other topics with similar scope and impact, e.g. oblivious RAM (see [Gol87]), see regular contributions at a wide range of venues. For these reasons, I believe that the topic of cryptographic deletion has a lot of potential for future research. I discuss some of this potential in the summary of this thesis.

In the following, I only consider publications that explicitly employ cryptography to delete data. Many publications on cryptography mention that deletion of a key leads to deletion of the encrypted data. All the approaches I cover have in common that they employ a specialized key management mechanism like the Key-Cascade presented in this thesis.

The earliest work that I found in my investigation is a 2005 technical report by R. Perlman of Sun Microsystems, who is well known for her development of the spanning tree protocol [Per05b]. Later in that year, Perlman published her findings at IEEE as well [Per05a]. In these publications, Perlman mentions previous work by a discontinued company called "Disappering Inc.". I could no longer find any information about this company or their product, but the technical report details the functionality of said product: It was a time-based secure deletion mechanism for distributed message exchange. Users of the system could exchange messages that had an expiration date, after which they could no longer be decrypted. This was achieved by a central server that acted as a key store. Users would request an encryption key for sending a message to a specific user from this key store and specify the desired expiration date. The recipient of that encrypted message would then request a decryption key from the key store. After keys have expired,

they can no longer be retrieved from the key store.

This approach has obvious downsides, which may be the reason behind the product's discontinuation: The central key store must store a large amount of keys, which makes securely deleting these keys difficult. The keys transmitted between users and key store are not secured against interception. Finally, nothing prevents the recipient of the message from simply keeping the key as well as the message. The expiration date is ineffective in this case, since it is intended to protect the message if its recipient gets compromised in the future.

In her technical report, Perlman presents improvements on Disappering's system. But without fundamentally changing the design, only the first of the above shortcomings could be addressed: the storage requirement on the key store. Perlman proposes two alternative key management approaches to improve on the "store n keys for n messages" approach by Disappearing. Firstly, she presents a method that only requires a single key per expiration date. Using asymmetric cryptography and key wrapping, she derives n message keys for n messages with the same expiration date from a single "date key". Since the message keys can be derived from this date key, they need not be stored themselves.

Secondly, she proposes a further improved method that only requires storing a single key in total. This is achieved through hash chains and time windows. Assuming a time window of one minute, the method works as follows: at $t = 0$, an initial key "k" is generated. A user requests a key for a message that should expire at $t = 30$. Perlman's "Ephemerizer" now computes the 30th hash of k: $h_{30}(k)$. In other words, 30 iterations of computing the hash of the hash of k are performed. This resulting $h_{30}(k)$ is then used as the encryption key for that message. Once the time progresses to $t = 1$, the Ephemerizer calculates the first hash of k: $h_1(k)$ and stores the result in k's place; i.e., k is replaced by its hash each minute.

Until $t = 30$, it is still possible to retrieve the hash for the message from before: at $t = 1$, the 29th hash has to be computed and so on. At $t = 29$, the hash of the current value of k is the key for the message. At $t = 30$, the current value of k is the key for the message. But at $t = 31$, it is no

longer possible to compute the value of the message key. Assuming the hash function can not be reversed, which, however, is one of the main design goals of cryptographic hash functions.

Perlman's improvements describe a clever time-based cryptographic deletion approach. Continuous hashing and replacing is the basis for some of the later time-based approaches as well. Note that it is still necessary that the Ephemerizer has a storage location with secure deletion for the key hash. Otherwise, an adversary could recover the deleted old values.

In 2009, Geambasu et al. propose a novel approach to time-based cryptographic deletion: harnessing the forgetfulness of distributed hash tables (DHT) in order to "securely" delete keys [GKLL09]. Their "Vanish" system focuses on automatic, time-based deletion of encryption keys. Without the possibility for user influence, deleted encryption keys disappear from the system after a certain period. The time-based secure deletion of Vanish is achieved by making use of the forgetfulness of DHT networks. Users can delete key/value pairs from the distributed hash table, but at this point the data is not considered to be securely deleted. In Vanish, secure deletion is achieved passively. Over time, nodes leave and join the network and balance the hash table distribution. Data that was deleted by the users, will not be replicated to new nodes. Vanish considers data to be securely deleted, when none of the nodes in the current DHT network has a copy of that data.

Vanish describes two components: a user who encrypts data using a random encryption key and a large DHT that stores the keys. The user keeps no copy of the encryption keys they used. This use of DHT makes Vanish capable of handling very large amounts of encryption keys, depending on the number of nodes in the network. However, a strong adversary can infiltrate large parts of the publicly distributed hash table and gain control over the keys. Such an attack has been proven to be successful against the TOR network [Din14].

Operating the DHT on trusted, on-premise nodes prevents such an attack, but limits the size of the network. This raises the additional issue that only few nodes hold the keys. Each of those nodes would require a secure storage location for the keys. Otherwise, the forgetfulness is not given in practice.

However, in such a scenario, the forgetfulness of DHT is no longer a necessary property. Vanish is useful when the DHT is running on a large number of diverse internet nodes. Due to the reliance on DHT forgetfulness, Vanish does not support selective deletion of objects at the users' discretion.

In 2010, Melvin et al. of Zytec Communication Corp. published a white paper describing a time-based cryptographic deletion approach [Mel10]. It provides a granularity on the file level and is designed for local disk encryption. Their "Forgetful Disk Drive" concept uses a single, current key for encrypting all newly written files. A new current key is generated at set time intervals, while a key management system retains a queue of keys of a certain length. This approach imposes a time-to-live on all the data written on the device and guarantees that old data, which hasn't been re-written, would eventually become unrecoverable. This is achieved through the queue of constant length and continuous insertion of new current keys. It is not possible to selectively delete data on the device prior to their expiration date. Additionally, there is no provision for compressing the queue like Perlman's hash chains. The approach is only feasible with either a short time-to-live but small windows, or with a long time-to-live but large windows. The reason is that a long time-to-live with small windows leads to a very large queue size. However, they propose to use a Trusted Platform Module as a secure storage location for the key queue.

In 2012, Tang et al. present FADE, a cloud storage proxy system that provides cryptographic access control and assured deletion [TLLP12]. With Perlman as co-author, their work is based on the earlier Ephemerizer system from 2005. Similar to the SDOS approach presented in this thesis, they design and implement their system based on an existing cloud storage API. However, they do not implement an API proxy so that the clients must interact separately with their key manager component and the cloud storage system.

The clients generate random data keys to encrypt the actual data and delegate the encryption of these data keys to the key managers. The clients only keep an encrypted version of the data keys and an access key, which they use to authenticate against the key managers. The key managers store

the control keys which they use to encrypt the clients' data keys. The trusted key managers can now delete the control keys upon request or depending on policies, in order to securely delete the data stored by the clients. Similar to the Key-Cascade presented in this thesis, they use a hierarchy of wrapped encryption keys. However, they do not use re-keying and therefore have to provide secure deletion for each individual control key stored on the key managers. This is contrary to the Key-Cascade, where secure deletion must only be provided for a single key. With this, FADE lacks a mechanism for securely managing large amounts of keys in the key managers.

However, FADE presents a novel combination of cryptography-based access control and secure deletion. Cryptography-based access control means that access permissions on data are not enforced by program logic (as is usual), but by cryptographic key relations. With this method, access control is inherently enforced and can not be circumvented, unless the adversary has the correct key.

The most recent work on cryptographic deletion was published between 2013 and 2014 by a group at ETH Zürich. Reardon et al. describe an approach similar to the Key-Cascade with their B-tree-based cryptographic deletion method [RRBC13; RBC13; RBC14]. They use a dynamic tree to handle large amounts of keys and also require a separate, external data structure for referencing keys to objects. They also use a single root key and the paths in the tree are chains of encrypted keys, as is the case in the Key-Cascade. In order to provide cryptographic deletion of data through secure deletion of the root key, they propose a method similar to the re-keying operation on the Key-Cascade. Their method is called "shadow graph mutation". They determine a new graph in which only the non-deleted leaves are reachable and then introduce new intermediary nodes under a new root key. With this method, they securely replace the root key and retain access to only the desired leaves.

However, they apparently do not take into account the effects of rebalancing the tree. Their construction includes rebalancing the B-tree during insertion. Rebalancing changes the position of the node contents in the tree; these nodes are encrypted with keys stored in the parent nodes. In order

for secure deletion to be necessary, we have to assume that old data can be reconstructed. This includes the state of the B-tree prior to rebalancing. When a shadow graph mutation is performed on a previously rebalanced tree, these old nodes are not taken into account. There is a possibility that reconstructed old nodes contain a path, decryptable with the current root key, to a presumably deleted object. In other words, the secure deletion can be ineffective after rebalancing. This can be avoided if a root key change and appropriate shadow graph mutations are performed after rebalancing.

To summarize, only few but very different approaches have been investigated so far. Time-based cryptographic deletion is a popular approach. The reason seems to be that more optimized key management solutions are possible when the order of the deletions is known. This can be seen in Perlman's perpetual hashing method. She manages to completely eliminate the need to store any keys other than a master key. All selective cryptographic deletion approaches still need to store a large volume of keys. The most recent development in this area is the use of a hierarchy of wrapped keys. This approach is found in the work by Reardon et al. and the Key-Cascade. This hierarchy of wrapped keys brings two desirable benefits: i) It transfers the secure delete property of a single key to a large volume of data. Only this single key must be kept secure. ii) It allows selective deletion and continued use of the system. The single key can be replaced and a new key is only valid to decrypt select paths.

Common to all related work discussed here is that they do not focus on usability aspects and possible integration into existing applications. The most advanced solutions in that regard are FADE and the prototype from Reardon's group: SoK. However, both do not offer a standard client interface. This means that existing applications must be adapted to use those solutions. They also only briefly discuss practical solutions for managing the master key. Some mention Trusted Platform Modules as a key store, but none have a working implementation like SDOS.

2.4.3 Encryption in Database Systems

Encrypted database systems share some design and application aspects with encrypted cloud storage systems: In both cases, the data management system should have some access to the content of the encrypted data. In database systems, this is because query processing ideally should happen where the data resides. However, this creates a conflict with encryption that hides the contents. In cloud storage systems, the situation is similar. Some operations are ideally performed on the cloud, where the encrypted data resides.

For example, when the object names in an encrypted cloud object store are accessible to the provider, they can offer easier access to specific objects. If this is not the case, then each user must perform some operations to obfuscate and de-obfuscate the object names. Generally, cloud providers are interested in offering rich services to their customers. Such services, like advanced querying, are only possible if some access to encrypted data is possible. Therefore, it is useful to investigate how encrypted database systems address this issue.

In the following, I distinguish two different ways of realizing encrypted database systems: (i) Using specialized cryptography that allows operations on encrypted data. (ii) Allowing the database to decrypt some of the data for processing.

Cryptographic algorithms that enable operations on the encrypted data are called homomorphic encryption. They are actively researched today and far from being universally applicable. Fully homomorphic encryption describes such an algorithm that enables arbitrary computations on the encrypted data. The possibility of such algorithms has been suspected soon after advanced encryption algorithms were developed in the 1970s. It was however first proven to be possible in 2009 by Gentry et al. [Gen09]. Many improvements on his concept have been proposed since, but they are all highly inefficient. A recent advancement by Brakerski et al., for example, still requires seconds to minutes for simple operations [BV14].

Fully homomorphic encryption today works by leveraging a small number of operations in order to emulate arbitrary computation. In Gentry's original

construction, the underlying partial homomorphic encryption allows addition and multiplication on encrypted data. This leads to very computationally expensive execution.

For these reasons, fully homomorphic encryption is not used in production database systems today. However, the potential of homomorphic encryption is widely recognized and applications are conceivable in almost every area of distributed and cloud computing. Cryptographic deletion in the context of homomorphic encryption is an interesting research problem for this reason. Any system that uses data encryption can provide cryptographic deletion, if an appropriate key management mechanism is used. This is true for homomorphic encryption as well. Databases that are encrypted in this way, could integrate cryptographic deletion into the regular delete operations. This would add a potentially desirable feature to such databases.

Partial homomorphic encryption directly supports certain operations and is used in production database systems today [FG07]. These approaches bear another issue: Each encryption algorithm supports only one or few operations. This means that multiple copies of the data must be stored, each encrypted differently for the different types of required operations.

The most notable implementation of an encrypted database system that uses partial homomorphic encryption, is CryptDB [PRZB11]. This and similar approaches work as follows: An unmodified database system is used, but clients encrypt and modify their queries. They leverage a set of encryption algorithms to support the basic operators required for SQL queries, like comparison, ordering, or addition. Insert queries are duplicated by the client, so that multiple copies of tables exist, each encrypted to support a different kind of operator. For example, deterministic encryption always produces the same output for the same input, if the same key is used. This effect is best demonstrated by encrypting bitmap image data, as shown in Figure 2.6. It also highlights an issue with these approaches: deterministic, order preserving, or similar encryption algorithms always leak some information about the encrypted data.

Different approaches to implementing such a system exist. In CryptDB, an API proxy for MySQL was implemented so that unmodified clients and MySQL

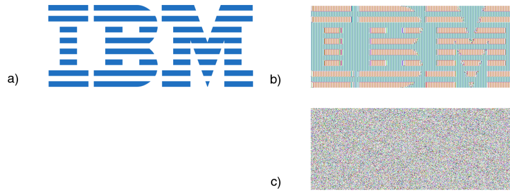

Figure 2.6: Deterministic (b) compared to randomized (c) encryption.

servers may be used, similar to the Swift API proxy approach presented in this thesis.

Since these database systems must store differently encrypted copies of all tables, they impose a high storage overhead. However, this overhead may be tolerable and the performance of these systems is acceptable. At least for simple read queries, the query processing in the database is not different from an unencrypted table. Only insertions are expensive and the overhead from the API proxy must be taken into account. Ada Popa et al. say CryptDB "...explores an intermediate design point to provide confidentiality ..." until fully homomorphic systems become feasible.

The following approaches comprise the second type of encrypted database systems (ii) introduced above; systems where the data is encrypted at rest, but the execution engine has access to keys. This is the approach taken by commercial database vendors today.

With Microsoft Cipherbase, Arasu et al. describe a relational SQL database system that uses encryption to provide confidentiality of stored data [ABE+12; ABE+13; AEJ+15]. They do not employ homomorphic encryption for the reasons stated above, but use regular cryptographic algorithms. The main idea behind Cipherbase is to allow the cloud to decrypt the data for processing, but do so in a secure manner. This is achieved through hardware/software co-design with an HSM to create a small trusted enclave inside the untrusted cloud server, in which data decryption and processing of sensitive content happens. This trusted enclave is inaccessible to the cloud provider and administrators and will only process legitimate queries by an authenticated client. The results are encrypted again inside the trusted enclave and then

transmitted back to the client, through the untrusted environment.

With this approach, it is possible to provide full server-side SQL query processing on an encrypted database. The downsides are that special hardware is required on the cloud server side and that this hardware must be trusted. It is also difficult to achieve high performance: Without optimization, all decryption/encrypting and query processing would be done inside the trusted enclave which has limited processing power. The actual cloud server would do very little work. So for every database operator it must be investigated if part of the processing can be done on the cloud server on encrypted data. The mode of encryption must be adjusted to fit these processing requirements, while still maintaining the confidentiality of the data.

Bajaj et al. also use the "trusted enclave" approach in TrustedDB [BS14]. They offer full SQL capabilities on an encrypted database by using a trusted coprocessor inside the cloud server. Instead of using custom hardware for the trusted enclave like Cipherbase, they use a generic Hardware Security Module from IBM. This "CryptoCard" has authority over the encryption keys used for the database and only responds to legitimate requests from authenticated clients. An advantage of using a generic cryptographic coprocessor over custom hardware is their certified security. Their disadvantage is the lower performance compared to custom hardware.

Janusz et al. descibe an approach for executing range queries on encrypted databases [JT17]. In their ZeroDB system, they implement an approach similar to Cipherbase, except that decryption and processing happens on the client side and not inside the cloud. Part of the query processing is delegated to the client. The database system on the cloud only provides the logical storage layer.

Chapter 3

Key-Cascade Method

In the following, I present the main contribution of this thesis: The Key-Cascade method.

The Key-Cascade method describes a data structure and operations for managing the encryption keys used in cryptographic deletion.

The general principle of cryptographic deletion is detailed in Section 1.3: Data are encrypted and stored separately from their encryption key. A trusted storage location is then used for the encryption key, so that no deleted key can be recovered later. This *secure delete* property of the trusted storage is then transferred to the stored data by means of encryption: If the key can not be recovered, neither can the encrypted data.

As I have shown with the key management problem in Section 1.5, it is non-trivial to manage the necessary encryption keys. The reason is that only a small number of keys can be stored and managed securely, but a large number of keys is actually needed in order to encrypt all necessary objects.

The Key-Cascade solves this key management problem by creating a hierarchy of wrapped encryption keys (see Section 2.2) in a tree structure. With this method, only a single root key must be stored securely, but a large number of leaf keys is available for encrypting the objects. Furthermore, the

Key-Cascade achieves this without violating the storage, performance, and granularity requirements stated in Section 1.6.

The Key-Cascade method is independent of a specific storage system or application topology. For consistency with the rest of the thesis however, I describe the concepts in this chapter in the context of an untrusted cloud storage service that is accessed by clients through a trusted gateway as shown in Figure 3.1. In this context, the Key-Cascade resides on the trusted gateway. All the operations discussed in this chapter are executed on this gateway.

I design and implement the contributions in this thesis around the data models and interfaces of an object store. For this reason, I also use the term *object* as the unit of data that gets stored, retrieved, and deleted in the following examples. Objects are comparable to files in a file system, they are addressable by an object identifier. The actual data model of the object stored is discussed in detail in Chapter 5. For the following conceptual discussion, it is enough to assume a single, flat hierarchy of objects with consecutive numbers as identifiers.

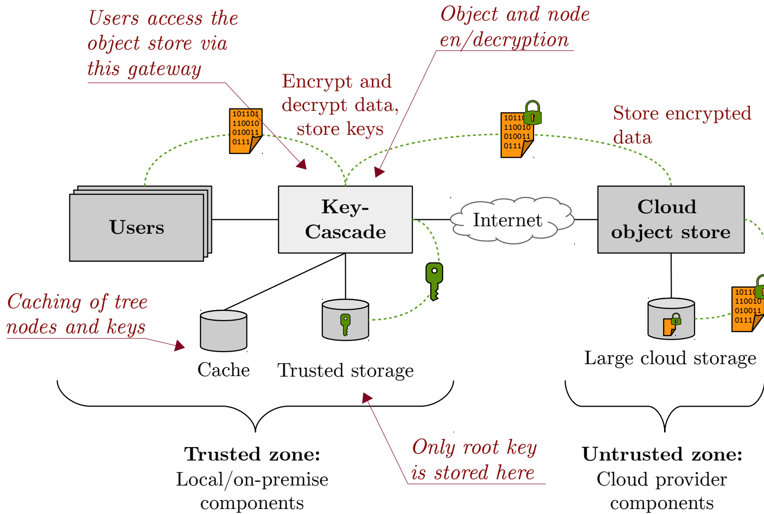

Figure 3.1: Overview of the reference application.

3.1 Key-Cascade: Abstract Idea

The abstract idea behind the Key-Cascade is to use a tree structure in order to establish relations between a single root key and a large number of keys in the leaves. This makes use of key-wrapping (see Section 2.2), so that parent nodes in the tree contain encryption keys for their children. Each path in this tree is a chain of wrapped keys.

The Key-Cascade manages encryption keys for objects in the leaves of its tree. The two parameters *tree height* and *node size* determine the geometry of the resulting tree. In the following example, I explain how such a Key-Cascade with tree height 2 and node size 4 is constructed. Figure 3.2 illustrates the following five steps:

1. Four objects (O_{oid}) should be stored encrypted. First, four object keys are randomly generated and used to encrypt each object with an individual key: ok_{oid}. These encrypted objects can now be stored on the cloud.

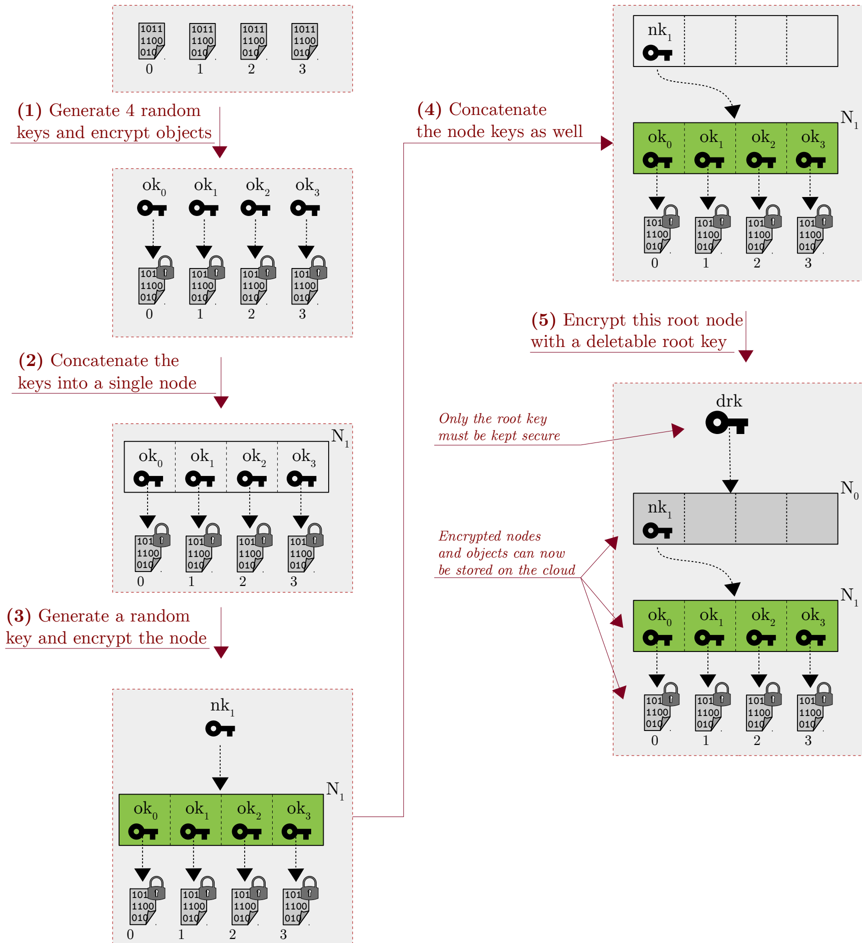

Figure 3.2: Construction of a new Key-Cascade with tree height 2 and node size 4.

2. The four object keys are now concatenated together to form an **object-key node** N_{nid}.
3. A new random node key (nk_{nid}) is now generated and used to encrypt the previously created node. This encrypted node can now be stored on the cloud.
4. Only one node key exists in this example, but up to four node keys are brought into the same node format as the object keys before. These inner nodes, containing node keys for object-key nodes, are called **node-key nodes**.
5. Because the tree has height 2, this node-key node is the root. The root is finally encrypted with a deletable root key: drk. This encrypted root node can now be stored on the cloud and the deletable root key on the trusted storage location.

The goal of this approach is that secure deletion of the root key leads to secure deletion of select object (leaf) keys. In other words, when the root key is replaced with a new root key, this new root key is no longer valid to decrypt these select object keys. I achieve this with the cascaded re-keying method presented in Section 3.7

3.2 Data Structure and Key Organization

The structure of a Key-Cascade is given in Figure 3.3[1].

The bottom row represents the actual data objects stored on the object store. Each object is identified by its **object identifier or object ID:** oid. These oids are sequential numbers, starting at 0 for the first object. The objects are encrypted with individual **object keys:** ok_{oid}, which are randomly generated. The object keys are identified by the same numbers used as object identifiers. In other words, object key ok_i is the key for the object with the ID i.

[1] The illustrations in this Chapter use a Key-Cascade with height 2 and a node size (degree) of 4, but these two parameters can take arbitrary values.

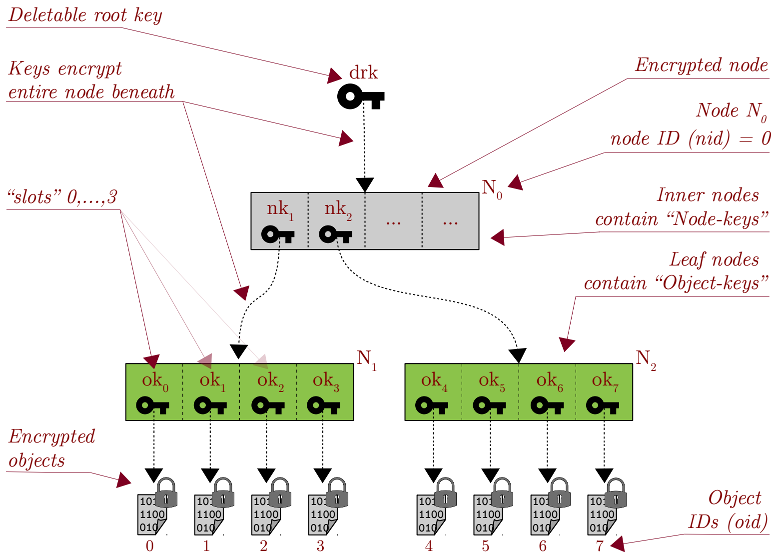

Figure 3.3: Structure of a single Key-Cascade with height 2 and a node size of 4. Nodes 3 and 4 are not shown.

The leaf level of the Key-Cascade groups these object keys into nodes that contain a fixed number of slots. These object-key nodes are formed by ordering the object keys in ascending order of the object ID and then grouping them by the node size: In the example, the first object-key node contains object keys $0, 1, 2, 3$.

These initial nodes, containing the object keys, are further on encrypted themselves, and their node keys grouped into further nodes. This process is repeated for as many levels as the Key-Cascade has.

The Key-Cascade's leaves are object-key nodes and the inner nodes are node-key nodes. Each of these nodes is identified by a **node identifier or node ID:** *nid*. These *nids* are chosen by numbering all nodes breadth-first, starting with 0 at the root. The breadth-first traversal is done based on a perfect k-ary tree[1], where *k* is the node size. This means that *nids*

[1]As described in [CSRL01].

are assigned based on the position of the node in the tree, rather than by enumerating the nodes actually used. In other words, if a node is missing because it was not used yet, it is still counted when assigning the *nids*.

3.2.1 Static Tree Benefits

An important design property of the Key-Cascade is that it forms a *perfect k-ary tree*. This property comes from using a fixed size for the nodes, as well as a fixed height of the tree. This yields the following two benefits:

Static tree benefit 1: Re-balancing the tree is avoided. Since the nodes of the tree are encrypted lists, re-balancing the tree involves re-encrypting of the node contents, which I want to avoid. The Key-Cascade would require a *cascaded re-keying operation* (see Section 3.7) for balancing, which eventually requires replacing the root key.

Re-balancing the tree without cascaded re-keying and replacement of the root key causes an issue: It leaves secondary access paths to leaf nodes behind. This renders future secure deletion through cascaded re-keying ineffective.

Static tree benefit 2: The second benefit of the perfect k-ary tree lies in locating the nodes. Since the tree is assumed to be completely filled, I can sequentially number all nodes in a breadth-first order and exploit the fact that each node has a fixed ID according to its position in the tree. This means that every object key and node key has a known location inside the tree and this location can be calculated from the tree geometry and the object ID *oid* alone. This property is especially useful in the case of the Key-Cascade, since the nodes are encrypted and stored externally. With only the knowledge of an object ID, I can locate and retrieve all the nodes along its access path at once and then process them locally. In a dynamic tree, I would first have to retrieve and decrypt the root node in order to determine the location of the next node and so on. This structure allows decoupling the retrieval and the processing of the encrypted nodes.

The downsides to this structure are the possibility of poorly utilized sparse trees and the fixed number of maximum leaf nodes. Sparse trees can be

avoided with a depth-first allocation algorithm, that is explained in Section 5.2. The fixed number of leaf nodes, and therefore object keys, is handled by choosing a sufficiently large tree. Only the nodes that are actually used need to be materialized and stored. This means that the physical size of the tree grows with the amount of used objects keys. Section 3.4 gives the formulas for calculating the tree size and its capacity. However, growing the Key-Cascade's tree to create room for additional object keys is possible as well. A discussion on resizing the tree is found in Section 4.1.

3.3 Data Structures and Storage Locations

The central data structure is the Key-Cascade that consists of the individual nodes. How this and other data structures are stored as objects on the object store, is illustrated in Figure 3.4. Each node gets encrypted on the gateway and stored on the cloud as an individual object identified by its *nid*. In their decrypted form, a node is a list with a fixed number of slots that contain only the encryption keys. This structure allows for a minimal size of the nodes, since they only store the encryption keys and no additional data. This means that the edges between the nodes, as indicated in the illustrations, are not stored as part of the data structure. They are given implicitly by the identifiers of the nodes and objects as described in Section 3.6.

In order to gain access to the contents of the Key-Cascade, the gateway stores a *root key* that encrypts the root node. This is the only data that is permanently stored on the gateway and also the only data for which secure deletion (e.g. by a Hardware Security Module) must be guaranteed, since the Key-Cascade transfers this secure-deletion property from this root key to all managed data. For this reason, I call this key the *deletable root key* in the further discussions.

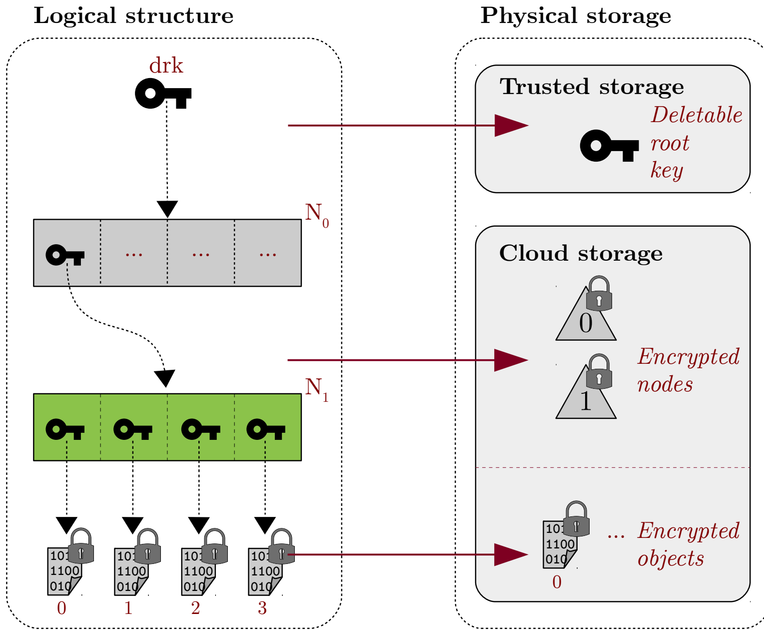

Figure 3.4: Key-Cascade objects and their storage location.

3.4 Complexity and Efficiency Consideration

There are two parameters that determine the geometry of the Key-Cascade: the node size S_n and the tree height h. When these parameters are set, they determine the further properties of the Key-Cascade. In the following, I give an explanation of these properties and their respective formula. The formulas for the number of nodes and object keys are derived from the formulas found in literature for calculating properties of a perfect k-ary tree [CSRL01].

Node size parameter: S_n

It determines the number of slots (keys) in each node and, therefore, also the number of child nodes (i.e., degree / branching factor).

Tree height parameter: h

The tree height determines the number of levels in the Key-Cascade.

The tree in Figure 3.5 for example has a height of two.

Number of object keys: $R = S_n{}^h$

The number of object keys we can store in a cascade is given by the number of object-key nodes $S_n{}^{h-1}$ (i.e., leaves in a perfect S_n-ary tree) multiplied by the node size: $S_n{}^{h-1} \cdot S_n = S_n{}^h$.

Number of nodes: $L = \sum_{i=0}^{h-1} S_n{}^i = \frac{S_n{}^h - 1}{S_n - 1}$

Represents the maximum number of nodes I need to store (in a fully utilized Key-Cascade) and allows calculating the required storage capacity. It is given as the closed-form expression of the sum of nodes on all levels of the tree.

First object-key node ID: $nid_{fon} = \frac{S_n{}^{h-1} - 1}{S_n - 1}$

This is the nid of the first object-key node (“1” in Figure 3.5). All further object-key nodes have consecutive following $nids$ until $L - 1$ (because the enumeration starts at the root with 0).

Cascade storage: $K = L \cdot S_n \cdot key\ length$

Gives the number of bytes required to store the Key-Cascade (all nodes). The *key length* represents the size of the keys stored inside the nodes. In SDOS, this is always 32 bytes due to the AES256 encryption algorithm.

Re-key complexity, worst case: $C_{rk} = h \cdot (S_n + 2)$

Re-keying involves a copy operation for each key in each node and two additional operations for encrypting/decrypting each node. Note that one object key doesn’t need to be copied (the deleted one), while one new deletable root key must be generated. These two operations offset each other so that both can be omitted. This worst case is given when all involved nodes are fully utilized so that every key slot contains a key.

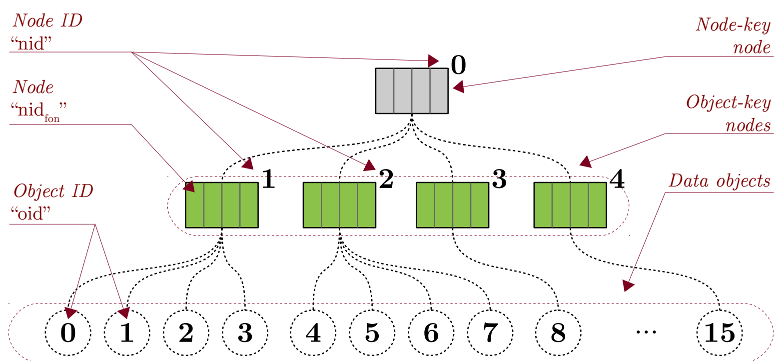

Figure 3.5: Representation of a Key-Cascade's tree structure showing only the nodes and IDs. Height $h = 2$, node size $S_n = 4$.

3.5 Key-Cascade Geometry Examples

In the following two examples, I show the properties of two differently parametrized Key-Cascades. Further, much larger geometries are presented and evaluated in the performance evaluation in Section 5.6.

Example 1:

Tree height $h = 2$, node size $S_n = 2^2 = 4$.

This results in a cascade consisting of 5 nodes with space for 16 object keys. When fully utilized, the Key-Cascade needs 640 bytes to store the nodes. Re-keying requires up to 12 operations. With objects of 100 kilobytes average size, this cascade can store keys for 1.6 megabytes of data. The cascade therefore imposes a storage overhead of 0.04%.

This Key-Cascade geometry is used in the illustrations and calculations of this thesis because of the manageable size and number of nodes.

Example 2:

Tree height $h = 3$, node size $S_n = 2^8 = 256$.

This more realistic parameter setting results in a cascade consisting of 65,793 nodes with space for 16,777,216 object keys. When fully

	Individual key	Master key	Key-Cascade
Volume of trusted local key storage	**512 MB**	64 bytes	64 bytes
Volume of keys stored on cloud	0	0	514 MB
Add. operations for deleting	1	**16,777,215**	774

Table 3.1: Storage and computational overhead comparison for preliminary approaches from Section 1.5 and Key-Cascade. Using an object store with 16,777,216 objects and the Key-Cascade from Example 2.

utilized, the Key-Cascade needs 514 megabytes to store the nodes. Re-keying requires up to 774 operations (note that these are small in-memory operations). With objects of 100 kilobytes average size, this cascade can store keys for 1.6 terabytes of data. The cascade therefore imposes a storage overhead of 0.03%.

In Table 3.1, I give a comparison of the storage and computational overhead between the two preliminary approaches (see Section 1.5) and the Key-Cascade. This is an extension of Table 1.1 from Section 1.6 at the end of the Introduction, where Requirements 1.1 and 1.2 are introduced. The two highlighted fields show again where the preliminary approaches violate the requirements, while the Key-Cascade satisfies them. Note that the $16,777,215$ additional operations, listed for the master key approach, refer to re-keying operations on stored objects. These are expensive I/O-heavy operations. The 774 additional operations for the Key-Cascade method, however, mostly consist of in-memory copy operations. A detailed break down of the operations, necessary for secure deletion with the Key-Cascade method, is given in Section 3.7.

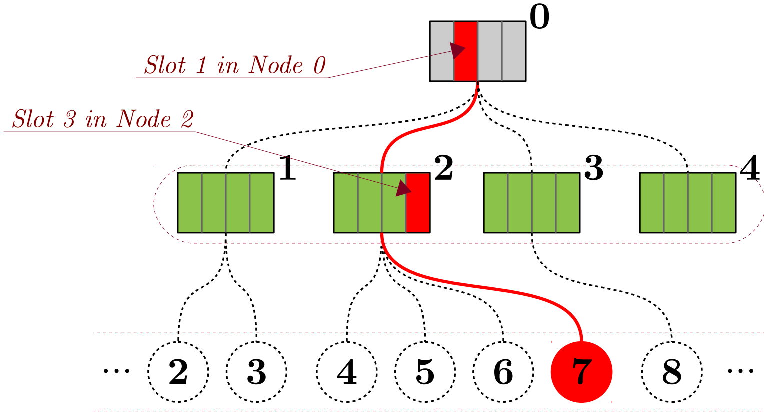

Figure 3.6: The path to the object key for Object 7, found through ID calculations.

3.6 ID Calculations

ID calculations are used during operations on the Key-Cascade in order to determine paths from the root to desired object keys. They are enabled by the use of a perfect k-ary tree and allow determining paths without inspecting node contents.

Each object-key references and encrypts exactly one object. The objects and their object keys are identified by their **object ID** *oid*[1]. They are numbered sequentially starting from 0 and in this example (see Figure 3.5) ending at 15. A request for reading, writing, or deleting an object starts with the object's ID. So I need to determine the **node ID** *nid* of each node along the path from the root to the node containing the object key for the object to be accessed. I also need to determine the **slots inside the nodes** that hold the keys for their children along this path. For this, I have to carry out two types of calculations: Firstly, to calculate the ID/slot for the object-key node once. Secondly, to calculate the ID/slot for the path of node-key nodes

[1]oids are only used internally. Section 4.4 introduces their mapping to object names as used in SDOS/Swift.

iteratively ($h-1$ times) in order to traverse the whole path. This is done only by ID calculations that don't require additional data structures.

The following formulas are derived from the formulas found in literature for calculating properties of a perfect k-ary tree [CSRL01].

Example: Decrypting Object $oid = 7$ (Figure 3.6 highlights the resulting path). I know that the tree has a height of $h = 2$ and the nodes have a size of $S_n = 4$. I start by calculating ID/slot for the object-key node and then continue with the node-key nodes, until reaching the root.

Object-key node ID: $nid = i + nid_{fon}$
where $i = oid \text{ div } S_n$.
In this example: $nid = 7 \text{ div } 4 + 1 = 2$. I first calculate i, the i-th object-key node, from the *oid*. Since the first node containing object keys is nid_{fon}, I have to add that to i.

Object-key node slot: $slot = oid \bmod S_n$
$slot = 7 \bmod 4 = 3$. Here I calculate the slot inside the object-key node that holds the key for the object.

Node-key node ID: $nid_{parent} = (nid - 1) \text{ div } S_n$
$nid_{parent} = (2-1) \text{ div } 4 = 0$. Here I calculate the parent ID for a node.

Node-key node slot: $slot = (nid - 1) \bmod S_n$
$slot = (2-1) \bmod 4 = 1$. I calculate the slot inside the parent node-key node.

This yields Slot 3 in Node 2 as the location of the object key for Object 7; i.e., the encryption key used for Object 7. The node key for that node lies in Slot 1 of Node 0. Since I know that the height of the tree is 2, I already know that in total only two iterations of calculating node IDs and slots are required. Termination is also indicated by reaching Node 0 (the root).

Note that these formulas apply in general and that they can be simplified for particular Key-Cascades of a known geometry. In this example, the node ID of the object-key node (first step in the above example) can be expressed [1] as $\lfloor \frac{oid+4}{4} \rfloor = \lfloor \frac{7+4}{4} \rfloor = \lfloor \frac{11}{4} \rfloor = \lfloor 2.75 \rfloor = 2$.

[1] Where $\lfloor x \rfloor = \max\{m \in \mathbb{Z} \mid m \leq x\}$, or the "floor" of x, or x rounded down

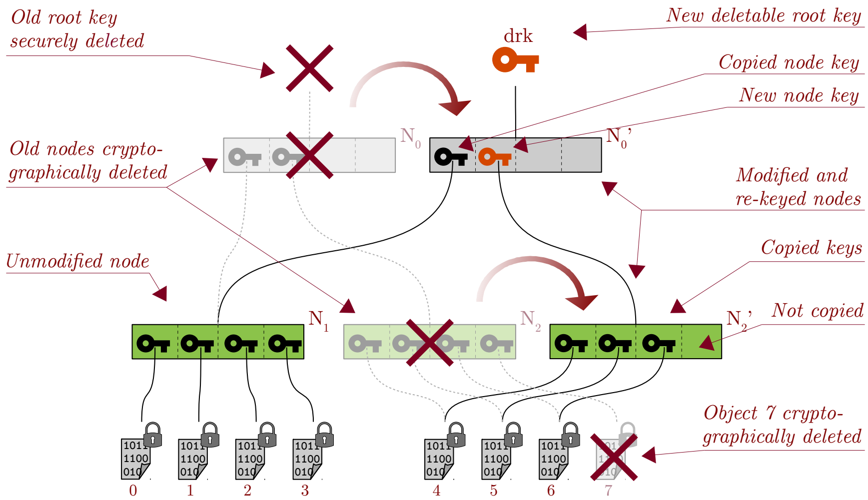

Figure 3.7: Cascaded re-keying results in cryptographic deletion of Object 7.

3.7 Secure Deletion Through Cascaded Re-Keying

Cascaded re-keying is the operation by which cryptographic deletion is achieved.

The cascaded re-keying operation is an extension of the re-keying operation introduced in Section 1.5. This extension is necessary in order to apply re-keying to the chain of nodes inside the Key-Cascade, rather than an individual key. This operation securely deletes an object key from one of the object-key nodes by successively re-keying the nodes along the path, until reaching a new deletable root key for the whole cascade. Cascaded re-keying is done sequentially instead of concurrently, because parent nodes contain the keys for their children. The decryption, copying, and encryption operations are performed in main memory on the trusted gateway.

Figure 3.7 shows the Key-Cascade after the deletion of Object 7. First, determine the path of the involved nodes by using ID calculations. Then, start processing the nodes, starting from the top.

Definition 3.1 (Cascaded Re-Keying)

1. *Load the current deletable root key,* ***drk*** *(only if this is Node 0)*
2. *Generate a new deletable root key,* ***new drk*** *(only if this is Node 0)*
3. *Retrieve the current node (first iteration is Node 0)*
4. *Decrypt the node with the previously extracted key (or drk if this is Node 0)*
5. *If this is an inner node (node-key node):*
   a) *Extract the current key for the next node on the path (here the key for Node 2 from Slot 1 in Node 0)*
   b) *Generate a new node key for this next node*
   c) *Copy the node and replace the key for the next node with the newly generated one*
6. *If this is an object-key node:*
   a) *Copy the node* ***without*** *the key for the deleted object*
7. *Encrypt the node with the key generated in the previous iteration (or with the new drk)*
8. *Write the modified encrypted node to the object store*

Then the next iteration starts with the (next) child node at Step 3. The trusted gateway always holds three keys temporarily between iterations: first the new deletable root key, second the current key for the next node on the path, and third the new key for the next node. Once all modified nodes are written to the object store, the old deletable root key can be securely deleted and replaced with the newly generated new deletable root key. Without access to the old deletable root key, none of the old nodes and referenced objects, can be decrypted.

The above example explains the operation for deleting a single object key. However, cascaded re-keying can easily be adapted for deleting multiple object keys in a single run. This is achieved by determining all necessary paths first and then by combining the operations that affect the same nodes.

Each necessary node only has to be processed once in this case, therefore only a single replacement of the deletable root key is needed as well.

This is the way in which cascaded re-keying is implemented in SDOS (see Section 4.7).

3.8 Operations: Key Creation, Retrieval, and Deletion

Finally, three operations are offered for using the Key-Cascade as a key-management mechanism: **Create** for generating new object encryption keys, **retrieve** for retrieving the previously generated keys, and **delete** for securely deleting them through cascaded re-keying. These operations presented here constitute the external interface to the Key-Cascade. For this reason, they operate on object encryption keys (which are located in the leaves of the tree). An external interface for directly manipulating nodes or node encryption keys is not present in my concept or implementation, since these data structures are only managed internally by the Key-Cascade itself.

The **create** operation may be used on an empty Key-Cascade because it will inherently create missing nodes and keys. The input for the create operation is an unused object ID *oid*. I then use ID calculations to find the nodes along the path to the object encryption key as well as the slots in those nodes. I follow the example from Section 3.6 and therefore need to retrieve Nodes 0 and 2.

Two situations may occur in this scenario: first, the nodes along the path already exist. In this situation, I retrieve and decrypt these nodes (after loading the root key from the trusted storage location). Then I insert the new randomly generated object encryption key into the object-key node and encrypt it again with the same node encryption key. Now the single, modified node can be written back to the store.

In the second situation, the path does not contain nodes until the end. This situation occurs in either an empty tree, or when no key was previously created that went into the same object-key node. In this situation, I also retrieve all nodes along the path, to get the topmost node encryption key

(root key in empty tree). I then create the missing nodes and sequentially insert the object encryption key and child-node keys into their parent nodes. This new branch of nodes either includes the root node (tree was empty) or is the child of an already present node. If the new branch includes the root node, I only have to write these new nodes to the store. If it has a parent node, then I additionally have to update this parent node in the same way as described in the first situation above. After the Key-Cascade was updated in this way, the newly created object encryption key is returned and can be used.

In the **retrieve** operation, I first use ID calculations to determine the path and slots and retrieve the nodes. I successively decrypt the nodes to reach the object encryption key, which concludes retrieval.

The **delete** operation is provided by cascaded re-keying as described in Section 3.7.

3.9 Related Publications

The contributions in this Chapter are supported by the following of my publications:

My diploma thesis from 2012 discusses architectural patterns and best practices for securing cloud applications, especially through the use of encryption and key management [Wai12]. In this thesis, I present the concept of separating the management of keys and credentials from their use in encryption or authentication. The proposed CKS (Centralized Key Store), a program that stores and manages access to encryption keys and credentials, is a precursor to the trusted storage that stores the deletable root key for the Key-Cascade method.

The initial idea for a tree-based encryption key management was published as a US patent application in 2015. This application was granted US patent 9,298,951 in 2016 [BLM+15] and was awarded "Invention of the Quarter" at IBM software group in the first quarter of 2015.

In 2017, a poster presentation about the Key-Cascade concept and the

prototype of SDOS as well as the larger MCM system was accepted at BTW [Wai17]. In the same year, another poster presentation and demo that showed the integration with Trusted Platform Modules as deletable key source was accepted at EDBT [WWM17].

Chapter 4

Extensions to the Key-Cascade Concept

The previous Chapter introduced the Key-Cascade concepts, data structures and general operations. In this Chapter, I discuss extensions to the core concepts that are necessary for using a Key-Cascade-based encryption key management in a production storage system. This includes master key management, the parallel execution of the operations on the Key-Cascade data structures, static and dynamic resizing of those data structures, the integration of authentication and authorization, as well as additional data structures that improve performance.

The first extension “Resizing the Key-Cascade” is a conceptual extension that is not implemented in the prototype, while the others are realized in SDOS.

4.1 Resizing the Key-Cascade

The Key-Cascade data structure is based on an underlying perfect k-ary tree of which only a subset of nodes are used based on the current utilization. The actual materialized data structures therefore grow and shrink dynamically based on the current demand, but always stay within the bounds of this static tree. The static tree imposes an upper bound for the amount of object keys that can be stored in any given cascade.

This property of the Key-Cascade must be appreciated in any practical implementation by either i) specifying the static tree sufficiently large, ii) making the tree shrink and expand dynamically, or iii) by resizing the tree in a static, offline manner.

i) Specifying Sufficiently Large Trees: The formulas for calculating the capacity are given in Section 3.4. Examples are given in Section 3.5.

According to Swift developers and online discussion[1], no more than 1 to 10 million objects should be stored in any single container. The Key-Cascade from Example 2 already supports over 16 million object keys and performs well according to performance measurements (see Section 5.6). For these reasons, it is possible to configure a sufficiently large Key-Cascade so that resizing is not necessary in practice.

ii) Dynamic Resizing The Key-Cascade's capacity is determined by the node size and the tree height. It can be changed by modifying these two parameters. Dynamic resizing refers to the process of changing these parameters without modifying the existing data structures (i.e., nodes). This allows continuing to use the stored nodes, while new keys are written to new nodes that extend the existing Key-Cascade.

In the following, I show dynamic resizing by increasing the height of the tree that underlies the Key-Cascade[2].

This is achieved by adding a new root node with the old tree as the leftmost child. The old root key then becomes a node key inside the new root node.

[1] `https://ask.openstack.org/en/question/13242/how-many-objects-per-container/` and `https://answers.launchpad.net/swift/+question/181977`

[2] Resizing by changing the node size is also possible with the same approach.

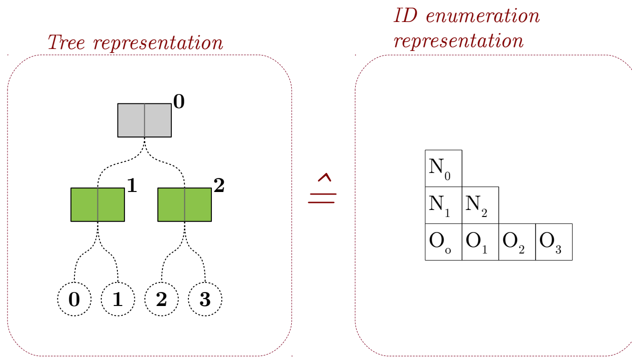

Figure 4.1: Introducing the ID enumeration representation compared to the tree representation. Both figures represent the same Key-Cascade.

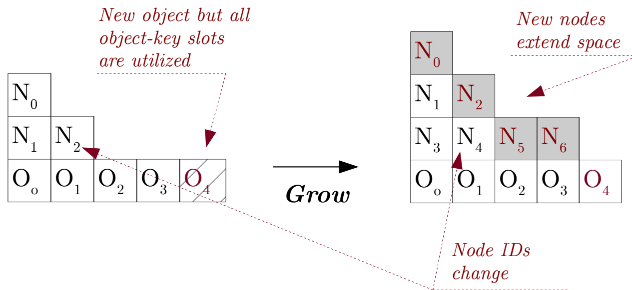

Figure 4.2: Cascade growing with regular ID enumeration.

This effectively adds one new level to the Key-Cascade without the need to re-encrypt any data since all the child-parent relationships stay the same. But on the downside, this operation affects the node IDs. To illustrate the effect, I introduce a new visual representation for the Key-Cascade that focuses on the ID enumeration in Figure 4.1.

Figure 4.2 shows the effect of adding a new root. This Key-Cascade uses the depth-first ID enumeration that is introduced in Chapter 3 and implemented in SDOS. The old Node 2 has become Node 4. In order to use this resizing method, the internal storage layer and ID calculations have to be aware of

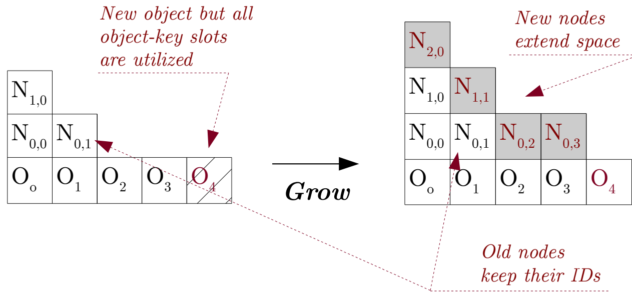

Figure 4.3: Cascade growing with alternative 2D ID enumeration.

the resizing so that the different enumerations can be taken into account. This means that a resize-marker must be stored with the cascade geometry, because a resized Key-Cascade is different from one that was initialized with the same geometry.

The node IDs serve two purposes: i) they identify and locate the nodes when they are stored as objects inside the object store, ii) they enable traversing the Key-Cascade by calculating parent IDs and slots (see Section 3.6). Any enumeration that fulfills these two functions can be used in a Key-Cascade. Therefore, I present another possible enumeration schema that allows transparent resizing. Only the changed tree height must be stored with this enumeration and the internal storage layer and ID calculations need not be aware of the resizing.

Figure 4.3 shows an example before and after resizing. The alternative row/column (2-dimensional) ID enumeration allows the existing nodes to keep their IDs, while the new nodes get new IDs. With this enumeration, a resized Key-Cascade is identical to one that was initialized with this geometry. This alternative schema can be used with a Key-Cascade instead of the regular ID enumeration discussed in this thesis. The only non-trivial change would be the ID calculations (see Section 3.6), which would use different formulas for the row/column IDs.

iii) Static Resizing Static resizing refers to the process of increasing, or decreasing the size of a Key-Cascade by creating a new one and inserting

the object keys from the old cascade into the new structure. Static resizing is always possible without explicit support in the implementation, since it only uses regular read/write operations of the internal API.

This approach of resizing by recreating is often implemented in file systems for block storage devices [RBM13]. Resizing is a rarely used operation in this domain so that speed and cost of the operation are not critical. The major benefit of “resizing by recreating” is that the run-time code needs no special conditions for resized file systems. A resized file system is the same as one that was not resized. This makes the implementation simpler and easier to verify.

4.2 Deletable Key

This extension introduces a wrapped key between the deletable key and the Key-Cascade root node in order to support more flexibility for the deletable key.

The Key-Cascade theory describes a **deletable root key** which must have the secure delete property. This deletable root key encrypts the root node of the Key-Cascade and therefore transfers its secure delete property to all further keys. In the implementation, I replace this deletable root key with a separate root key and a deletable key. The deletable key wraps (encrypts) the root key. Figure 4.4 shows the extended structure.

This extension decouples the deletable key from the root node, which allows more flexibility, because now the deletable key is only used to en/decrypt a single key which is a small object (here 32 Bytes for AES256). Without this decoupling, the deletable key would be used to en/decrypt the entire root node which can be a large object depending on the Key-Cascade geometry (e.g. 8KB for node size 256). This enables the use of cryptographic hardware that has limited storage and processing capabilities (such as the TPM key source introduced below).

Furthermore, this decoupling adds a useful abstraction for implementing the Key-Cascade node operations. With the decoupled deletable key, SDOS

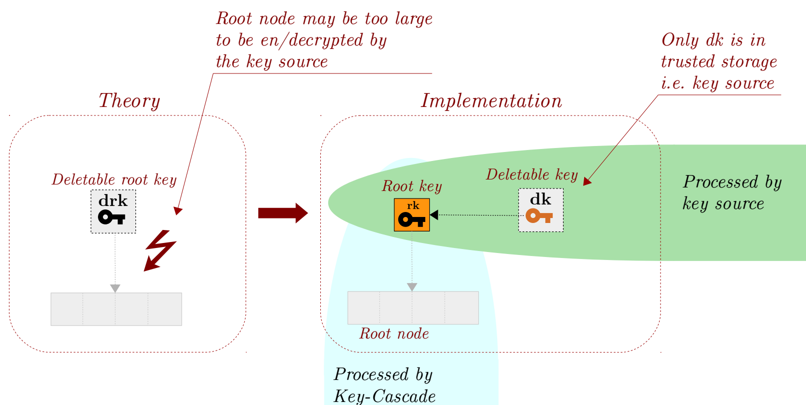

Figure 4.4: In the implementation, a chain of two keys is used as root key for the Key-Cascade. Root key and nodes are handled by the Key-Cascade code, deletable key and en/decryption of the root key are provided by "key sources" (see Section 4.3).

can use any encryption algorithm that the user wants for encrypting the nodes (here AES256), as long as the key source is capable of encrypting a key for this chosen algorithm. SDOS can then treat the root node in the same way as any other node.

The deletable key and root key each have three stages during re-keying and initialization:

- **Current deletable key:** The d.k. currently used to encrypt the current root key.
- **Next deletable key:** The d.k. that will replace the current deletable key. It will be used to encrypt the next root key.
- **Old deletable key:** The d.k. that was replaced by the next d.k. It was securely deleted and is no longer available.
- **Current root key:** The r.k. currently used to encrypt the root node of the Key-Cascade.
- **Next root key:** During re-keying, this key is used to encrypt the re-

keyed root node of the Key-Cascade.

- **Old root key:** After re-keying, this key is no longer used. It decrypts the old root node and therefore the old Key-Cascade, but is no longer accessible since the old deletable key was securely deleted.

4.3 Deletable Key Sources

With the decoupled deletable key, it is now possible to add different sources that provide deletable keys. In SDOS, these "key sources" contain the deletable key and they provide en/decryption of the root key. As shown in Figure 4.4, the key source only processes the root key and no other parts of the Key-Cascade data structure. Similarly, the Key-Cascade implementation in SDOS only processes the nodes and root key. The deletable key always stays inside the key source.

In the Key-Cascade theory, I introduced the properties that the deletable root key must have. With the decoupled root key, these properties can now be defined on the *deletable key source* rather than on the *deletable key itself*:

- The deletable key source encrypts and decrypts a given root key when requested.
- For this, the deletable key source contains a current deletable key.
- The deletable key source replaces its current deletable key with a new deletable key when requested.
- Once replaced, the old deletable key must not be recoverable/restorable (i.e., securely deleted).

SDOS manages a pool of key source instances, one instance for each encrypted container. Therefore, key sources are mapped to a certain container on instantiation and subsequent operations do not need to contain a reference to the container. The key sources are integrated with SDOS by an interface with the following operations.

- `decrypt_current_key`: The key source receives the encrypted root key, decrypts it, and returns the plain root key.
- `encrypt_next_key`: The key source receives the new plain root key. It securely deletes and replaces its deletable key and then encrypts the root key with this new deletable key. It then returns the encrypted root key.

  This operation is used to securely replace the deletable key before re-keying the Key-Cascade.
- `initialize_root_key`: Is used to initialize the first root key and the deletable key for a new Key-Cascade. During initialization, a current deletable key, that is empty, is allowed in the key source. With this, the semantics of this initialize operation are the same as in the above `encrypt_next_key` operation.
- Additional authenticate/lock operations are provided depending on the type of key source. These implement authentication before using a key source, or lock the key source by clearing a present authentication.

These operations are used to perform the following three interactions with the key source: i) To initialize a root key for a new Key-Cascade (i.e., for a new storage container). ii) To decrypt the current root key with the current deletable key. iii) To securely replace the current deletable key with a new one and encrypt a new root key.

Figure 4.5 shows the interaction for initializing a root key to be used with a new Key-Cascade; this interaction is done when a new storage container is created by a user. First, SDOS generates a random root key which will be the current root key that is valid until the first re-keying operation. This key is then loaded into the source with the `initialize_root_key` operation. The key source then generates its current deletable key and uses it to encrypt the provided root key. The encrypted root key is returned so that SDOS can store it.

Figure 4.6 shows the interaction for decrypting the current root key. This interaction is done whenever SDOS needs to read Key-Cascade nodes (when

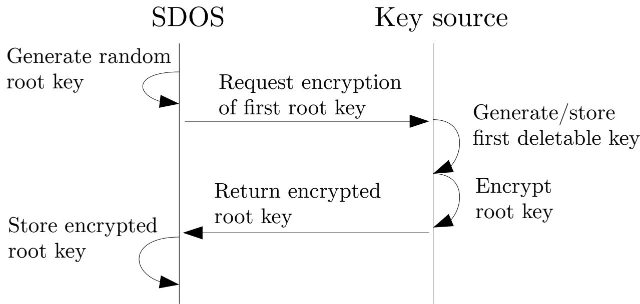

Figure 4.5: Key source interaction: initialize a new root key with the `initialize_root_key` operation.

reading/writing data and during re-keying). SDOS first loads the encrypted root key from the store and loads it into the key source, which will decrypt it using the current deletable key. After the decrypted root key is returned, SDOS can use it to decrypt and read the root node. SDOS can cache this response to avoid using the key source with every Key-Cascade operation.

Figure 4.7 shows the interaction for replacing and securely deleting the current deletable key. This interaction is used only after re-keying the Key-Cascade, in order to encrypt the new root key, and assures that the old root key can not be accessed anymore. During re-keying, SDOS created a next root key which it now loads into the key source. The key source then securely deletes its current deletable key and generates the next one. This key is then used to encrypt the next root key provided by SDOS. After SDOS received the encrypted root key, it stores this new key and the modified nodes, which completes re-keying.

Three different types of key sources are implemented in SDOS[1]: static key, password-derived key, and Trusted Platform Module. The "static key" key source contains a hard-coded deletable key. It is useful as an interface documentation, for test and development, as well as for benchmarking the Key-Cascade operations. The "password-derived key" key source is useful

[1] See: https://github.com/sdos/sdos-core/blob/master/mcm/sdos/core/MasterKeySource.py

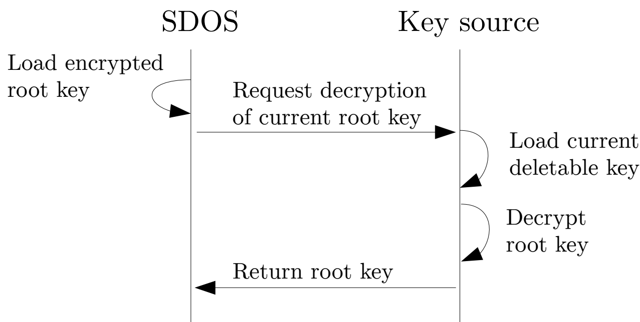

Figure 4.6: Key source interaction: decrypt the current root key with the `decrypt_current_key` operation.

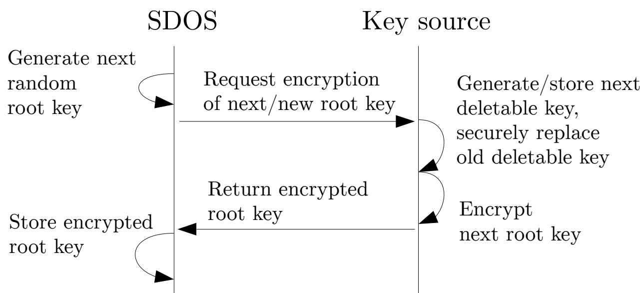

Figure 4.7: Key source interaction: securely delete and replace the deletable key with the `encrypt_next_key` operation.

for demonstration purposes since it lets the user experience what the key source does. Finally, the "Trusted Platform Module" key source provides a practically usable implementation of a secure key source. In the following, I explain these three types in more detail.

4.3.1 Key Source: Static Key

This deletable key source is only useful for evaluation and benchmarking purposes; it is included here for completeness and because it was used for the performance evaluations in Section 5.6. A static key is used for all operations

and is never replaced. This key source satisfies the functional requirements from above and therefore can be used as a key source. But it does not satisfy the non-functional requirement secure-deletion, as it never replaces the deletable key.

4.3.2 Key Source: Password-Derived Key

This implementation uses a human as the deletable key source. It is useful to help understand the role of the key source in SDOS and could be practically used in a single-user, a personal cloud storage, or a personal disk encryption scenario.

The deletable key is derived from a password memorized by the user. The implementation of this key source receives this password when the user wants to decrypt the root key and then uses a regular key-derivation algorithm to derive the current deletable key from the password. In SDOS, the password can be kept in memory in order to reduce user interactions.

In order to securely replace the deletable key, the user must provide a new password and *securely forget* the old one.

4.3.3 Key Source: Trusted Platform Module

This deletable key source uses a Trusted Platform Module in order to store the deletable key and perform en/decryption of the root key. Section 2.3 details the features and limitations of Trusted Platform Modules and compares them to other cryptographic hardware.

This TPM key source can be instantiated multiple times for different storage containers. It will use a different key in a different internal storage location for each container. The number of supported keys depends on the capacity of the used TPM. Authentication before using the TPM is configured on the TPM level, outside of SDOS. The TPM improves security over other key sources since it guarantees secure deletion of the internal keys. SDOS uses non-migratable keys inside the TPM to avoid key leakage of the deletable key. When SDOS needs access to the root key, it loads the encrypted root

key into the TPM. The TPM then decrypts that root key and SDOS loads it from the TPM memory.

This key source provides secure deletion of the deletable key in practice and shows one possible configuration of SDOS that satisfies the security requirements of Section 1.6.

4.4 Object ID Index

This extension allows using arbitrary object names instead of consecutive, numerical identifiers.

As discussed in Section 3.2.1, the Key-Cascade uses a perfect k-ary (static) tree as the underlying data structure. One of the benefits of this structure is that node labels are sufficient to represent the edges between nodes. No information about the edges needs to be stored inside the nodes, or in a separate data structure, enabling offline calculation of paths (see discussion in Section 3.6).

The ID calculations used to navigate the Key-Cascade tree structure make use of numerical identifiers for the leaves. These leaves represent the individual object keys, therefore each object key has such a numerical ID. These IDs are called **object ID** (oid). So each object encryption key is identified by its oid and this ID also identifies the object that gets encrypted with that key. All read, write, and delete operations on the Key-Cascade need this oid in order to identify the correct object key location in the data structure.

Until this point, all the Key-Cascade examples in this thesis used these $oids$ to identify the objects. This means that the object names are the numerical $oids$, which is not useful in practice. For this reason, this extension adds an index to SDOS that maps arbitrary Swift **object names** to the internal **object IDs** (oid). With this, any arbitrary name can be used for the objects and the index maps this name to the internal oid.

These object names are arbitrary strings, which means they are from a very large name space. Our object identifiers are from a smaller space of fixed size due to the static tree. Because of this imbalance, I use an index to explicitly

map object names to object identifiers, instead of using an implicit mapping function. A benefit of using an explicit mapping is the ability to map any object name to any object ID, which enables the use of an allocation strategy. In the prototype, SDOS always uses the first (lowest integer) available object ID in order to minimize the number of utilized nodes and avoid sparse trees. However, other allocation strategies are possible. For example, if a group of objects is known to get deleted together, their object keys could be grouped into neighboring nodes.

The index is implemented as a hash table with collision resolution in the SDOS prototype. The keys of this hash table are the object names, the values are the $oids$. No reverse mapping (oid to object name) is implemented at this time, as none of the functions in SDOS require this. The hash table gets stored as an object inside the appropriate SDOS management container in Swift. The index is stored unencrypted since this does not affect the secure-deletion property. This is the only additional data structure that needs to be persisted along with the Key-Cascade.

4.5 Free List

This extension adds a data structure to keep track of free object keys.

When a new object should be written, a new object key and therefore object ID (oid) must be allocated. In theory, these IDs can simply be incremented with each insertion, starting at 0. But in a production environment, insertions and deletions occur arbitrarily interleaved. For this reason, SDOS needs to know which object identifiers are available for inserting new objects. Otherwise, no more insertions would be possible once the last oid was reached, even if deletions freed other oids.

I implemented a free list in SDOS in the form of a bit vector which keeps track of the free and allocated object identifiers. For the Key-Cascade of Example 2 (Section 3.5), a bit vector with 16,777,216 positions is needed. This requires 2 megabytes of storage. The free list can always be derived from the Key-Cascade itself by scanning the leaf nodes (object-key nodes)

for empty slots. It is only needed for performance reasons and does not get persisted.

4.6 Node Cache

This extension adds caching for the Key-Cascade nodes in order to improve performance.

Each node of the Key-Cascade is stored as an individual object inside the Swift Object Store. All operations on the Key-Cascade involve at least reading as many nodes as the cascade has levels and the root node is needed in every operation. For these reasons, performance can be improved by caching the nodes.

In SDOS, the node cache is implemented transparently between a node pool and a Swift API so that it can be disabled for performance comparison. The node pool is explained in Section 5.4, it coordinates concurrent access to node objects from multiple threads. When a node is requested from the pool, the pool will request this node from its underlying storage provider (either the cache or Swift) and receive a bytestream comprising a serialized and encrypted node.

The node cache only holds serialized and encrypted nodes in the same form as the object store itself. Therefore, the cache does not need to be stored in a trusted storage location, but can be implemented on any available storage system that has faster access times than the Swift object store.

The pool and node cache are designed in this way so that decrypted/deserialized nodes are kept in memory for the shortest time possible. This reduces the risk of key leakage through memory dumps.

4.7 Batch-Delete Log

The batch-delete log allows deferring the secure deletion (re-keying) of objects so that multiple deletions (a batch) can be processed at once.

Without the batch-delete log, each deletion of an object immediately

triggers re-keying (see Section 3.7) of the Key-Cascade, which eventually requires replacing the deletable key. Immediate re-keying has several downsides that are addressed by the batch-delete log:

- Immediate re-keying causes a large run-time overhead. As discussed in Section 5.6, cascaded re-keying is the most expensive operation because it requires read and write operations for a whole tree path.
- Cascaded re-keying requires locking and exclusive access to the Key-Cascade nodes. No other requests can be processed while cascaded re-keying is running (see Section 5.4).
- Secure deletion and replacement of the deletable key can be a slow and expensive operation that might include human interaction, based on the type of key source used (see Section 4.3).

With the batch-delete log, SDOS writes the names of deleted objects into a log and immediately forwards the delete request to the Swift Object Store. The object is therefore deleted from Swift and can not be retrieved anymore, but the object keys are still in the Key-Cascade and the current deletable key is still valid to decrypt them. This means that the objects may still be reconstructed, because secure deletion (re-keying) has not yet occurred.

A user can then trigger re-keying at a later time, if the delete log has one or more entries. Cascaded re-keying will commence as described in Section 3.7, but in a batch-optimized manner. With this batch optimization, re-keying will only visit each necessary node once. This is achieved by processing the nodes depth-first so that all affected child nodes are re-keyed before the modified parent is committed.

Batch deletion brings the following benefits:

- Cheaper object deletions. Deleting an object only has the baseline SDOS proxy overhead.
- Better overall re-keying efficiency. With immediate re-keying, deleting n objects causes n read as well as write requests for node 0[1]. With batch-

[1] In any case, independent of Key-Cascade geometry. There may be additional requests for further nodes depending on the geometry.

optimized re-keying, exactly one read and write request is needed for any affected node.

- Re-keying at user discretion. Re-keying requires replacing the deletable key.
- Time-based cryptographic deletion (see Section 2.1) can be supported by the batch-delete log.

Use of the batch-delete log can be enabled and disabled individually for each container that uses cryptographic deletion.

The log, as it is implemented in SDOS, stores the Swift object names (Strings) and not the internal object IDs (*oids*, Integers). Both are usable to provide a delete log and present certain advantages for handling re-insertions. When the delete log is active, a user can delete objects and possibly re-insert objects with the same names as the deleted ones before processing the batch delete. The SDOS prototype currently does not handle this case in a special way and using it in this manner will lead to the re-inserted objects losing their object keys when the batch deletion is processed. However, different mitigation strategies for this case are conceivable and depending on their design, they will benefit from either a object name based delete log, or an *oid* based one.

Chapter 5

SDOS: The Key-Cascade Implementation

In this Chapter, I present the second contribution (C2) of this thesis: a practically usable implementation of Contribution C1, the Key-Cascade method.

SDOS stands for the “Secure Delete Object Store”, as this prototype is implemented on top of the Swift Object Store by the OpenStack project. The system is designed as an API proxy for the REST-based Swift API so that it can be integrated transparently between any unmodified Swift client and server. I chose this approach because of the motivation for this work: supporting cryptographic deletion in cloud storage systems.

The reasons for choosing the Swift Object Store instead of other storage services are:

- Object storage services are ubiquitous among cloud providers[1].
- Such object stores are the de-facto standard for large-scale storage outsourcing, especially for back-up and archiving purposes. Back-up

[1] IBM Bluemix Swift object store, Google Cloud Storage (GCS), Amazon Simple Storage Service (S3), Microsoft Azure blob storage.

and archiving are also the applications where secure deletion is usually practiced today.

- Swift is one of the largest open-source object stores today; using this open-source implementation allows deployments outside the cloud, or in private clouds as well. IBM even uses Swift as their object storage service, while all other providers use proprietary implementations. Their data models are very similar so that SDOS could be adapted with little effort.

The reasons for choosing the API proxy approach are its benefits over the two alternatives: integration into the server and integration into the client.

- Integrating cryptographic deletion into the object store server component would prohibit the use of any of the existing cloud object storage services. It would also require more trust in the cloud provider, since it allows the provider access to the encryption keys.
- Implementing cryptographic deletion in the client application would require modifying existing applications, raising the entry barrier to using the solution.

However, there is also a disadvantage to the API proxy approach: It introduces an inherent overhead, since all operations have to pass through an additional component (the API proxy). I discuss this overhead in the performance evaluation in Section 5.6.

SDOS is implemented using the Python programming language and runs as a multi-threaded WSGI[1] application. WSGI is a Python interface for implementing HTTP-serving applications and multiple application servers exist that can run WSGI applications like SDOS. The SDOS code can be found on Github[2] including a fully automated deployment procedure in Docker.

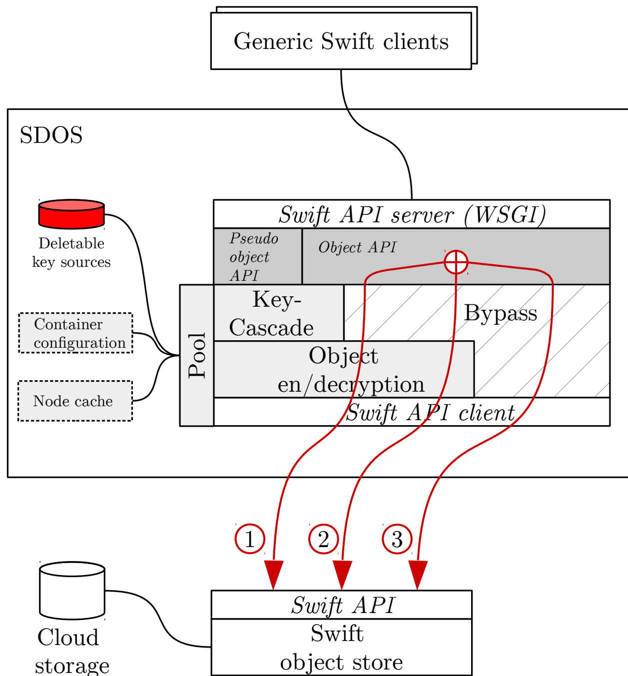

Figure 5.1: Architecture of the SDOS prototype showing the three operational modes.

5.1 System Architecture

The high-level architecture of SDOS is shown in Figure 5.1.

A WSGI-based HTTP server waits for requests from clients and triggers all further operations in SDOS. Each request is executed in a new thread, so that new requests can be accepted immediately. SDOS supports parallel access by multiple clients with different Swift accounts and can manage multiple Key-Cascades and deletable-key source instances for different containers[1] in Swift.

[1]https://www.python.org/dev/peps/pep-0333/
[2]https://github.com/sdos/sdos-core
[1]Swift stores objects in containers, no nesting of containers is possible.

The requests are processed by a stack of components. The first is an internal object, or pseudo object API, that dispatches the request based on type. Usual object requests are handled by the object API. The pseudo-object API allows access to SDOS runtime information over regular Swift API operations. It is explained in detail in Section 5.5. The Key-Cascade component implements the operations necessary for cryptographic deletion; it operates on the nodes that contain the encryption keys. The next layer provides encryption and decryption functionality for the actual data objects during read or write requests. The last layer in this stack is the official Swift client which interfaces with the cloud object store in order to read and write data objects, encrypted Key-Cascade nodes as well as container metadata.

Three modes of operation are supported based on configuration attached as metadata to containers in Swift: 1) Object encryption with cryptographic deletion using the Key-Cascade. 2) Object encryption without cryptographic deletion. 3) Bypassing, that directly forwards requests to Swift.

The configuration from the metadata also determines the Key-Cascade geometry by containing the parameters introduced in Section 3.4, as well as the type of deletable key that is used (see Section 4.3). With this design, SDOS can be configured differently for each container.

SDOS uses an object pool (here: in-memory program runtime objects) in order to coordinate parallel access to internal data structures from the multiple threads. This pool only contains object instances that are currently used. Some object instances can be used in parallel, which is managed by this pool. Its concurrency behavior is detailed in Section 5.4.

The three attached data types are: Deletable key sources (see Section 4.3), container configuration and node cache.

The **deletable-key sources** provide access to an encryption key that is needed for requests of type 1 and 2. When cryptographic deletion is used, this key is used as the deletable key for the Key-Cascade. In the case of regular encryption, the key is used as a master encryption key for encrypting the objects. The memory or storage location of these objects depends on the type of key source. All key sources implement a lock and unlock operation. While a key source is unlocked, it is considered in-use and available in the

pool.

The **container configuration** store is an in-memory cache for the runtime settings of known containers. It is populated by the Swift container metadata, where container configuration is persisted, as well as by runtime data. This cache contains information whether a container uses encryption or cryptographic deletion, about the type of key source and Key-Cascade geometry. It improves access times since it allows SDOS to immediately determine the correct settings to handle a request. Without this cache, the container metadata must be retrieved from Swift before processing requests, often doubling response times.

The container configuration settings from the Swift container metadata are committed to the cache whenever a response from Swift includes the metadata. The contents of this metadata are described in Section 5.2. Swift only returns this metadata in the response to an explicit request for container metadata. For this reason, SDOS usually makes such a metadata request once for each container, when the container is first used. Subsequent user access to the container does not trigger metadata requests anymore, since the cache is now used.

Explicit metadata requests by SDOS are not needed in three cases: i) when a container was used before (i.e., this cache can be used). ii) When the user issues a container metadata request on their own, before accessing the container in any other way. iii) When the container was newly created during this run time of SDOS. In this third case, the container-create request, or subsequent update requests, had to contain all the relevant metadata. SDOS also populates this configuration cache from container create requests.

Besides the configuration settings from container metadata, this container configuration cache also contains non-persistent runtime data. This includes the batch-delete log for each container where cryptographic deletion and batch deletion are configured. A batch-delete log is considered in-use when it contains entries; during that time it is kept in memory by the pool. The batch-delete log is not persisted to container metadata or anywhere else, so that it must be processed during runtime of the SDOS process.

The **node cache** is an in-memory cache that contains encrypted Key-

Cascade nodes in the same serialized binary format that is used for persistence in the object store. The node cache acts as a buffer between the pool and the object store. A node is considered in-use while it is read or modified by a Key-Cascade operation. The pool contains decrypted, deserialized nodes while they are in use by a thread. If a node was modified in the pool and is no longer in-use, it is serialized and returned to this cache. The cache regularly flushes modified nodes back to the object store for persistence.

5.2 Swift Integration

Swift's data model consists of accounts, containers, and objects in a hierarchy. Figure 5.2 shows that each of those entities has attached metadata that may also contain custom fields. Containers belong to one or more accounts and comprise the top-level. Containers only contain objects; no hierarchy of containers is possible. If a hierarchical organization of objects is desired, like in a file system, the client has to flatten the hierarchy through the use of pseudo paths in the object names.

The entities in Swift can be manipulated via the REST operations GET (read), HEAD (read metadata only), POST/PUT (write, create) and DELETE.

SDOS transparently forwards all account and container requests to Swift,

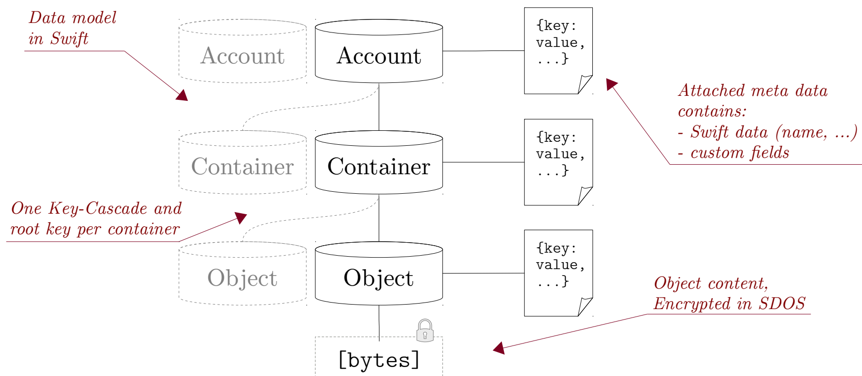

Figure 5.2: Account, container, and object hierarchy in Swift.

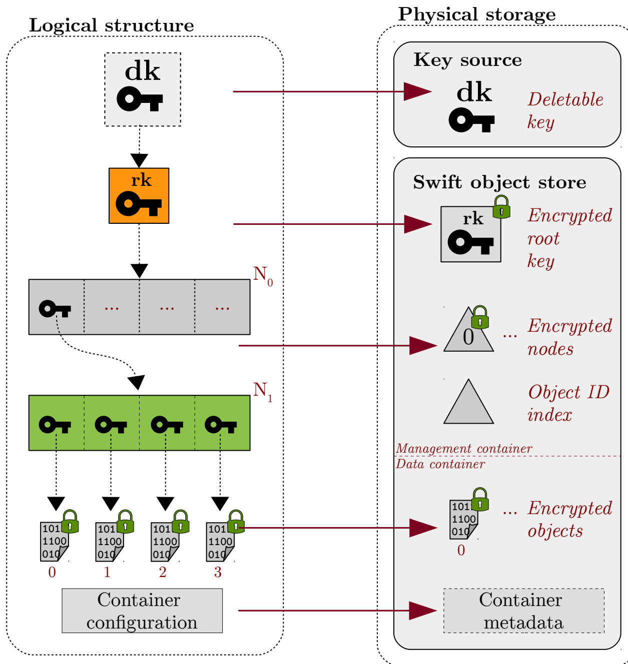

Figure 5.3: Mapping of SDOS data structures to Swift Object Store.

but intercepts object requests if they belong to a container that has encryption or cryptographic deletion configured. If a container was created or accessed before, this information can be found in the container configuration cache. Otherwise, SDOS reads this information from Swift with a HEAD request on the container.

In the following, I first discuss the Swift integration regarding storage and those regarding operations.

5.2.1 Swift Storage

SDOS applies the Key-Cascade concept to the container level so that one Key-Cascade holds the encryption keys for all objects inside a single container. A separate management container is used for the SDOS data structures. The

name of this management container is derived from the name of the actual data container. This separation is illustrated in Figure 5.3.

Separating management and data containers has the benefit that the entire name space of the data container remains available for data objects.

The **data container** in Swift contains the following:

- Encrypted objects. The encrypted objects are stored under their unencrypted name; SDOS does not encrypt the object names or any of the object metadata fields.
- Container metadata. Swift supports custom metadata fields on containers as well as objects. The client can set certain fields during container creation in order to enable and configure SDOS. These fields are never modified by SDOS. These fields are:

  `encryption-enable`: A Boolean indicating whether data encryption should be used.

  `key_cascade-enable`: A Boolean indicating whether cryptographic deletion through Key-Cascade key management should be used.

  `key_source-type`: An enumeration type describing the type of key source.

  `key_source-key_id`: A number that correlates a key with a container. Some key sources, like Trusted Platform Modules, manage multiple keys and need this correlation.

  `key_cascade-node_size`: A number specifying the node size for the Key-Cascade.

  `key_cascade-tree_height`: A number specifying the height of the Key-Cascade.

  `batch_delete-enable`: A Boolean indicating whether the batch-delete log function should be used.

The **management container** in Swift contains the following:

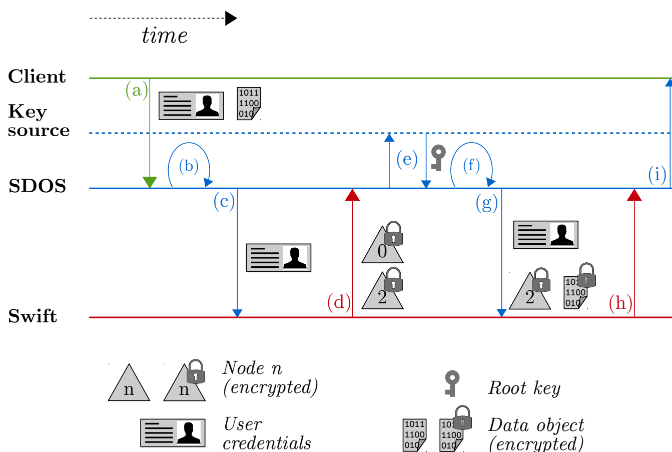

Figure 5.4: Sequence of operations for storing a new object.

- The encrypted root key for the Key-Cascade. This root key can be decrypted with the corresponding deletable key inside the key source. In the case of regular encryption (without cryptographic deletion), an encrypted master key for the objects is stored here.
- The encrypted nodes comprising the Key-Cascade.
- The unencrypted object ID index that maps the object names from Swift to the internal, numerical object key IDs.

5.2.2 Swift Operations

SDOS only intercepts and modifies requests on objects inside containers that have encryption or cryptographic deletion enabled. Furthermore, only the GET (read), POST (write) and DELETE operation must be considered. The HEAD operation only reads the unencrypted metadata directly from Swift and the PUT operation has the same semantics as POST in Swift's protocol.

In the following, I describe the effects of the write, read and delete operation on containers with cryptographic deletion enabled.

Some steps are common to all three operations: In the first operation on

a container, SDOS reads the container metadata in order to populate the container configuration in the pool. Next, the object ID index is loaded into the memory of the pool. Then, the key source is initialized. This requires user (or programmatic application) input depending on the type of key source. This input must be provided via the pseudo-object API (see Section 5.5). Initialization of the key source also includes retrieving and decrypting the root key from Swift.

Write operation: The write operation is shown in Figure 5.4.

(a) The client initiates the request by sending credentials and object to SDOS via the Swift API. These client credentials are used by SDOS in order to interact with Swift on the clients behalf; SDOS has no separate set of credentials for Swift.

(b) SDOS reads the container configuration and allocates a free object key ID using the free-list and object ID index.

(c) SDOS uses the clients credentials in order to retrieve the necessary nodes from Swift. I use the same example as in Section 3.6, therefore nodes 0 and 2 are requested from Swift.

(d) Swift returns the encrypted nodes.

(e) SDOS interacts with the appropriate key source in order to retrieve the decrypted root key.

(f) SDOS uses the root key to decrypt the two nodes and inserts a newly generated object key into node 2. Using this new object key, SDOS encrypts the object that the client sent in step (a).

(g) SDOS then writes the modified node 2 and encrypted object into Swift.

(h) Swift confirms the success of the operation to SDOS.

(i) SDOS forwards the reply of the object write request to the client.

Authentication and authorization are verified in the same way in the read and delete operation. They are omitted for brevity in these operations.

Read operation: SDOS first finds the corresponding object-key ID by looking up the requested object name in the object ID index. ID calculations then determine the path and slots. The Key-Cascade layer in SDOS loads and decrypts the necessary nodes via the pool and cache. After decryption, the object key and the request are passed on to the decryption layer. Here, the object is retrieved from Swift, decrypted and returned to the client.

Delete operation: The delete operation consists of two steps. First, SDOS forwards the delete request for the object to Swift and returns the result code to the client. The object is now removed from Swift, but not securely deleted yet. The object key is still present in the Key-Cascade and can still be decrypted with the current deletable key.

The second step depends on the batch-delete log. If it is enabled, SDOS only appends the object name to the log and the operation is concluded for the time being.

If the batch-delete log is not enabled, then cryptographic deletion is performed immediately. In this case, SDOS first performs cascaded re-keying as described in Section 3.7. But the modified nodes are not written to Swift yet. SDOS then interacts with the key source in order to securely replace the deletable key and encrypt the new root key as described in Section 4.3. SDOS only writes the modified nodes to Swift after the deletable key has been securely replaced. If the key source fails to perform this operation, SDOS discards the modified nodes.

Executing the deferred cryptographic deletion for the objects in the batch-delete log follows the same pattern. There are only two differences: i) cascaded re-keying is performed on a list of object keys instead of a single key and ii) the operation is triggered via the pseudo-object API rather than by an individual object deletion.

5.3 Authorization and Multi-Client Support

SDOS supports requests from multiple clients belonging to different accounts, or tenants. These clients may have different sets of permissions for accessing

the containers and objects. I support this scenario by using Swift itself as the authentication and authorization provider. Swift manages permissions on a per-container basis and SDOS sets the same permissions for the SDOS management container as the data container has. When a user issues an operation to the SDOS gateway, they use their regular Swift credentials since SDOS has the same API as Swift. SDOS then uses these credentials in order to interact with Swift on the user's behalf. Swift inherently authenticates the user and authorizes the operation this way.

If the authentication and authorization succeeds, the credentials are correct and SDOS uses them to access the necessary Key-Cascade data structures in the appropriate SDOS management container as well as the associated data object that the user has requested. Since these two containers share the same permissions, the user's credentials are valid for both. SDOS only controls access to the cached root key inside the key source, all other data structures are managed by Swift. SDOS never releases keys in any way but only uses them internally to process the Key-Cascade data structures. If a user's credentials are validated by Swift and all required data structures can be retrieved, the request is considered valid. The SDOS gateway then uses the appropriate root key to process the Key-Cascade and to execute the requested operation.

With this approach, I reduce overhead since I avoid having a separate authentication provider or permission management. I don't need to maintain a user registry and I also don't need to have a separate Swift user account (i.e., a proxy user) that the SDOS gateway uses for accessing Swift.

Swift itself is part of the OpenStack project which also includes an authentication service called Keystone. Swift itself checks the authorization (permissions) of a user, while keystone authenticates the user (tenant name, user name, and password validation).

5.4 Multi-threading in SDOS

SDOS uses multi-threading in order to provide parallel processing of requests. This provides potential performance benefits over serial processing in two cases:

1. A single client application, even if it runs serially itself, may issue a second request before receiving the response to the first. This is a common behavior in Swift clients since the Swift protocol does not support batch requests. For example, when a client writes 10 objects, it will issue all 10 requests in sequence and then wait for the 10 responses asynchronously.
2. When multiple clients are involved, their requests will be inherently non-serialized.

Both of these cases lead to the situation that SDOS receives a new request, while still processing a previous one. Parallel processing of these requests enables SDOS to potentially achieve a higher throughput. In the following, I first discuss the operations involved in processing a request and show which are parallelizable. I then discuss the differences between multi-threaded, multi-processed and distributed execution of Key-Cascade operations.

5.4.1 Parallel Request Processing

Parallel processing in SDOS only considers which internal Key-Cascade data structures can be used concurrently. Operations on the Swift Object Store are parallelized or serialized by Swift itself. Furthermore, only requests that affect the same container (i.e., that use the same Key-Cascade) are considered. No relevant data structures are shared between containers, so that requests on different containers are always parallelized.

Swift does not have a transaction isolation concept for requests so that multiple clients, reading and writing the same object, can overwrite each other's updates. For this reason, SDOS needs a method outside of Swift for

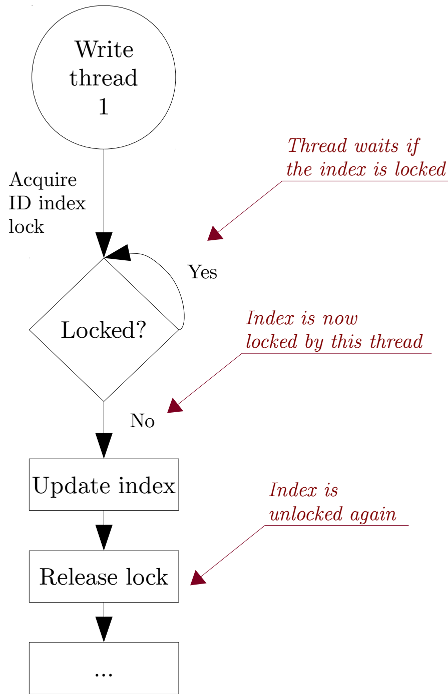

Figure 5.5: Thread synchronization during writing.

synchronizing object access in order to prevent such write-write or read-write conflicts [HR99].

SDOS uses an object pool in order to coordinate access to internal data structures from multiple threads. This pool uses the singleton pattern (see [GHJV95]) when instantiating objects in order to assure that all data are consistent between the different threads. The singleton objects exist in exactly one instance that is accessible by multiple threads at the same time. By default, all operations on the singleton objects occur concurrently; only two operations require exclusive access to certain data structures: object ID allocation and cascaded re-keying.

Reading objects: Read operations modify none of the data structures, they don't require exclusive access and can be executed concurrently.

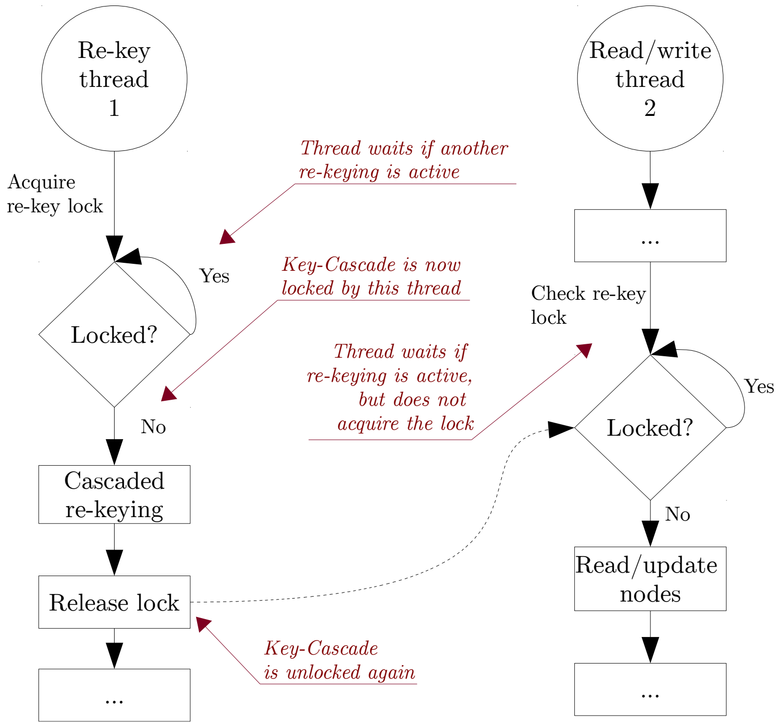

Figure 5.6: Thread synchronization during cascaded re-keying.

Writing new objects: Writing new objects leads to modifications in the object ID index as well as the Key-Cascade nodes. Serialization is however only needed in the beginning when the object ID is allocated. Figure 5.5 shows the sequence when a thread writes a new object. Synchronization between the threads is provided by locking. The thread tries to acquire a lock[1] and waits if it is already locked. After updating the index, the thread releases the lock, allowing another thread to acquire it.

Cascaded re-keying: This operation may not run concurrently with other operations. The reason for this is that it replaces the root key after success-

[1]The lock can only be acquired once at any time. When acquired, the lock is "locked". In SDOS, the "lock" object from the python standard library for multi-threading is used.

fully modifying some nodes. Any other access, that happens at the same time, experiences the following problem: Depending on the timing, the thread may read old nodes but a new root key, or new nodes but an old root key.

I found three possible solutions to this problem: i) Catch these specific errors in the read and write threads and re-try the operation later. ii) Synchronize individual node and root key access to avoid stale reads and writes. iii) Allow the re-keying operation exclusive access.

I decided to implement the third approach in SDOS because it is the least error prone and most reliable solution. The second approach, fine-grained locking, potentially provides higher performance. But since I intend re-keying to be a rarely executed operation, I prefer a safer solution.

Figure 5.6 illustrates this cascade-locking approach. A single lock object exists on each Key-Cascade. A re-keying thread acquires this lock, then starts executing the cascaded re-keying operation. Read and write threads check if this lock is acquired and wait in that case. Once finished, the re-keying thread releases the lock and the other threads are woken up.

The effect of this approach is that all read and write threads wait for cascaded re-keying to finish. It is however still possible that a slow read or write thread, that started before the re-keying thread, reads or writes a stale object. In SDOS, this situation is handled by approach i) introduced above; catching the error and repeating the operation later. The repeated execution will then wait for the re-keying lock to be released. It is generally no problem to fail Swift requests. Swift itself fails requests under certain conditions and clients are expected to repeat them. The situation can however be avoided by the following two approaches: 1) Read and write threads do not just wait for the cascade lock, but acquire it as well. This serializes all operations in the Key-Cascade and eliminates concurrency issues, at the expense of throughput performance. 2) Implement a barrier condition so that cascaded re-keying waits for all read and write threads to finish, after acquiring the lock and before starting its execution.

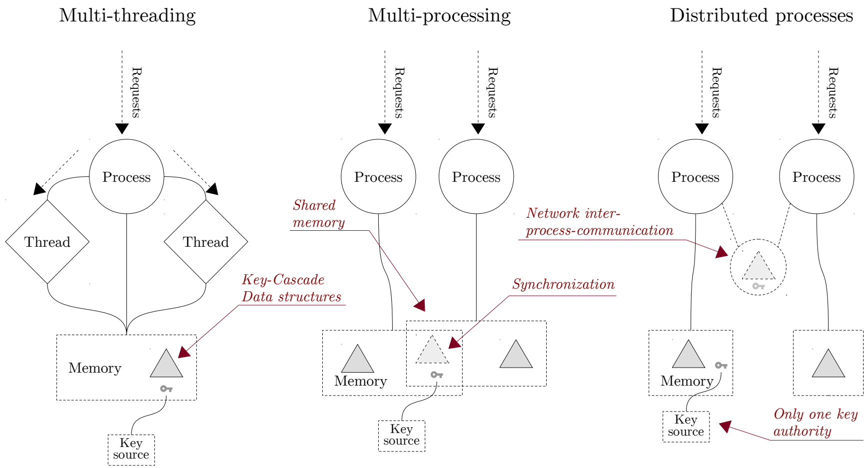

Figure 5.7: Parallel processing methods with distributed data structures and key source access.

5.4.2 Implementation of Parallel Processing

I consider three types of parallel-processing for a Key-Cascade implementation: multi-threading (a), multi-processing (b), and distributed processing (c). The SDOS prototype implements multi-threading as explained above. Figure 5.7 illustrates the differences for the Key-Cascade data structures.

Multi-threading: This type of parallel processing runs a single operating system process. This "main process" can spawn any number of threads, each of which has its own stack. The threads however can access the same memory (heap). Blocking operations, like disk input/output, only block the thread that issued them and not the others. multi-threading is reasonably efficient in modern operating systems and usually the preferred approach over multi-processing. The main downside to multi-threading is the implementation of the main thread. This main thread must spawn the other threads and delegate requests to them. If the main thread is blocked or slow to delegate requests, the whole process throughput suffers. It is also possible that program errors in one of the threads cause the operating system to kill the

entire process, including all threads.

Multi-processing: This approach uses multiple, separate operating system processes to distribute the work. This provides a higher degree of isolation in case of errors. The main performance benefit comes from the way requests are delegated. The operating system can already delegate requests to individual subprocesses. This is especially useful in the case of networking applications, like the HTTP server in SDOS. The main process only spawns subprocesses and registers them in the operating system. Less work is done in the main process than in the main thread in the multi-threading case. There is usually a lower risk of blocking the main process than there is of blocking the main thread in multi-threading.

The main downside to multi-processing is that the processes have isolated memory regions. Any data that must be accessed by all processes, must be explicitly exchanged via interprocess communication methods (IPC). One of those IPC methods is a shared memory region that processes can use to synchronize their operations on shared data structures. This makes multi-processing less suitable for applications that primarily operate on the same data.

Distributed processing: This method is similar to multi-processing, but the individual processes do not run on the same host. In this case, a network-based inter-process-communication method is needed to synchronize access to shared data. Network communication introduces a higher latency between the processes that must be taken into account when designing a locking and synchronization strategy. Network latencies in distributed locking mean that locks are kept longer than necessary. For this reason, it is advantageous in distributed locking scenarios to implement fine-grained locking.

Accessing the key source is another operation that must be implemented with the networked IPC. In SDOS, the key source interface already works on the level of the root key (see Section 4.3). This interface enables the Key-Cascade implementation access to the current and after secure deletion/replacement, the new root key. The same interface could be exposed to distributed SDOS processes over the network.

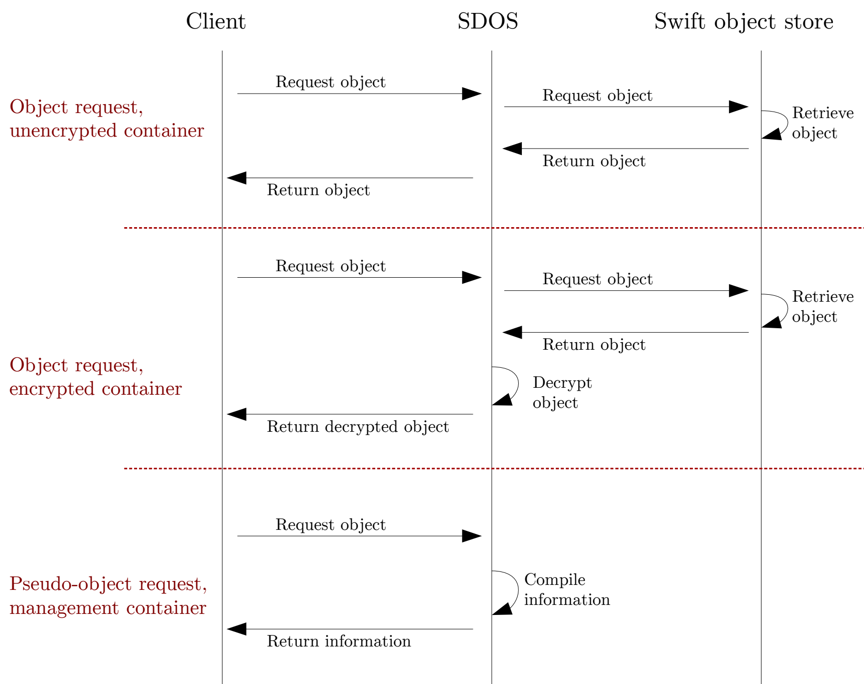

Figure 5.8: Pseudo-object API compared to requests on regular and encrypted objects. Showing object / information retrieval.

5.5 Pseudo-Object API

The pseudo-object interface in SDOS allows clients to execute custom operations in SDOS over the regular Swift REST API. These custom operations include unlocking the key source, retrieving information about the Key-Cascade and its capacity, as well as executing batch secure deletion. It is designed in a way that allows standard Swift clients to use this interface. See Figure 5.1 for a reference on the SDOS architecture.

This is achieved by SDOS responding to requests on non-existent pseudo-objects inside the management containers, as shown in the bottom of Figure 5.8. As mentioned above, SDOS creates a management container for

each encrypted “data” container. This management container holds the encrypted nodes, the root key and the object ID index. But SDOS also reacts to operations on non-existent objects with a certain name inside these containers. These objects represent operations on the pseudo-object interface. Operations that need an input from the client, for example, providing a password to unlock a key source, are realized by performing an object upload on the specific pseudo-object. Operations in which SDOS provides output to the client, e.g., retrieving the number of utilized keys, are realized by performing an object download.

The pseudo-object interface in SDOS provides the following operations for retrieving information:

`key_stats` Allows clients to retrieve information about the key source. This includes the type of key source, the state it is in, as well as a hash of the currently used root key.

`sdos_used_nodes` Allows clients to retrieve a list of the Key-Cascade nodes that are currently in use. This is a subset of all available nodes unless the Key-Cascade is fully utilized.

`sdos_objectid_index` Allows clients to retrieve the object ID index that shows which Swift object names are assigned to which object IDs (oid). This oid locates the encryption key for the object inside the Key-Cascade and so tells the client exactly where the keys for the objects are.

`sdos_key_cascade_stats` Allows the client to retrieve a JSON data structure that contains the Key-Cascade geometry and utilization statistics. Together with the above two operations, this allows a client to derive the structure of the Key-Cascade. This is used in the custom Bluebox user interface (see Section 6.2.2) for generating an interactive visualization of the tree structure and objects.

`sdos_slot_utilization` Allows clients to retrieve a compressed list of utilized $oids$. These object IDs identify all the slots in the leaf nodes, that contain the object keys. This list tells the client which slots are

used. This list is used in Bluebox for generating a usage visualization / key distribution graph. This utilization data could also be derived by a client from the object ID index.

`sdos_batch_delete_log` Allows clients to retrieve the batch-delete log. This is a list containing the names of objects that have been deleted and whose object keys are queued for cascaded re-keying.

The pseudo-object interface in SDOS provides the following operations for executing functions in SDOS:

`sdos_batch_delete_start` This operation starts the processing of the batch-delete log as explained in Section 4.7.

`sdos_key_unlock` This operation unlocks the key source and decrypts the current root key so that the Key-Cascade can be used. Depending on the type of key source, this operation needs user input (e.g. a password).

`sdos_key_lock` This operation removes the decrypted root key from memory so that the Key-Cascade can no longer be used. It also locks the key source if possible, depending on type.

`sdos_next_deletable_unlock` This operation makes the key source ready to securely delete and replace the current deletable key. Some realistic key sources may require user input before performing this operation. Two such examples are given by two of the key sources implemented in SDOS (key sources are discussed in Section 4.3). The password key source and the TPM key source. The password key source uses a user-provided password as the deletable key. Secure deletion and replacement in this case means querying the user for a new password. This operation allows the user to enter this password. Another such example is given by the TPM key source which can be configured to request additional authentication, before securely deleting and replacing an internal key. Here, the client uses this operation in order to relay the authentication to the TPM.

`sdos_next_deletable_lock` This operation undoes a previous unlocking as explained above. With the two examples above, this is only possible when the next deletable key is ready but has not been used yet by a cascaded re-keying operation.

This pseudo-object API, combined with the Key-Cascade configuration inside container metadata, make the SDOS implementation fully compliant with the Swift protocol and any Swift client, while still enabling custom functionality. This capability was for example used in the performance evaluation. Here, an existing Swift benchmarking tool could be used without modification in order to create, use and measure differently configured containers.

5.6 Performance Evaluation

The performance evaluation was conducted in order to answer a set of specific questions. In the following, I first state these questions, then describe the evaluation setup, and show the results.

- Q1: What is the cost of using the SDOS API proxy, compared to a direct client to server connection?
- Q2: What is the cost of encryption and Key-Cascade operations, compared to using encryption with a single master key?
- Q3: What are good node sizes and tree heights for a Key-Cascade?

I consider the answer to Question 2 the most relevant result of this evaluation. It directly represents the fitness of the Key-Cascade method, independent of the chosen implementation or environmental factors. This result states, what the cost is of adding cryptographic deletion to an already encrypted storage system. Together with the answer to Question 3, this gives insight into how a practical system could be laid out and what performance to expect from it.

The evaluation setup consists of two virtual machines running on a single host (server) with 12 Intel Xeon E5-2630 CPU cores at 2.30 GHz. Each

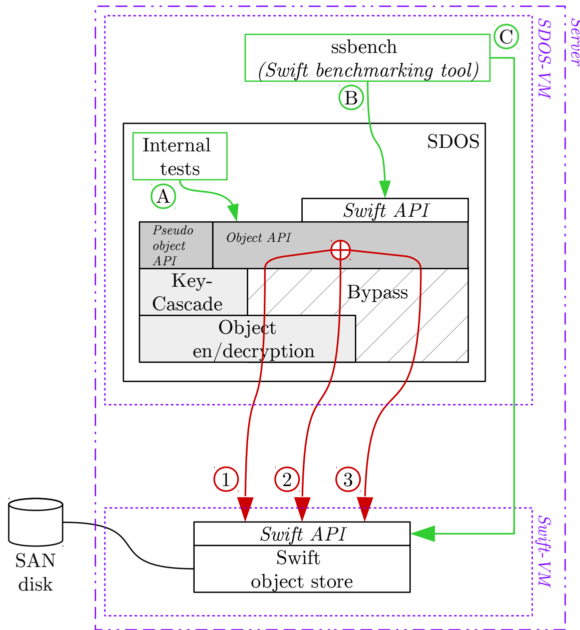

Figure 5.9: Layout of the evaluation setup for SDOS.

virtual machine was assigned 4 CPU cores and 32 GB of main memory, while the storage disk for Swift was provided by a large storage area network (SAN) attached via 16 Gbit/s Fibre Channel. Both virtual machines ran an Ubuntu 14.04 operating system with a 3.13 Linux kernel and were connected by a virtual network limited to 1 Gbit/s. The first virtual machine (Swift-VM) was running a single-node production build of Swift version 2.5 (indicated as "Swift Object Store" in Figure 5.9), while the second one (SDOS-VM) was running the SDOS component indicated as "SDOS". A single-server layout with two virtual machines was chosen for two reasons: Firstly, to eliminate outside noise or network related artifacts and, secondly, to isolate the two components. The server has more resources than both virtual machines use together, so that good isolation should be given.

All measurements were done by issuing requests to SDOS or directly to

Swift and measuring the response time as well as the throughput of requests over time. This was done with two different tools generating requests: An internal SDOS component as well as `ssbench`, an open-source benchmarking tool for Swift. These tools were used in a total of three configurations, indicated as A, B, and C in Figure 5.9. Configuration **A (internal)** uses the SDOS internal testing component to generate requests and so excludes the cost of the API proxy, but still allows measuring the Key-Cascade and encryption. Configurations B and C use the `ssbench` tool. In **B (proxy)**, `ssbench` connects to the SDOS API-proxy, while it is directly connected to Swift in Configuration **C (swift direct)**.

The SDOS configurations, indicated as 1, 2, and 3 in Figure 5.9, represent the three modes of operation introduced in Section 5.1. They determine how SDOS processes the requests internally, before forwarding them to Swift. Configuration **3 (bypass)** uses the bypass function in SDOS so that all requests, either from the internal or Swift API, are directly forwarded to Swift. This allows measuring the pure cost of the API proxy.

The next configuration **2 (enc. only)** adds the object encryption feature with a fixed key. These results represent the cost of encrypting the data objects, compared to the bypass mode. Finally, configuration **1 (enc. + KC)** enables object encryption as well as the Key-Cascade key management.

For benchmarks with the internal tester (Configuration A), I used data objects generated from random bytes. I generated four test data sets with objects of the sizes 10 MB, 1 MB, 100 kB, and 1 kB. Each set contains 10,000 such objects. The measurements with Configuration A represent the median value of 10,000 consecutive operations on one of those sets. The measurements with `ssbench` (Configurations B and C) used the same object sizes, but `ssbench` generates its own set of objects. With `ssbench`, 5,000 repetitions were used to determine the median value.

Where not stated otherwise, I used a Key-Cascade with a height of 3 and a node size of 256 keys. This results in a capacity for 16,777,216 object keys.

5.6.1 Q1: Proxy Impact

Question 1: What is the cost of using the SDOS API-proxy, compared to a direct client-to-server connection?

In order to determine the cost of the SDOS API-proxy, configurations B3 and C were used. This means that `ssbench` was sending requests through SDOS in bypass mode in one instance and directly to swift in another. The results from measurements with three sizes of objects are given in Figure 5.10. Up to 20 concurrent worker processes were used, their number is shown on the x-axis. Each worker sequentially creates requests, issuing one after the previous one is finished. The y-axis gives the total number of completed requests per second, across all processes. The four data series in the Figures' three diagrams show the results for writing and reading objects directly to and from Swift, compared to the same operations using the SDOS proxy in-line.

A first observation is that Swift performs well for read requests, almost saturating the virtual 1Gbit/s network, but comparatively poor for write requests. The reason is that Swift stores and commits three copies of each object before finishing a write request. This is usually done in order to distribute objects for fault tolerance, but was also configured in this single-node setup to emulate a real-world installation. Another observation is that Swift achieves the highest throughput with concurrent requests. This behavior is common in parallel computing. Certain hardware resources are only needed during a certain stage in request processing, so that they are idle during the rest, if requests only occur sequentially. Parallel requests allow utilizing these resources during a larger portion of the time, until they are saturated. Often overall throughput decreases when the concurrency is increased beyond this saturation point, since no further unused resource capacity can be leveraged, but scheduling and queuing efforts increase. This turning point was not clearly reached in the above measurements. This indicateqs that the network bandwidth was saturated first. The fact, that throughput does not quickly decrease after this point, indicates that CPU time and memory were not exhausted, so that enough capacity was present

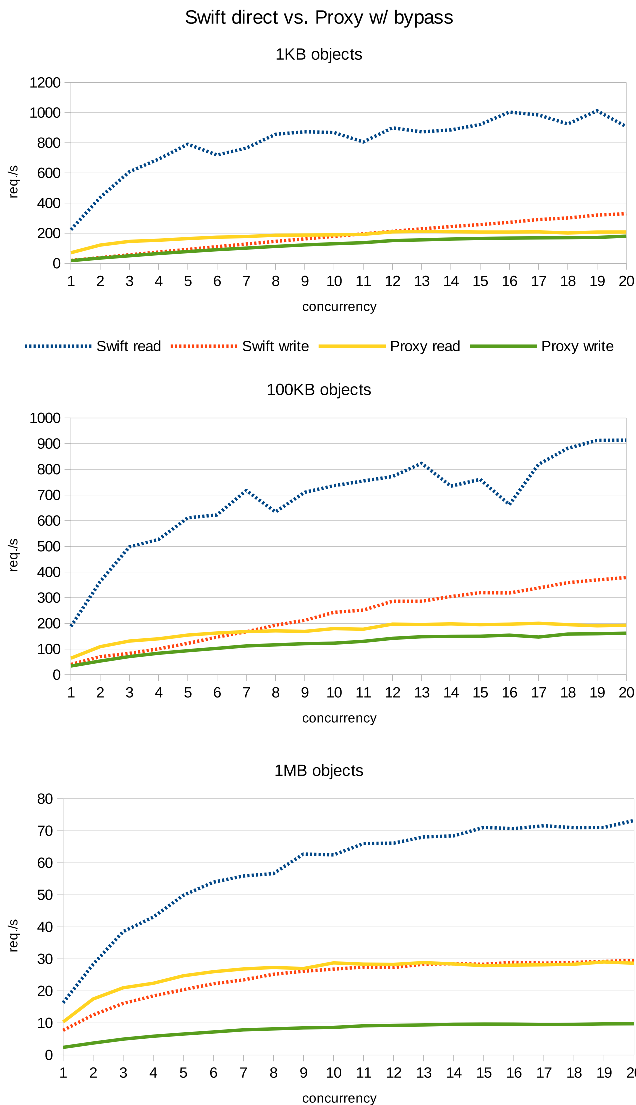

Figure 5.10: Measurement results comparing a direct client-to-server connection (Configuration C) to the API proxy (Configuration B3).

to queue more concurrent requests. In a production deployment, the Swift VM could be given less of those resources.

The next two data series **Proxy read** and **Proxy write** show that a much lower throughput is achieved when using the API proxy from SDOS. In this configuration, no further processing of the requests in done in SDOS. The pure cost of the API-proxy is measured. Higher response times are expected from any proxy and can not be avoided, since an additional component is introduced to the path. However, throughput can still be maximized with a proxy in theory. The reason is that response-time delays only aggregate for sequential requests, but not for parallel ones.

Response time and throughput are both important properties in a practical system. A system might have prohibitively high response times, but still achieve high throughput through massive parallelism. For many applications, high response times are not acceptable however. On the other hand, a system might have very low response times, but not increase its throughput with concurrency. Many applications inherently lead to concurrent requests, prohibiting the use of a data store that does not "scale". For this reason, both have to be considered.

In order to understand the proxy behavior further, Figure 5.11 shows the response time of individual requests. A first observation is that response times increase with concurrency. The **linear increase** shown in the Figure, would be given by a serialized execution that has no parallelism. This is the case since, without parallelism on the server, it can be saturated by a single worker. Adding another worker halves the time the server spends on each, doubling their response times.

These response time results allow the following observations: Firstly, the API proxy increases the response times by a few ms up to 100ms. Secondly, the proxy benefits less from concurrency than Swift itself. With increasing concurrency, the response time delay of the proxy grows quicker then the delay of Swift. The proxy data series shows that the proxy handles requests in parallel, but does so less efficiently than Swift. The SDOS proxy only executes a small part of request processing in parallel.

The reason for the large increase in response time, caused by the proxy, is

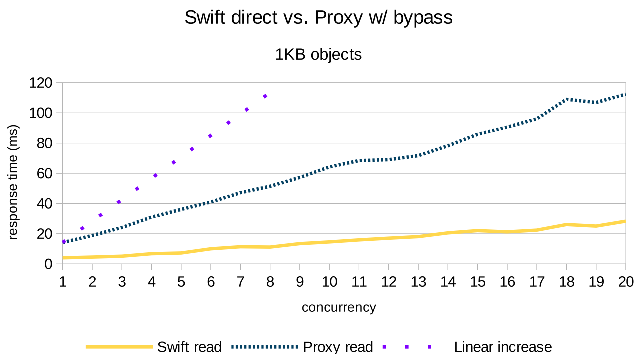

Figure 5.11: Measurement results comparing a direct client-to-server connection (Configuration C) to the API proxy (Configuration B3). Results show the average response-time of the 95th percentile of fastest requests.

the poor performance of the proxy program. The SDOS proxy is implemented in Python and therefore runs on a Python interpreter. This programming environment is not optimized for high performance. An implementation, that uses a language with a more optimized interpreter, or a compiled language, will achieve a lower response time increase.

The low benefit from concurrency, that the proxy shows compared to Swift, is a different issue. I suspect the reason for this in the multi-threaded execution of the API proxy. As discussed in Section 5.4, SDOS is implemented as a single-process multi-threaded application. This means that certain portions of the request handling happen in a single main process before the rest is delegated to the worker threads. Python, as many other languages, allows using an application server when handling HTTP requests. The application server is usually a stand-alone binary program that handles the HTTP protocol and then forwards the request contents to the actual program processing them. In Python, WSGI is the interface between the Python program and the application server. I evaluated different application servers for WSGI and found little difference in their performance. Even a configuration without

application server, where the python program handles HTTP itself, only showed a small increase in response time and no improvement in scaling behavior. This suggests that Python WSGI servers heavily rely on parallel Python processes to achieve high throughput, but make little use of multiple threads. This is expected since the Python interpreter itself is known to benefit little from multithreading.

To conclude, the response times increase by a factor of 2-3 when using the proxy and the throughput is reduced to 30% when writing or 50% when reading objects on average. But the proxy can handle concurrent requests and perform within acceptable parameters. This proxy performance is the baseline for all further benchmarks, since encryption and Key-Cascade run on top of it. However, a better parallel performance of the proxy would be desirable. With this, negative effects of the Key-Cascade locking would be more visible.

5.6.2 Q2: Encryption Impact

Question 2: What is the cost of encryption and Key-Cascade operations, compared to using encryption with a single master-key? In other words, what cost is directly caused by the Key-Cascade key management?

In order to determine the effects of object encryption and Key-Cascade, measurement results from Configurations B1, B2, and B3 (see Figure 5.9) can be compared. For this, `ssbench` was connected to the SDOS proxy and used the pseudo-object API to select the three different modes.

The results from reading and writing differently sized objects are shown in Figure 5.12. The **Proxy bypass** data series is the baseline in this benchmark. The next data series **Encryption only** adds object encryption with a fixed key, comparable to how regular data encryption solutions work. The graph shows that the cost for encryption depends on the data size. With small objects, measured response times are almost the same. With 10MB objects, one third (100ms) of the response time is apparently spent on encryption.

The SDOS configuration with node cache (**Encryption + KC** data series) shows an excellent performance of the Key-Cascade operations. Their impact

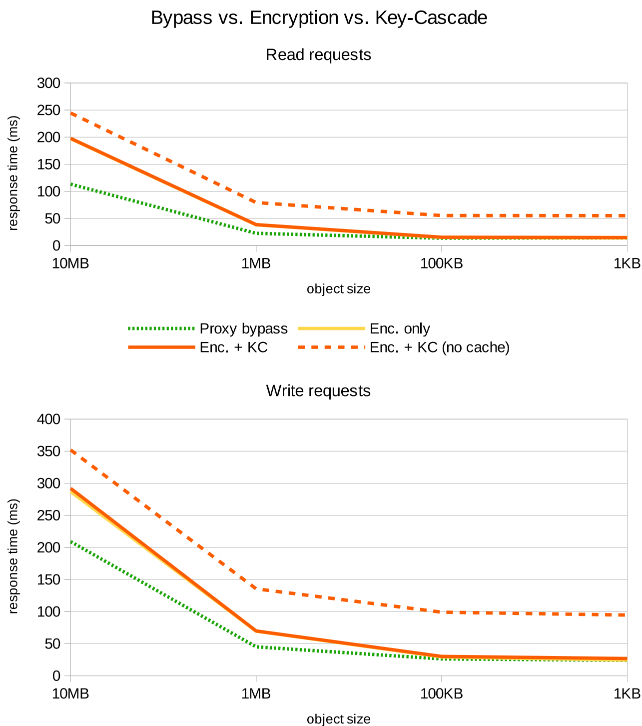

Figure 5.12: Measurement results showing the added cost of encryption and Key-Cascade (Configurations B1, 2, 3) for read and write operations.

is minimal and almost immeasurable. It is also constant and not dependent on data size, since the key management is the same regardless of encrypted data size.

The last configuration, **Encryption + KC (no cache)**, shows how the Key-Cascade performs without the node cache. Responses are 40ms to 80ms slower in this case. This is caused by the additional operations necessary to store and retrieve the nodes from Swift. In this experiment, the Key-Cascade had three levels of nodes. So each object read causes three additional read

requests for the nodes. Object write operations cause up to four read and write operations for the nodes.

No impact of encryption or Key-Cascade operations on the performance of concurrent operations could be observed in my measurements. In other words, the same performance increase through concurrency, that is found in the results to Question 1, is also found when encryption and Key-Cascade are used. The reason for this is that the API proxy already serializes large parts of the request processing. With this, parts of the Key-Cascade operations always run serially, so that locking has no impact on performance. A more optimized proxy implementation should be able to show a performance impact.

5.6.3 Q3: Key-Cascade Geometry

Question 3: What Key-Cascade geometry (node size and tree height) provides the best performance as well as key capacity?

The measurements here show the performance of the secure delete operation. However, the different Key-Cascade geometries have the same effect on read and write operations.

Configurations B1 and B3 from Figure 5.9 were used for the following measurements, so all requests were made through the proxy. The results in Figure 5.13 show the response time for a delete request on a container that has secure deletion. The node cache was enabled for these measurements. When no node cache is used, an additional read as well as write request is needed on each tree level. This adds a constant delay for each level, comparable to the results from Question 2 without cache. Configuration A3 was used to give a baseline reference of Swifts regular delete operation.

The Key-Cascade geometry is determined by the two parameters tree height and node size. They are explained in Section 3.4. They determine the number of object key slots (maximum number of supported object keys). Depending on the intended use, an appropriate size must be chosen, considering the possibility for future resizing discussed in Section 4.1.

Table 5.1 shows the number of object key slots resulting from different

Node size *in Bytes*	$S_n = 32$ *1 KB*	$S_n = 256$ *8 KB*	$S_n = 2,048$ *64 KB*	$S_n = 16,384$ *512 KB*
$h = 1$	32	256	2,048	16,384
$h = 2$	1,024	65,536	4,194,304	$>1 \times 10^{8}$
$h = 3$	32,768	16,777,216	$>1 \times 10^{9}$	$>1 \times 10^{12}$
$h = 4$	1,048,576	$>1 \times 10^{9}$	$>1 \times 10^{13}$	$>1 \times 10^{16}$
$h = 5$	33,554,432	$>1 \times 10^{12}$	$>1 \times 10^{16}$	$>1 \times 10^{21}$
$h = 6$	$>1 \times 10^{9}$	$>1 \times 10^{14}$	$>1 \times 10^{19}$	$>1 \times 10^{25}$
$h = 7$	$>1 \times 10^{10}$	$>1 \times 10^{16}$	$>1 \times 10^{23}$	$>1 \times 10^{29}$
$h = 8$	$>1 \times 10^{12}$	$>1 \times 10^{19}$	$>1 \times 10^{26}$	$>1 \times 10^{33}$

Table 5.1: Number of object keys for different Key-Cascade geometries. Node sizes in Bytes are for AES256 encryption keys.

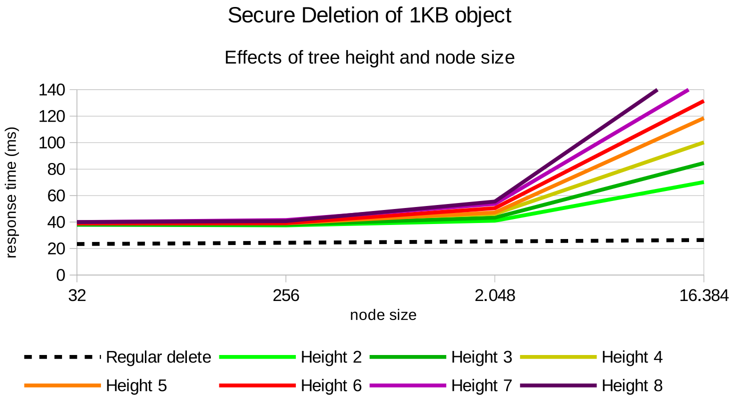

Figure 5.13: Measurement results showing the cost of the secure delete operation. The node cache was used and the regular delete operation is shown for comparison.

tree height and node size combinations. The node size is the number of encryption keys stored in each node. The table also shows how many bytes this amounts to when using AES256 encryption.

According to the results, processing the larger nodes is more expensive and this cost is multiplied by the tree height. However, the height has minimal impact on response times when the nodes are small. So a tree with small nodes and large height should be chosen.

A good overall node size is $S_n = 256$ according to the results and any height can be chosen. According to Table 5.1, a Key-Cascade with height $h > 4$ can hold above 1×10^{12} object keys at this node size.

5.6.4 Conclusion and Areas of Improvement

The prototype implementation as an API proxy adds a constant and measurable delay to the response times, but they remain in acceptable ranges for benchmarking purposes. The key management for secure deletion adds almost no delay to the read and write operations in an encrypted container. This suggest that such a key management could be used in existing encrypted storage systems. The secure delete operation in encrypted containers is 20ms slower than the regular delete operation in my experiments (for the above recommended Key-Cascade geometry).

Improvements to the proxy could be made by implementing it in a more optimized language. Using a specialized application proxy like Envoy[1], instead of implementing a client and server interface in Python, will also lower the response time. In order to improve concurrency further, the protocol and encryption handling should also be moved to a separate program, that can easily run in parallel. Only Key-Cascade operations and the deletable key source would remain in the current SDOS program. Further improvements are parallel and distributed execution, discussed in Section 5.4.

Finally, it is also possible to move the encryption and key management components into the client or server program. This eliminates the issues of an API-proxy, but has other drawbacks. The benefits of the API-proxy

[1] Envoy service proxy: https://lyft.github.io/envoy/

design are stated at the beginning of Chapter 5. A client-side implementation is non-trivial for many reasons, if more than one client should access the system at any given time. Distributed locking and key management are only a solution when mutual trust is given. In a situation with untrusted parties, a key distribution scheme with minimal trust could be used.

A server side implementation would likely be the most practical approach. Most encrypted storage systems today encrypt server side already. Integrating a Key-Cascade implementation with an existing encryption module, achieves this. An SDOS module, as suggested above, would be a possible solution for the server side as well. The above discussion of parallel processing then also applies here.

5.7 Related Publications

The contributions in this Chapter are supported by the following of my publications:

The initial idea for a tree-based encryption key management was published as a US patent application in 2015. This application was granted US patent 9,298,951 in 2016 [BLM+15] and was awarded “Invention of the Quarter” at IBM software group in the first quarter of 2015.

In 2017, a poster presentation about the Key-Cascade concept and the prototype of SDOS as well as the larger MCM system was accepted at BTW [Wai17]. In the same year, another poster presentation and demo that showed the integration with Trusted Platform Modules as deletable key source was accepted at EDBT [WWM17].

Chapter 6

MCM: a Demonstrator Application for SDOS

MCM (the Micro Content Management System) is a demonstrator application for SDOS. MCM contains a Web-based user interface and services for extracting and visualizing object metadata. The user interface supports uploading, downloading, and listing of objects and also allows configuring the Key-Cascade parameters for SDOS. It contains an interactive visualization of the Key-Cascade data structures and allows the user to interact with the hardware key store, if used.

I included the extraction and visualization of object metadata in order to show how cloud providers can still offer advanced services, even when customers encrypt their data. In MCM, the user can decide which metadata they want to extract and store unencrypted, so that further processing on this data by the cloud provider is possible.

MCM is designed as a multi-component cloud application that can be deployed in multiple topologies. As part of MCM, I provide a fully automated deployment of all the components, including SDOS and the Swift Object

Store.

In the following sections, I first discuss the design goals and required functionality for MCM. Then I show the system architecture and explain the role of each of the components. This is followed by a presentation of the possible use cases of the system. Finally, I present an executable model that uses Docker containers. Using this model, interested readers can quickly test and evaluate SDOS and MCM with little effort, since the deployment is fully automated and contains all needed components.

6.1 Design Goals

MCM is designed as a demonstrator system that shows how cryptographic deletion with SDOS can be integrated into a cloud storage service. Part of the motivation for this work in cryptographic deletion is to support outsourcing enterprise content archives. Such archives, containing, for example, invoices and contracts, are typically driven by Enterprise Content Management Systems (ECM).

Enterprise Content Management (ECM) systems were created to help companies deal with the large volume of documents and data they produce and consume. They may handle invoices and contracts, project documentations, scanned letters and documents, e-mails, images, and others. ECM systems help companies to store and archive this data, to categorize and retrieve it, as well as to keep it confidential. Security measures are especially relevant for ECM systems due to the sensitive nature of the data they manage.

ECM systems typically have a complex architecture that consists of many legacy applications. These applications are deployed and operated in on-premise data centers and are often integrated with other enterprise information systems as a data source or sink.

For these reasons, I have the following design goals for MCM:

- Provide a cloud-native implementation of basic ECM functionality. This creates a framework to evaluate concepts, architecture patterns, and implementations of individual cloud-based ECM components.

- Integrate SDOS to provide encryption and cryptographic deletion of stored content.
- Provide a user interface that allows configuring and using the SDOS features.

In the following, I discuss the tasks and functionalities of ECM systems that are implemented in MCM.

The tasks and functionalities of ECM systems revolve around the "document lifecycle" that describes document creation, distribution, use, maintenance, and disposal (or deletion). Documents (i.e., objects) are created by ingesting them from a data source. Sources are typically bulk ingested from a file system, other systems pushing data via an API, or direct upload by a user via a front-end [GHH+12].

Objects in an ECM system consist of two parts: The actual binary data object and attached metadata. Extensive support for metadata is one of the key features that separates ECM systems from file systems and the likes. Typically, ECM systems employ three separate storage back-ends for these two types of data; a large file system for the binary data and a relational database, as well as a full text search index for metadata [Rei02].

Metadata forms the basis for advanced functionality like document classification, targeted retrieval, and analysis. This is often achieved by integrating enterprise reporting and analytics software.

A key aspect of maintaining documents is retention management. Documents in an ECM system may have a defined lifetime. Enterprises must be compliant with rules and regulations about this lifetime for certain types of documents. Contracts and legal documents must be kept for a certain period before they may be deleted. Enterprises must be able to produce these documents in front of a court or get fined. ECM systems support this requirement with retention-date support. The system will prevent users from deleting documents while they are in retention. Retention management and the document lifecycle end in the disposal of the document. Enterprises have a desire to dispose documents as soon as possible, either because they are required to by law, or because they want to avoid keeping potentially

compromising information. For these reasons, ECM systems have the functionality to automatically delete documents that are past their retention date.

6.2 System Architecture and Components

In order to effectively utilize cloud technologies and enable flexible deployment, I built the MCM system as a collection of small services that have a defined functionality and are loosely coupled by standard APIs. A component view of this system is given in Figure 6.1. At the bottom, Figure 6.1 shows the three data management systems and the three types of APIs they offer. These APIs (Swift-REST, SQL, Apache Kafka) are also the integration method for the custom components. These components either communicate via one of the data management systems, or by acting as a transparent API proxy for the Swift-REST protocol. This Swift Object Store from the OpenStack project[1] is the central data storage location. It holds all the managed data objects as well as their associated metadata. I use a relational database as a replicated metadata warehouse, as Swift lacks advanced querying capabilities for metadata (besides retrieving and listing). Finally, an Apache Kafka[2] messaging service enables communication between the components and is used to asynchronously execute management tasks.

On the top layer, Figure 6.1 shows the user interface or application interface to MCM. Bluebox is the custom graphical user interface for MCM, but any generic Swift client can be used.

There are two other custom components in MCM besides the Bluebox user interface: SDOS and the task runner. SDOS provides the functionality for data encryption and cryptographic deletion over the standard Swift protocol. The task runner asynchronously executes management tasks.

[1] http://docs.openstack.org/developer/swift/
[2] http://kafka.apache.org/

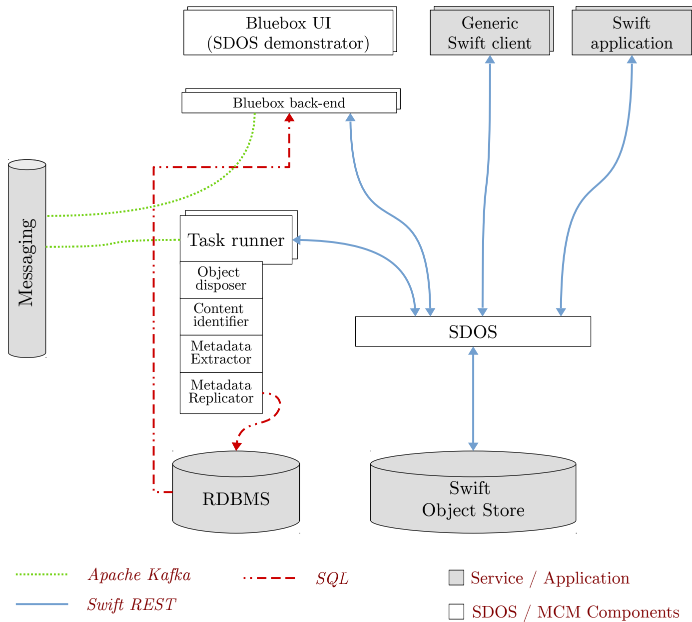

Figure 6.1: A high-level overview of the MCM components. The three custom components are Bluebox, SDOS, and the task runner.

6.2.1 SDOS

The SDOS core component is explained in detail in Chapter 5. In MCM, it provides capabilities for content encryption and decryption and also cryptographic deletion. SDOS is implemented as an API proxy for Swift so any object store access can happen through SDOS and selectively use its features.

6.2.2 Bluebox UI

Bluebox is the custom user interface for SDOS and the larger MCM system. It supports the standard Swift protocol but allows interaction with the SDOS custom metadata fields and pseudo-object interface.

Bluebox is separated in three functional components and two layers, as indicated in Figure 6.2. The layer on top consists of the client-side UI

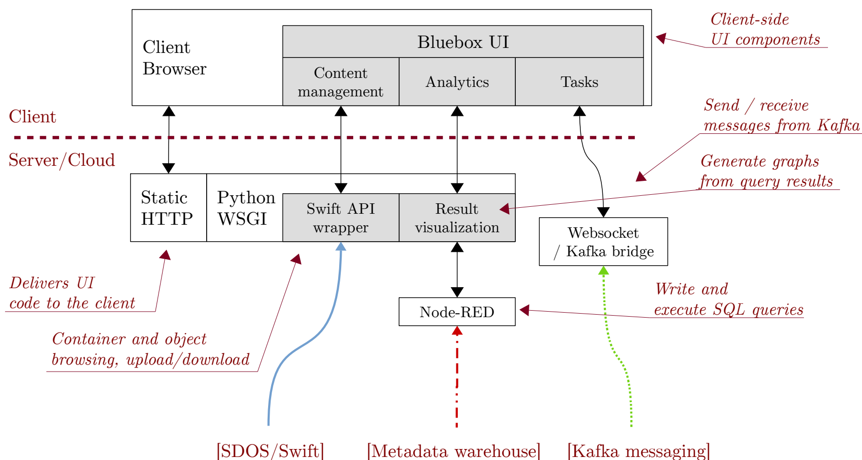

Figure 6.2: Internal components of the Bluebox user interface.

components that run as an AngularJS[1] application inside the client's browser. The bottom layer represents the server side. It consists of a static HTTP server which delivers the client application to the browser and a Python runtime that provides a back-end for this application.

This back-end consists of an API wrapper that interacts with Swift in order to provide this data via a REST/JSON API to the client-side application. It also contains a graph rendering component that is used in visualizing metadata. A separate service provides an interface between the front-end and the Apache Kafka messaging system. This allows users to run asynchronous management tasks.

This back-end holds no application or session state so that it can scale horizontally. The HTTP server can also scale easily since it only serves static content to the client.

Horizontally, the user interface is separated into three functional components:

Content Management: Here the user can interact with the Swift Object

[1] https://angularjs.org/

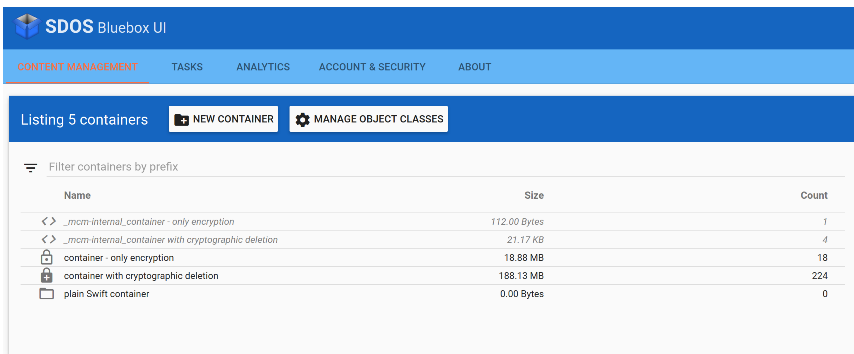

Figure 6.3: Screenshot of the container view in Bluebox.

Store in order to load and retrieve data. Bluebox provides a file-browser-like interface to the object store with capabilities to manage metadata, objects, containers, and the SDOS functions. Figure 6.3 shows a screenshot of the container listing in Bluebox.

Analytics: This component allows users to work with object metadata. It fulfills three functions: First, it shows the user the available data and schemas in the metadata warehouse. Second, it allows accessing a Node-RED[1] instance which has an SQL connection to the metadata warehouse. The user can create SQL queries and graphically link them together with Javascript functions inside Node-RED. Third, it provides endpoints for Node-RED to output the analysis results. This data is then forwarded to the client application by the API wrapper on the server side. The application locally renders the data into graphs in the client's Web browser.

Tasks: This component provides a user interface to the Apache Kafka messaging service. It allows users to trigger tasks and also to receive responses from the task runners. These tasks are used for executing content identification, metadata extraction, metadata replication, and

[1] http://nodered.org/

content disposal.

6.2.3 Task Runner

The task runner is a component that asynchronously executes tasks based on task descriptions received over Apache Kafka messages. These tasks execute long-running management operations like content identification and are triggered either by users or programs. In this application, a message-driven service has useful benefits [FLR+14]:

Asynchronous execution: Tasks may run for hours or days and the caller may want to receive status updates. Message-based communication is well suited for such an application.

Horizontal scaling: Each message contains a description for a single task. Horizontal scaling can be achieved just by adding multiple task runner instances. Kafka can be configured to deliver each message to exactly one (currently available) task runner instance.

Distributed environments: Task runners can execute different types of management tasks and some of these tasks involve more sensitive data than others. With the message-driven service architecture, it is possible to run multiple instances of the task runner in different trust zones. For example, one instance may run in a local trusted environment and another on cloud resources. Kafka can now be configured to distribute messages with different tasks to the different instances.

Monitoring and auditing: Task description and execution status messages can easily be exploited for creating audit logs and monitoring the execution of tasks. Especially in a cloud environment, many existing analytics services could be quickly connected to this messaging system.

The task runner in MCM uses a plug-in architecture to support the execution of different types of tasks. The task runner itself has an interface to the Apache Kafka messaging system and to the Swift object store (or SDOS). It receives and interprets messages and starts executing the tasks

in new threads. After starting task execution, the task runner also provides an iterator interface to the objects on the object store. This is used by some task implementations that only execute simple operations on each of the objects in a container. The task runner itself does not have access credentials to Swift, so each task-requesting message must contain a temporary access token provided by a user or application.

The four task plug-ins that are provided in MCM are the following:

Object disposer: In MCM, each stored object can have a metadata field "`retention-date`". Its value is a date that specifies until when this object must be kept. If an object has a retention date in the past, it can be deleted. The object disposer fulfills this function. It retrieves the metadata from each object in a container and compares the value of the retention date field to the current date. Expired objects are then deleted. This task is well suited when cryptographic deletion and the batch-delete log are used: The disposal task will fill the batch-delete log so that it can be processed afterwards.

Content identifier: The content identifier inspects the actual binary content of an object and determines its type in order to populate the metadata field "`content-type`". In Swift, it is the task of the uploading client to set the content type. Most clients, if at all, use the file name extension when uploading objects from a file system in order to determine the content type (e.g. a file ending in ".jpg" is considered a JPEG regardless of the actual content). This often leads to incorrect or unset `content-type` fields. For this reason, MCM contains this content type identifier that looks into the binary content of the object in order to find known sequences of bytes that identify a certain content type. I use the programming library `libmagic` from the well known Unix `file`[1] utility. It contains a comprehensive list of known content types and their binary fingerprint.

Metadata extractor: Once the content type for an object is known, MCM

[1] https://github.com/file/file

can use this knowledge about the objects' structure to automatically extract known metadata fields from it. For this, I include a metadata extractor that provides an execution environment and plug-in interface for content type filters. The filter plug-ins must specify which content types they support and what metadata fields they can produce. The extractor will determine the appropriate plug-in for each object based on the content type field, download the binary content of the object, and pass this data object to the filter plug-in. This plug-in returns the found data as key/value pairs that the extractor writes back to the object store.

In MCM, I include filter plug-ins for common image formats (JPEG, PNG, BMP, ...), PDF documents, and e-mails. For example, the JPEG filter may produce metadata fields "`resolution`" or "`compression`" while the PDF filter may produce "`author`" or "`title`".

Metadata replicator: The Swift Object Store only supports storing and retrieving metadata based on entity names; no advanced querying of the metadata itself is possible. In order to use metadata for analyzing or finding objects, MCM needs a data management system that provides advanced querying capabilities. For this reason, I include a relational database that is used as a metadata warehouse. The metadata replicator is responsible for retrieving the metadata from the object store and inserting it into the database. I only replicate the metadata unidirectional to a relational database to use it for advanced searching, querying, and analysis. This database is used as a warehouse only for reading data. All modifications of metadata happen directly on the object store.

The replicator handles metadata from two different sources: i) Object store "internal metadata", ii) metadata extracted by content type filters. The replicator maintains one database table for each type of metadata. They reference each other by foreign-key relationships on the object and container name.

i) Internal metadata is created by Swift itself and is always present

on all objects. It includes, for example, the object size, time and date of last access. The replicator maintains one database table with the appropriate columns for this type of metadata.

ii) Filter metadata: The replicator uses the same plug-in interface as the metadata extractor in order to read the list of metadata fields from the plug-ins. This data is used to create a table for each type of filter and to select the appropriate metadata fields from the object itself. Note that the extractor and replicator must have the same plug-ins in the same versions installed in order to replicate all the correct fields.

6.3 Use Cases and Functionality

MCM was designed as a demonstrator system for SDOS with the "enterprise document archive" scenario (see Section 6.1) in mind. In the following, I explain three examples of use cases that are possible in MCM: Content management, metadata analytics, and content disposal. These examples are included here to show how the MCM components interact and what functionality is possible with the system.

6.3.1 Container Creation, SDOS Configuration, and Content Management

This use case covers the basic functionality of content management systems: storing, organizing, and retrieving content. It also covers the integration with SDOS for configuring and using content encryption and cryptographic deletion.

The user starts this interaction by logging in to Bluebox with their Swift storage account and switching to the "Content Management" view that is shown in Figure 6.3. This view presents a list of the existing Swift containers.

The user creates a new container from this view. Bluebox then presents the dialog shown in Figure 6.4 where they can choose to create one of three container types: i) An unencrypted "plain" Swift container where no SDOS functionality will be used. ii) An encrypted container without cryptographic deletion. The user now has additional options to specify the key source type

Create Container

Container name

container with cryptographic deletion

CREATE

CREATE & CLOSE WINDOW

Enable content encryption

Enable SDOS key management for cryptographic deletion

Tree properties

Height: 2

Node size: 4
=> 2 Bits

The *Key Cascade* tree structure enables efficient cryptographic deletion. It manages all the object keys in its leaves. The height and node degree determine how many keys are available as well as the performance.

<- This example tree has height **2** and **2** node bits.

New container will have height **3** and **8** node bits. The nodes will have **256** slots (2^8).

The Key-Cascade will support up to **16,777,216** object keys.

Tree height 3

Node bits 8

Deletable key

Root key

Deletable key

First tree level

The root key for each cascade is stored encrypted with a *deletable* key.

Please select where the deletable key should come from. Note that the secure-delete property is derived from this deletable key.

'0000'

same key

'0000'

Static test key: Uses the same static key every time. Only useful for testing and debugging.

'passw0rd'

Secure forget

'new pass'

User password: Uses a password to derive a deletable key. Secure deletion is provided by setting a new password and "forgetting" the old one.
You must provide the current password before using the container, and provide a new password before starting secure deletion.

TPM

Secure replace

Trusted Platform Module (TPM): Uses TPM hardware to securely en/decrypt the root key. The deletable key is stored inside the TPM, and can never leave it. Secure deletion is provided by the TPM which can securely replace internal keys.

Batch deletion

true

Names of deleted objects get saved to a log so that secure deletion (re-keying) can be done later for a batch of objects.

Object contents are deleted from the object store immediately but the keys remain, and the names remain allocated, until batch deletion is triggered.

Figure 6.4: Screenshot of the container creation dialog in Bluebox showing the options for SDOS.

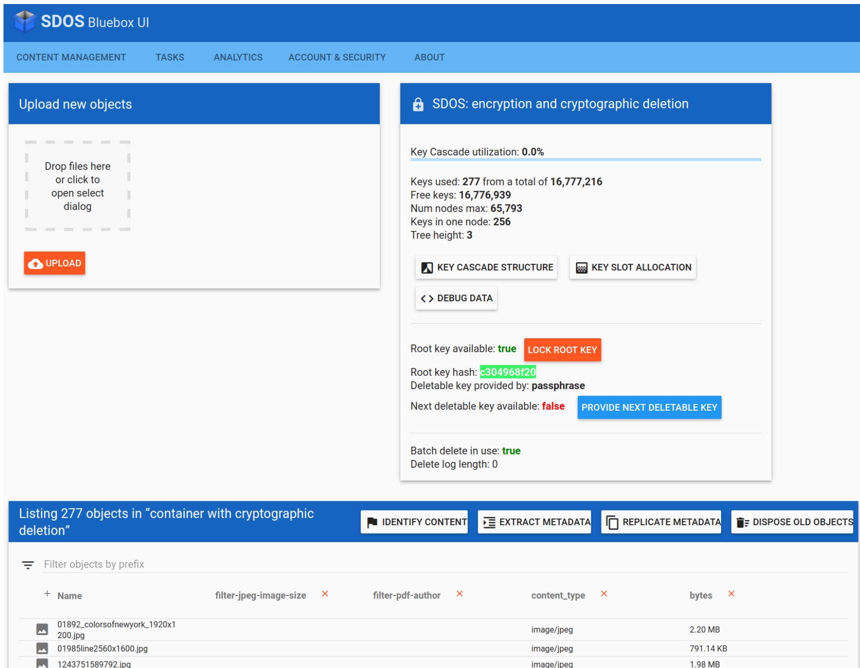

Figure 6.5: Screenshot of the object view in Bluebox. Information retrieved from the pseudo-object API is shown on the right.

and configuration (explained in Section 4.3). iii) An encrypted container with cryptographic deletion. Besides the key source, the user now also has options to specify the Key-Cascade geometry (node size and height) and to enable the batch-delete log.

After creating a container with cryptographic deletion and the password key source, the user enters the container. With this type of key source, they first have to provide a password which SDOS uses as the first "deletable key" for the new Key-Cascade. They enter this password using the SDOS functions shown on the right in Figure 6.5. The information displayed here is retrieved from the pseudo-object API in SDOS. It shows the geometry of the Key-Cascade, the number of available object keys, the state of the key source,

as well as the state of the batch-delete log. If the key source is operational and unlocked, a color-coded hash of the Key-Cascades root key is shown here as well.

After initializing the container with a password, the user uploads files using the drag-and-drop feature. The files are now stored as objects and organized inside a container. The user can use Bluebox to upload further files, download or preview their content and metadata, or delete objects. They can also use the demonstrator features for SDOS and inspect a visualization of the actual Key-Cascade used for this container. Figure 6.6 shows an example with a single object highlighted. An animation is available here as well, it shows the user how re-keying affects the Key-Cascade if the selected object gets deleted.

6.3.2 Metadata Management and Analytics

Metadata is a key part of Enterprise Content Management as it allows classifying, finding, organizing and analyzing objects. This use case also highlights an important aspect of an encrypted cloud storage systems: If client-side encryption and client-side key authority are used (as in this thesis), then the cloud provider can offer no advanced services on the data. In MCM, only the object content is encrypted while the object metadata is stored in plain. This metadata contains the object names so that storing and retrieving objects with their actual names[1] is possible. In MCM, the user can selectively extract further metadata from the objects and store it unencrypted as well. With this approach, the user can selectively give the cloud provider access to some data in order to have more functionality in the product.

One possible application could be to extract sender and receiver addresses from an encrypted e-mail archive, so that the provider can offer server-side search functionality. Another application for a personal cloud backup could be to extract date and time from camera images. The provider can then offer a time frame query interface so that users can quickly find old pictures. In the following example, I explain the workflow for extracting the x and y

[1]Other approaches that also encrypt the object names are discussed in Section 2.4.

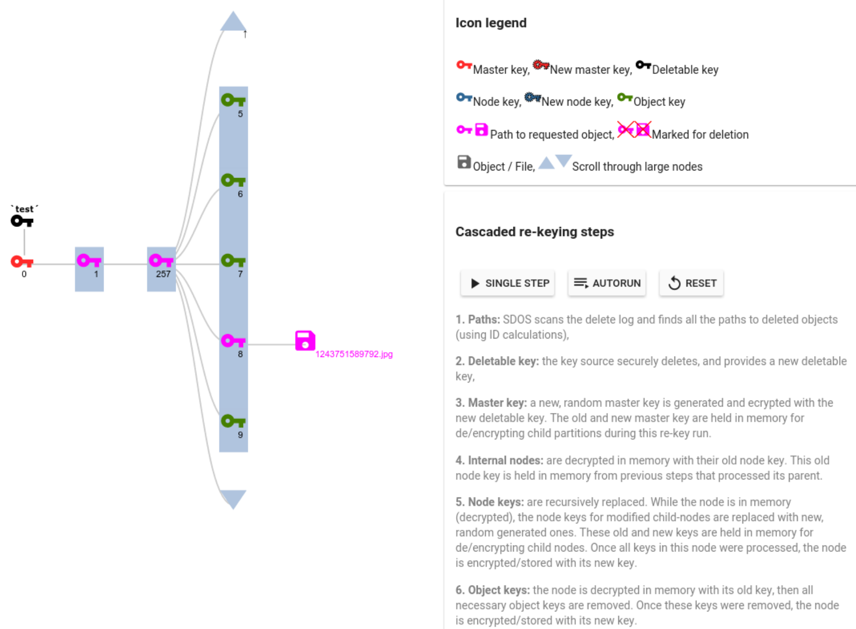

Figure 6.6: Screenshot of the Key-Cascade visualization in Bluebox. An animation shows the steps involved in cascaded re-keying.

dimensions from JPEG images in order to calculate statistics at the server side.

The extraction process is shown in Figure 6.7. It is performed by the task runner discussed in Section 6.2.3. The process starts on the top left with the JPEG object that does not have JPEG related metadata yet. Figure 6.5 shows such objects, note that the table column for “image-size” has no values. The objects in that table however already have the correct content type “image/jpeg” in their metadata; Step (1) in Figure 6.7 could be skipped in that case.

Step (1) of the process is identifying the content type. The content type

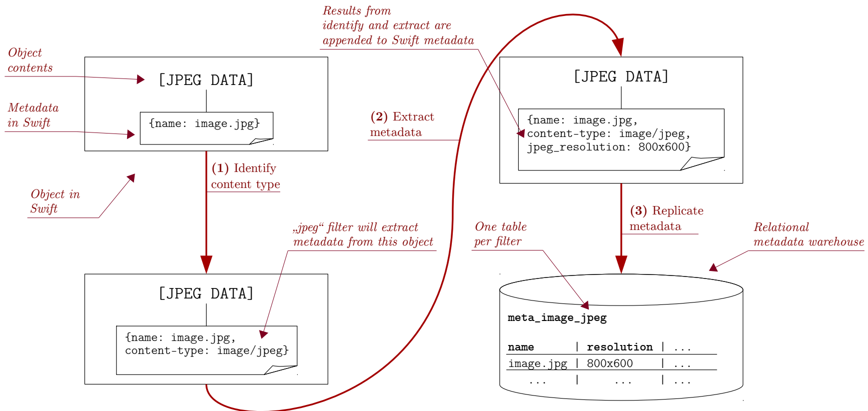

Figure 6.7: The process of extracting and replicating metadata from objects.

identifier will read the object, determine the type, and then write the value back into Swift as metadata for this object. Once the correct content type is set, the metadata extractor can process the object in Step (2). The content type is needed in this step because the extractor uses a set of plug-ins to extract information from different types of objects. The content type decides which "filter" plug-in to use. Just like the identifier, the extractor writes the results back into Swifts object metadata. In Step (3), this metadata is then replicated to the relational warehouse. The content type of the objects is used here again because the warehouse contains one table for each content type.

The replicator creates the following tables for metadata: One table with the internal metadata from the containers (container name, size, number of objects, SDOS configuration, ...). Another table with the internal metadata from the objects (container name, object name, size, content type, ...). The schemas for those two tables are static, since they only account for the known internal metadata fields. Then the replicator creates one table for each filter plug-in in the extractor. The interface definitions in those plug-ins specify which content types have which metadata fields, allowing the replicator to derive a table schema. When replicating object metadata, the replicator

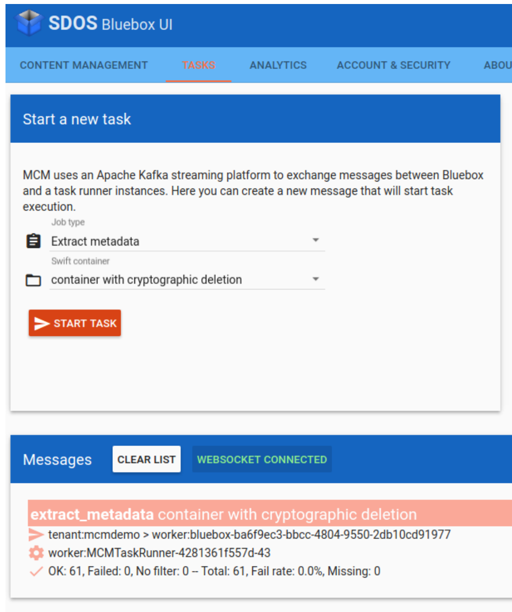

Figure 6.8: Screenshot of the task execution view in Bluebox. The request message and two replies from a task runner are shown.

first populates the container metadata for the affected container. It then populates the object metadata table; the object name and container name columns act as the primary key for the content-type-specific tables. This table gets populated next with the specific metadata fields; in this example "jpeg_resolution".

The metadata can now be analyzed in the warehouse, or the extraction process can be repeated in order to update the warehouse contents.

After logging in, the user can start the process either directly from the object view in Bluebox, or from the tasks view. The object view is shown in Figure 6.5; note the buttons in the lower blue bar. In the tasks view, as shown in Figure 6.8, they select the type of task (Steps 1-3 from above) and the container to process. Since the objects already have the correct content type, they start with Step (2), metadata extraction.

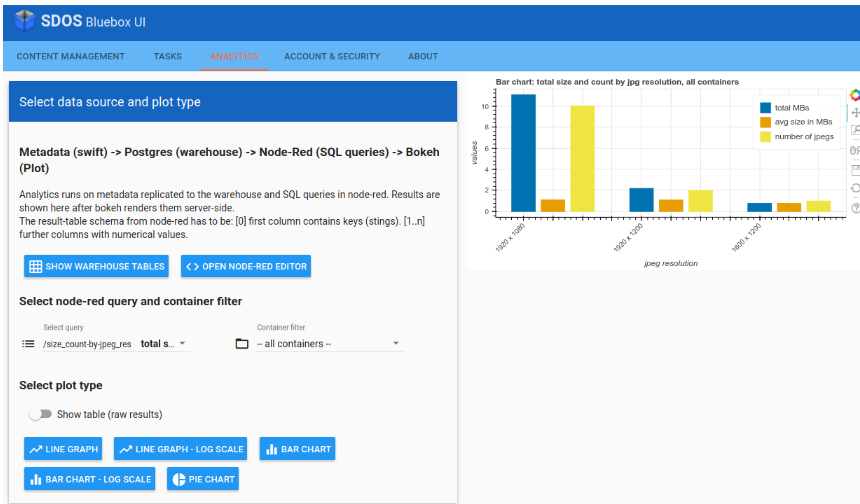

Figure 6.9: Screenshot of the analytics view in Bluebox. Template charts are filled with query result data.

In MCM, the tasks always run on all objects inside one container. When the user selects to start the task, Bluebox sends a message with a task description over the Apache Kafka messaging system. Bluebox includes the users current authentication token in this message; this is used by the task runner to access the object store. It also includes a random task ID used for correlating messages.

The tasks view in Bluebox shows all the messages in Kafka, grouped together by the task IDs. In Figure 6.8, the user can see the outgoing task request and two responses. The first response is sent by the task runner to indicate that execution is started. The second response is sent once the task is finished. It contains a small success report that shows that 61 objects were successfully processed. They can now continue with Step (3), metadata replication. They select the different task type from the drop-down menu and select “start task” once again. After receiving the success report for this task, they switch to the analytics view.

The analytics view is shown in Figure 6.9. Here, they select a predefined

query that calculates the count and size of JPEG objects, grouped by their resolution. The user can additionally limit the query to a single container. Finally, they chose one of the provided chart types and receive the visualized query results.

MCM contains a few such example queries, but users can write their own queries using the included Node-RED[1] editor.

6.3.3 Content Disposal and Batch Deletion

In this use case, I detail how retention dates, object disposal, batch deletion, and key sources interact. A practical application could be a document archive where expired documents get securely deleted periodically.

The user starts with the container they created in Section 6.3.1. It uses cryptographic deletion, a password as the deletable key source, and the batch-delete log. Firstly, they start by uploading new objects over a certain period. The user specifies a retention for these objects. This is either possible with the Bluebox UI at upload or a later time, or over the Swift API.

After some time, the user wants to dispose the expired objects, i.e., objects where the retention date lies in the past. The task runner has a disposal task for this purpose: It checks the retention date on each object by retrieving their metadata, and compares it to the current date. If the retention date lies is the past, it issues a delete command for this object over the Swift API.

They start the disposal task from the tasks view in Bluebox, and await the success report message detailing how many objects were deleted. Since the container uses cryptographic deletion, each object delete would trigger re-keying. But since the user configured the password key source, they would have to provide a new password for each deletion. For this reason, they also configured the batch-delete log for this container. In this case, object deletions are only performed in Swift but no immediate re-keying is triggered. Instead, the names of the deleted objects are appended to this batch-delete log.

[1] http://nodered.org/

After the disposal task has populated the batch-delete log, they can trigger re-keying to process the log at a later time. At this point, the user has to provide a new password so that a new deletable key can be derived. The cascaded re-keying method then processes all object deletions in a single run and only requires a new deletable key once.

The password key source and this above scenario highlight the usefulness of the batch-delete log. The password key source requires user interaction when the deletable key should be replaced. This makes it difficult to use when re-keying is performed often, or immediately on object deletion. But practical key sources in an enterprise setting may have the same problem. A deletable key for an enterprise archive, stored in a hardware security module, likely requires human interaction or special approval before secure replacement. This key replacement might be integrated into a business process in order to allow reviewing, accountability, and auditing. The password key source in MCM mimics this behavior.

The batch-delete log allows decoupling object deletion and cryptographic deletion and supports the use case of periodically deleting old data.

6.4 Authentication and Authorization

Because the MCM system consists of multiple services that are used by each other as well as external users, it needs a mechanism for authorizing requests. In MCM, this is realized with two approaches: Swift authentication and internal service accounts.

Swift authentication is an extensible authentication and authorization provider that can authenticate users and authorize their requests for operations on the object store. It can use an internal (i.e., as a configuration file) user database for development and testing but also supports a Keystone[1] back-end for use in production. Before any request to Swift can be made, a user has to authenticate. If this is successful, Swift issues an authentication token (i.e., Auth-Token) that is valid for a certain period. This token is then

[1] Keystone is the identity service from the OpenStack project: http://docs.openstack.org/developer/keystone/

used for authorizing the actual request. None of MCM's internal components has its own Swift account. SDOS is implemented as a Swift API proxy so it can use the user's Auth-Token to make requests on the user's behalf. The three metadata generation components as well as the retention manager are able to operate without direct user interaction by receiving tasks from the message queue. The solution for this is that users include a temporary Auth-Token in the task description when they create a task.

This design allows me to anchor all operations on the object store directly on a user's request. Without a user that has a valid account, none of the internal components can access the data inside the object store. Using Auth-Tokens inside the task description instead of user name and password has two advantages: i) Tokens can be easily revoked in case of security breaches. ii) Leakage of tokens is less critical since they're only valid for a small period.

Since the user's accounts are valid Swift accounts for the cloud object store, I need a mechanism that prevents circumventing our internal API proxy components by directly connecting to the underlying Swift Object Store. I solve this by using a Swift service account[1]. In this configuration, Swift requires a combination of two sets of credentials for authentication: A user's credentials as well as a services credentials. This way, users are not able to use Swift without going through the internal services that have the service credentials.

Internal service accounts are used by the internal services for authenticating against the metadata warehouse and the messaging system. During deployment of the MCM system, I create an account for each required access according to Figure 6.1. This means that the internal services have direct access to the metadata warehouse and messaging system without a user's request, contrary to the object store, which can only be accessed with the help of a user's request (via their Auth-Token).

These internal services also need to authorize requests from users for accessing data on the message queues and metadata warehouse. Here, they can not directly forward requests, as it is possible with the object store,

[1] http://docs.openstack.org/developer/swift/overview_auth.html#openstack-service-using-composite-tokens

since the users do not have the accounts for these two systems; but the services have. I solve this by anchoring all operations on these two systems to user operations on the object store. With this method, I can link a users permissions on swift entities to permissions on messages and tables in the metadata warehouse. For example, a user is logged in to the Bluebox UI with their Swift account that only enables access to Container A in Swift. The user requests to access metadata belonging to Container B from the metadata warehouse. In order to authorize this request, the UI will forward their Auth-Token to Swift and verify if the user has permissions to access Container B. In this example, Swift denies the authorization, so the Bluebox component will deny access to the metadata as well.

I implicitly link the user's permissions on the Swift Object Store to permissions on the message queues and metadata warehouse with this mechanism. This has the advantage that I do not need to configure the permissions in separate systems and keep them consistent.

6.5 Cloud Modeling and Deployment Automation

Cloud modeling refers to the practice of describing the high-level architecture of a cloud application in a machine-readable way. Today, the goal of this is to provide deployment automation for an entire multi-component application. However, research efforts like TOSCA aim to provide further benefits from such cloud application models like topology discovery, dynamic adaptation and migration, or declarative modeling [BBKL14].

In the following, I discuss the application topology of MCM with regard to its deployment on public or private cloud environments. I present three alternatives to how MCM's components can be deployed and discuss their implications on usability and security. Following this, I present an executable application model of MCM, written for the Docker Compose[1] automation engine, with which the MCM system can be deployed with little effort.

In Section 6.2, the high-level architecture of MCM was introduced. Fig-

[1] https://docs.docker.com/compose/

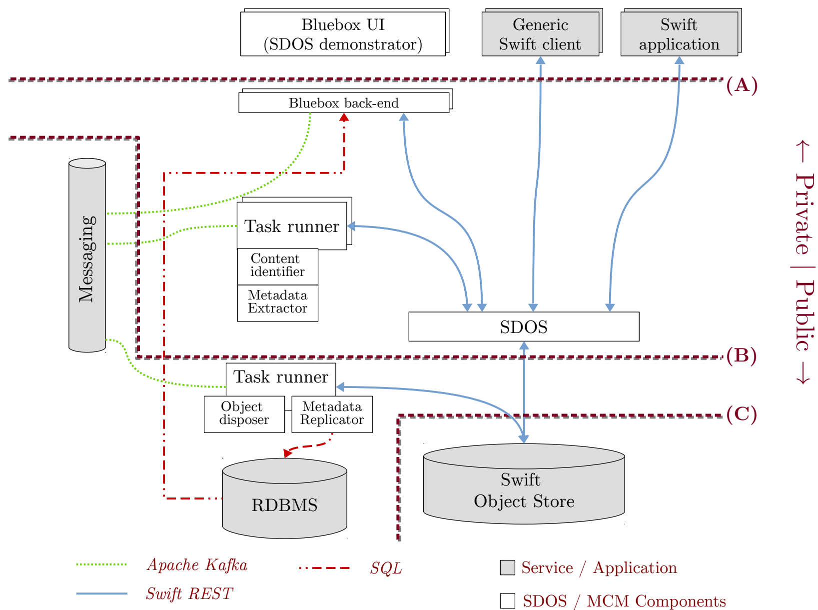

Figure 6.10: A high-level overview of the MCM components. The three red lines show possible deployment configurations.

ure 6.10 shows this architecture overlaid with three possible "border locations" that separate a public and private cloud environment, indicated as (A), (B) and (C). They differ in which components are deployed on-premise (i.e., in a private cloud) and in a public cloud. They have different security implications, but also enable different usage scenarios.

In deployment scenario (A), all system components run in a public cloud. Only client applications or Web-based user interfaces run on the users own systems. This scenario would be used by a cloud provider who wants to offer MCM as Software-as-a-Service (SaaS). Data security depends on how well the provider secures the encryption keys and protects against outside attacks. On the other hand, this scenario enables using the system from anywhere, on any device.

Scenario (B) operates the encryption and all components handling unen-

crypted data, on the private cloud. Only metadata and encrypted data are stored on the public cloud. An instance of the task runner is located at the public side. The tasks that this instance handles only involve metadata. It does not have access to SDOS, but only to Swift. Meaning that this instance of the task runner can not read the object contents. In this deployment scenario, the object contents are inherently secure against outside attacks and cloud provider breaches.

Scenario (C) also secures the metadata by moving it into the private cloud. Only encrypted data is stored on the public cloud in order to be secure against any attacks on this public infrastructure. In Scenarios (B) and (C), it is no longer possible to use the system from anywhere. Only users with access to the local private cloud have access to the system. These scenarios also impose higher operational cost for the customer, since they have to operate the private cloud instead of relying fully on public cloud offerings.

6.5.1 Trusted Enclave

Scenario (A) provides the lowest cost and highest functionality for customers, making such SaaS offerings very popular. The SaaS providers therefore often extend their cloud architecture with a "trusted enclave", in an attempt to increase security. An example of this approach is given by Microsofts Cipherbase, as discussed in Section 2.4.3.

This trusted enclave is a specialized hardware component[1] which securely stores encryption keys and carries out cryptographic operations. It is built into some of the cloud servers used to host the application. This is an effort to minimize the risk of leaking encryption keys or sensitive data and make attacks more difficult.

In the case of SDOS, the recursive encryption and decryption operations on the nodes, as well as data objects, could be implemented inside a HSM.

[1] HSM (Hardware Security Module) or custom FPGA/ASIC based solutions are common.

6.5.2 Executable Model in Docker Compose

Docker is a popular container[1] runtime for cloud applications at the time of writing this thesis. It provides a mechanism for packaging and delivering applications, as well as a standard way of running those containers. This enables developers to package their applications and run them on a variety of cloud platforms as well as regular workstations. Docker Compose allows developers to "compose" a larger application from multiple such containers. It enables packaging and running multiple containers as if they were a single application.

I used these two technologies to provide an automated deployment for MCM, which is available at: `https://github.com/sdos/deploy-sdos`. This Docker Compose file can be run on a local workstation for testing and development, or can be uploaded to a cloud provider. It contains a total of eight Docker containers that cover all the components shown in Figure 6.10. By default, this includes the Swift Object Store. However, it is possible to connect to an external Swift Object Store, which should be used for performance evaluation purposes. This is possible by changing the Docker Compose configuration; deploying Swift should be disabled and the associated Swift connection variables populated manually.

6.6 Related Publications

The contributions in this Chapter are supported by the following of my publications:

In 2013, a paper was accepted at BTW that discusses security issues in cloud application architectures. The distributed execution of SDOS represents such an application [WSM13].

Two of my publications discuss the representation of security aspects in the cloud application modeling language TOSCA [WWB+13b]. The implementation of these concepts is presented in a further publication at

[1] Here, "container" refers to a unit of program execution encapsulation. The word "container" otherwise refers to a data storage concept in this thesis.

DOA-Trusted Cloud [WWB+13a]. I co-authored a paper at GI-Jahrestagung in 2014 that discussed and presented an implementation of further use cases for TOSCA security policies [BKK+14]. The concepts presented in these publications allow a cloud application modeler to specify security requirements, like cryptographic deletion, in a declarative way. The TOSCA runtime environment could then add the SDOS component in-line with a cloud object store resource. However, for this thesis I decided to not implement a TOSCA model for SDOS and the MCM demonstrator system, but instead implement a working, executable model using Docker Compose, as explained above. I chose this approach because of the ubiquity of Docker models in research and industry today. Interested readers will find a Docker implementation more useful and are able to test and evaluate SDOS and MCM with little effort.

The application architecture of MCM is based on previous work on application architectures for data-intensive applications in cloud environments [RWWS14]. In another publication, I discuss the performance aspects of such distributed applications [MWL+14].

In 2015, a paper was accepted at the IBM Cloud Academy Conference that discusses different approaches to model-based deployment automation of cloud applications, with a special focus on parametrized or customized deployments based on application model transformations [WMW+15]. The findings in this contribution are the basis for some architectural decisions in MCM.

In 2017, a poster presentation about the Key-Cascade concept and the prototype of SDOS as well as the larger MCM system was accepted at BTW [Wai17]. In the same year, another poster presentation and demo that showed the integration with Trusted Platform Modules as deletable key source was accepted at EDBT [WWM17].

Chapter 7

Conclusions and Future Work

In this thesis, I discuss the challenge of securely erasing data, specifically in modern cloud storage systems. I show that the previous approaches, made for on-site data stores, are not applicable anymore in this new scenario and introduce the concept of cryptographic deletion. I discuss related work on data encryption, encrypted database systems, secure deletion in general, as well as cryptographic deletion. I point out that investigations into cryptographic deletion are few and far between. Most existing solutions have very different approaches from each other and a research gap still exists regarding their applicability to cloud storage systems.

Using a static tree with a re-keying operation is a novel approach that I investigated with the Key-Cascade method. This method provides a key management that enables efficient cryptographic deletion for encrypted cloud storage systems. To introduce the Key-Cascade, I present two initial concepts for applying cryptographic deletion to such storage systems. These initial approaches help in understanding the background and challenges.

Following this, I then present the Key-Cascade method with its hierarchical key management. I show the data structures of the Key-Cascade and detail how the operations on this structure are realized. I then calculate the geometries, storage, and computational requirements for differently sized Key-Cascades in order to show their suitability.

With SDOS, I provide a fully working implementation of the presented concepts. Besides providing a proof of concept prototype, another reason for building SDOS and MCM was to investigate the practical usability of the Key-Cascade method. As indicated by the extensions to the Key-Cascade method, which I also present in this thesis, further improvements were necessary in order to achieve a practically usable solution.

Using the SDOS prototype, I conducted performance measurements on secure deletion and the Key-Cascade operations. The results show that the API proxy can be a performance bottleneck. However, secure deletion only adds little cost to an encrypted storage system. The additional key management operations are cheap even for large Key-Cascade configurations. Designing and prototyping a solution, that transparently adds secure deletion to cloud storage while keeping the impact on response times low, is the main contribution of this work.

For these reasons, I consider the concepts in the Key-Cascade method, as well as the results of this thesis, a valuable addition to the existing research on cryptographic deletion. From the results of my evaluation, I conclude that the Key-Cascade method and its extensions provide a viable solution for secure data erasure in cloud storage systems.

Future Work

Cryptographic deletion is not a highly researched topic today. For this reason, many opportunities for future work exist in the concept of cryptographic deletion alone. The existing approaches, discussed in the related work, use a variety of different methods and data structures to solve the key management problem. The Key-Cascade is another such example, as it is the first that

uses re-keying on a static tree. Future research into improving, comparing, and applying the now existing methods is needed.

The Key-Cascade method itself could be expanded further as well. One possibility is to integrate access control into the data structure, a feature often found in encrypted multi-user storage systems. Another interesting area of research is applying the Key-Cascade method, or other approaches, to different storage systems. All previous approaches, including the Key-Cascade, focus on object stores or file systems due to their simple data model. An integration into database systems, especially encrypted database systems, poses an interesting research question.

I believe the most important area of future work lies in applying cryptographic deletion to existing encrypted storage systems, especially cloud storage systems and local disk encryption solutions. The results of this work, as well as the related work, show that cryptographic deletion can be used in practice. It is efficient enough, has acceptable storage overhead, and all necessary concepts and methods are present.

Interest from users as well as potential customers seems to be present, considering secure deletion is an established practice with on-site storage systems. The recent advances in legislation may drive demand even further. As I discussed in this thesis, cryptographic deletion can be viewed as a modification to the key management systems used today for data encryption. This opens the possibility of adding this feature to existing encrypted storage systems, lowering the effort for creating a production ready solution and also lowering the entry barrier for potential users.

Bibliography

[ABE+12] A. Arasu, S. Blanas, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, R. Ramamurthy, P. Upadhyaya, and R. Venkatesan. "Engineering Security and Performance with Cipherbase." In: *IEEE Data Eng. Bull.* 35.4 (2012), pp. 65–72. URL: http://sites.computer.org/debull/A12dec/cipher.pdf (cit. on p. 52).

[ABE+13] A. Arasu, S. Blanas, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, R. Ramamurthy, P. Upadhyaya, and R. Venkatesan. "Secure Database-as-a-service with Cipherbase." In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 1033–1036. URL: http://doi.acm.org/10.1145/2463676.2467797 (cit. on p. 52).

[AEJ+15] A. Arasu, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, and R. Ramamurthy. "Transaction processing on confidential data using cipherbase." In: *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE. 2015, pp. 435–446 (cit. on p. 52).

[BAV15] D. Brown, C. Arend, and A. Venkatraman. "EU Data Protection Reform Will Drive Growth in European Security and Storage Markets." In: *IDC ESS02X* (Oct. 2015) (cit. on p. 19).

[BBKL14] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann. "TOSCA: Portable Automated Deployment and Management of Cloud Applications." English. In: Advanced Web Services. New York: Springer, Jan. 2014, pp. 527–549. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INBOOK-2014-01&engl=1 (cit. on p. 146).

[Ber15] S. Beresford. *Deletion of records from national police systems*. Tech. rep. UK National Police Chiefs' Council, May 2015. URL: https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/430095/Record_Deletion_Process.pdf (cit. on p. 18).

[BKK+14] A. Blehm, V. Kalach, A. Kicherer, G. Murawski, T. Waizenegger, and M. Wieland. "Policy-Framework - Eine Methode zur Umsetzung von Sicherheits-Policies im Cloud-Computing." In: *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*. 2014, pp. 277–288. URL: http://subs.emis.de/LNI/Proceedings/Proceedings232/article54.html (cit. on p. 150).

[BL05] P. F. Bennison and P. J. Lasher. "Data security issues relating to end of life equipment." In: *Journal of ASTM International* 2.4 (2005), pp. 1–7 (cit. on p. 43).

[BLM+15] J. Barney, D. Lebutsch, C. Mega, S. Schleipen, and T. Waizenegger. "Deletion of content in digital storage systems." 9,298,951. U.S. patent. May 2015 (cit. on pp. 72, 124).

[BS14] S. Bajaj and R. Sion. "TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality." In: *IEEE Transactions on Knowledge & Data Engineering* 26.3 (2014), pp. 752–765 (cit. on p. 53).

[BV14] Z. Brakerski and V. Vaikuntanathan. "Efficient Fully Homomorphic Encryption from (Standard) LWE." In: *SIAM Journal on Computing* 43.2 (2014), pp. 831–871. eprint: http://dx.doi.org/10.1137/120868669. URL: http://dx.doi.org/10.1137/120868669 (cit. on p. 50).

[Con09] T. Conde. *To Delete or Not Delete - That's the Question: A Company's Obligations to Preserve Records Under the New Electronic Discovery Rules*. Tech. rep. Stoel Rives LLP, 2009. URL: http://www.stoel.com/to-delete-or-not-deletethats-the-question-a (cit. on p. 18).

[CSRL01] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. 3rd. McGraw-Hill Higher Education, 2001 (cit. on pp. 60, 63, 68).

[Day14] A. Dayley. *Financial Services Context: Magic Quadrant for Enterprise Information Archiving*. Tech. rep. G00262994. Gartner, Nov. 2014 (cit. on p. 19).

[Din14] R. Dingledine. *Tor security advisory: "relay early" traffic confirmation attack*. The Tor Project. July 2014. URL: https://blog.torproject.org/blog/tor-security-advisory-relay-early-traffic-confirmation-attack (cit. on p. 46).

[FG07] C. Fontaine and F. Galand. "A Survey of Homomorphic Encryption for Nonspecialists." In: *EURASIP Journal on Information Security* 2007.1 (2007), p. 013801. URL: http://dx.doi.org/10.1155/2007/13801 (cit. on p. 51).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Vienna, 2014, pp. 141, 136. URL: https://books.google.de/books?id=aum9BAAAQBAJ (cit. on p. 132).

[FS03] N. Ferguson and B. Schneier. *Practical Cryptography*. 1st ed. pages 83 ff. New York, NY, USA: John Wiley & Sons, Inc., 2003 (cit. on p. 37).

[Gen09] C. Gentry. "Fully Homomorphic Encryption Using Ideal Lattices." In: *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*. STOC '09. Bethesda, MD, USA: ACM, 2009, pp. 169–178. URL: http://doi.acm.org/10.1145/1536414.1536440 (cit. on p. 50).

[GHH+12] K. R. Grahlmann, R. W. Helms, C. Hilhorst, S. Brinkkemper, and S. van Amerongen. "Reviewing Enterprise Content Management: a functional framework." In: *European Journal of Information Systems* 21.3 (May 2012), pp. 268–286. URL: http://dx.doi.org/10.1057/ejis.2011.41 (cit. on p. 127).

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, p. 128 (cit. on p. 104).

[GKLL09] R. Geambasu, T. Kohno, A. Levy, and H. M. Levy. "Vanish: Increasing Data Privacy with Self-Destructing Data." In: *Proc. of the 18th USENIX Security Symposium*. 2009 (cit. on p. 46).

[Gol87] O. Goldreich. "Towards a Theory of Software Protection and Simulation by Oblivious RAMs." In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC '87. New York, New York, USA: ACM, 1987, pp. 182–194. URL: http://doi.acm.org/10.1145/28395.28416 (cit. on p. 44).

[Gro16] T. C. Group. *TPM 1.2 Protection Profile*. 2016. URL: https://www.trustedcomputinggroup.org/tpm-1-2-protection-profile/ (cit. on p. 41).

[GS03] S. Garfinkel and A. Shelat. "Remembrance of data passed: a study of disk sanitization practices." In: *Security Privacy, IEEE* 1.1 (Jan. 2003), pp. 17–27 (cit. on p. 33).

[Gue12] S. Gueron. *Intel Advanced Encryption Standard (AES) New Instructions Set*. 2012. URL: https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf (cit. on p. 38).

[Gut96] P. Gutmann. "Secure Deletion of Data from Magnetic and Solid-state Memory." In: *Proc. of the 6th USENIX Security Symposium*. SSYM'96. San Jose, California, 1996, pp. 8–8. URL: http://dl.acm.org/citation.cfm?id=1267569.1267577 (cit. on pp. 33, 42).

[HR99] T. Härder and E. Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer-Verlag, 1999, pp. 412–422 (cit. on p. 104).

[IBM] IBM. *IBM 4767-002 PCIe Cryptographic Coprocessor (HSM) Data sheet*. URL: http://www-03.ibm.com/security/cryptocards/pciecc2/pdf/4767_PCIe_Data_Sheet.pdf (cit. on p. 39).

[JT17] D. Janusz and J. Taeschner. "Privatsphäre-schützende Bereichsanfragen in unsicheren Cloud-Datenbanken." In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Workshopband*. Ed. by B. Mitschang,

N. Ritter, H. Schwarz, M. Klettke, A. Thor, O. Kopp, and M. Wieland. Vol. P-266. LNI. GI, 2017, pp. 237–247. URL: https://www.gi.de/service/publikationen/lni/gi-edition-proceedings-2017/gi-edition-lecture-notes-in-informatics-lni-p-266.html (cit. on p. 53).

[Ken05] K. Kenan. *Cryptography in the database: the last line of defense*. Upper Saddle River, NJ: Addison-Wesley, 2005, XXII, 277 p. (Cit. on p. 36).

[KSSL06] R. Kissel, M. Scholl, S. Skolochenko, and X. Li. “NIST special publication 800-88 Guidelines for Media Sanitization.” In: *Information Technology Laboratory, National Institute of Standards and Technology* (2006) (cit. on p. 43).

[McC14] R. McCormick. “Tim Cook says Apple will send security alerts to help stop iCloud hackers.” In: *The Verge* (Sept. 2014). URL: http://www.theverge.com/2014/9/4/6108695/apple-tim-cook-says-will-send-security-alert-stop-icloud-hack-nude-celebrities/in/5863348 (cit. on p. 20).

[Mel10] S. Melvin. *The Forgetful Disk Drive - Cryptographic Data Deletion Using Time-Based Key Management*. Tech. rep. Zytek Communications Corporation, 2010 (cit. on p. 47).

[MWL+14] C. Mega, T. Waizenegger, D. Lebutsch, S. Schleipen, and J. M. Barney. “Dynamic cloud service topology adaption for minimizing resources while meeting performance goals.” In: *IBM Journal of Research and Development* 58.2/3 (2014). URL: http://dx.doi.org/10.1147/JRD.2014.2304771 (cit. on p. 150).

[Nat01] National Institute for Standards and Technology. *AES Key Wrap Specification*. 2001. URL: http://csrc.nist.gov/groups/ST/toolkit/documents/kms/key-wrap.pdf (cit. on pp. 35, 37).

[oOnt08] A. of Ontario. *Recordkeeping Fact Sheet; The Fine Art of Destruction: Weeding Out Transitory Records*. Tech. rep. Recordkeeping Support Unit, Ministry of Government Services, June 2008. URL: http://www.archives.gov.on.ca/en/recordkeeping/documents/Fact-Sheet-Transitory-Records.pdf (cit. on pp. 18, 19).

[Ost09] A. Ostrow. “Facebook Responds to Concerns Over Terms of Service.” In: *Mashable* (Feb. 2009). URL: http://mashable.com/2009/02/16/facebook-tos-response/#pAfXlF1j4Eqr (cit. on p. 20).

[PC16] E. Parliament and Council. *Regulation 2016/679: General Data Protection Regulation (GDPR)*. Apr. 6, 2016, Article 13 "Right to be Forgotten". URL: http://eur-lex.europa.eu/legal-content/EN-DE/TXT/?uri=CELEX:32016R0679 (cit. on p. 18).

[Per05a] R. Perlman. “File system design with assured delete.” In: *Third IEEE International Security in Storage Workshop (SISW’05)*. Dec. 2005, p. 6 (cit. on p. 44).

[Per05b] R. Perlman. *The Ephemerizer: Making Data Disappear*. Tech. rep. Mountain View, CA, USA: Disappearing Inc., 2005 (cit. on p. 44).

[Pid11] H. Pidd. “Facebook could face EUR 100,000 fine for holding data that users have deleted.” In: *The Guardian* (Oct. 2011). URL: http://www.theguardian.com/technology/2011/oct/20/facebook-fine-holding-data-deleted (cit. on p. 20).

[PRZB11] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. “CryptDB: Protecting Confidentiality with Encrypted Query Processing.” In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP ’11. Cascais, Portugal: ACM, 2011, pp. 85–100. URL: http://doi.acm.org/10.1145/2043556.2043566 (cit. on p. 51).

[RBC13] J. Reardon, D. Basin, and S. Capkun. “SoK: Secure Data Deletion.” In: *Security and Privacy (SP), 2013 IEEE Symposium on*. Institute of Electrical & Electronics Engineers (IEEE), May 2013, pp. 301–315. URL: http://dx.doi.org/10.1109/SP.2013.28 (cit. on p. 48).

[RBC14] J. Reardon, D. A. Basin, and S. Capkun. “On Secure Data Deletion.” In: *IEEE Security & Privacy* 12.3 (2014), pp. 37–44. URL: http://dx.doi.org/10.1109/MSP.2013.159 (cit. on p. 48).

[RBM13] O. Rodeh, J. Bacik, and C. Mason. “BTRFS: The Linux B-Tree Filesystem.” In: *Trans. Storage* 9.3 (Aug. 2013), 9:1–9:32. URL: http://doi.acm.org/10.1145/2501620.2501623 (cit. on p. 79).

[Rei02] J. A. Reimer. “Enterprise Content Management.” In: *Datenbank-Spektrum* 4 (2002), pp. 17–22 (cit. on p. 127).

[RRBC13] J. Reardon, H. Ritzdorf, D. Basin, and S. Capkun. “Secure Data Deletion from Persistent Media.” In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS ’13. Berlin, Germany: ACM, 2013, pp. 271–284. URL: http://doi.acm.org/10.1145/2508859.2516699 (cit. on p. 48).

[RWWS14] P. Reimann, T. Waizenegger, M. Wieland, and H. Schwarz. “Datenmanagement in der Cloud fur den Bereich Simulationen und Wissenschaftliches Rechnen.” In: *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*. 2014, pp. 735–746. URL: http://subs.emis.de/LNI/Proceedings/Proceedings232/article176.html (cit. on p. 150).

[Sch09] B. Schneier. “The battle is on against Facebook and co to regain control of our files.” In: *The Guardian* (Sept. 2009). URL: https://www.schneier.com/blog/archives/2009/09/file%5C_deletion.html (cit. on p. 20).

[Sch11] S. Schroeder. “Facebook Facing USD 138,000 Fine for Holding Deleted User Data.” In: *Mashable* (Oct. 2011). URL: http://mashable.com/2011/10/21/facebook-deleted-data-fine/#P4KFGXySP0q3 (cit. on p. 20).

[Sch14] R. Schafer. *Magic Quadrant for IT Asset Disposition Worldwide*. Tech. rep. G00261957. Gartner, Dec. 2014 (cit. on p. 19).

[TLLP12] Y. Tang, P. Lee, J. Lui, and R. Perlman. “Secure Overlay Cloud Storage with Access Control and Assured Deletion.” In: *Dependable and Secure Computing, IEEE Transactions on* 9.6 (Nov. 2012), pp. 903–916 (cit. on p. 47).

[Wai12] T. Waizenegger. “Data security in multi-tenant environments in the cloud.” Englisch. Diplomarbeit. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Apr. 2012. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3242&engl= (cit. on p. 72).

[Wai17] T. Waizenegger. "Secure Cryptographic Deletion in the Swift Object Store." In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*. Ed. by B. Mitschang, D. Nicklas, F. Leymann, H. Schöning, M. Herschel, J. Teubner, T. Härder, O. Kopp, and M. Wieland. Vol. P-265. LNI. GI, 2017, pp. 625–628. URL: https://www.gi.de/service/publikationen/lni/gi-edition-proceedings-2017/gi-edition-lecture-notes-in-informatics-lni-p-265.html (cit. on pp. 73, 124, 150).

[WMW+15] F. Wagner, C. Mega, T. Waizenegger, V. Raskin, and S. Kukhtichev. "A Practical Approach to Model-Based Cloud Service Deployment: Using a Smart Interpreter and a Domain Specific Language." Englisch. In: *Proceedings of the 3rd International IBM Cloud Academy Conference ICACON 2015*. IBM, May 2015, pp. 1–10. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INMISC-2015-05&engl= (cit. on p. 150).

[WSM13] T. Waizenegger, O. Schiller, and C. Mega. "Datensicherheit in mandantenfähigen Cloud Umgebungen." In: *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*. Ed. by V. Markl, G. Saake, K. Sattler, G. Hackenbroich, B. Mitschang, T. Härder, and V. Köppen. Vol. 214. LNI. GI, 2013, pp. 477–489. URL: http://www.btw-2013.de/proceedings/Datensicherheit%20in%20mandantenfaehigen%20Cloud%20Umgebungen.pdf (cit. on p. 149).

[WWB+13a] T. Waizenegger, M. Wieland, T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, B. Mitschang, A. Nowak, and S. Wagner. "Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing." In: *On the Move to Meaningful Internet Systems: OTM 2013 Conferences - Confederated International Conferences: CoopIS, DOA-Trusted Cloud, and ODBASE 2013, Graz, Austria, September 9-13, 2013. Proceedings*. Ed. by R. Meersman, H. Panetto, T. S. Dillon, J. Eder, Z. Bellahsene, N. Ritter, P. D. Leenheer, and D. Dou. Vol. 8185. Lecture Notes in Computer Science. Springer,

2013, pp. 360–376. URL: http://dx.doi.org/10.1007/978-3-642-41030-7_26 (cit. on p. 150).

[WWB+13b] T. Waizenegger, M. Wieland, T. Binz, U. Breitenbücher, and F. Leymann. "Towards a Policy-Framework for the Deployment and Management of Cloud Services." Englisch. In: *SECURWARE 2013, The Seventh International Conference on Emerging Security Information, Systems and Technologies*. Ed. by H.-J. Hof and C. Westphall. Barcelona, Spain: IARIA, Aug. 2013, pp. 14–18. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2013-44 (cit. on p. 149).

[WWM17] T. Waizenegger, F. Wagner, and C. Mega. "SDOS: Using Trusted Platform Modules for Secure Cryptographic Deletion in the Swift Object Store." In: *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*. Ed. by V. Markl, S. Orlando, B. Mitschang, P. Andritsos, K. Sattler, and S. Breß. OpenProceedings.org, 2017, pp. 550–553. URL: http://dx.doi.org/10.5441/002/edbt.2017.67 (cit. on pp. 73, 124, 150).

[Yod13] K. Yoder. *POWER7+ Accelerated Encryption and Random Number Generation for Linux*. Tech. rep. IBM Linux Technology Center, 2013 (cit. on p. 38).

[ZDB09] V. J. Zimmer, S. R. Dasari, and S. P. Brogan. *Tcg-based Firmware. White Paper*. Intel Corporation and IBM Corporation Trusted Platforms, 2009 (cit. on p. 42).

All URLs were last followed on 25.04.2017.

List of Figures

1.1 General principle of cryptographic deletion. 20
1.2 Representation of the adversary model. 23
1.3 The individual key approach: one separate key per object. . . 25
1.4 The master key approach: a single key for all objects. The re-keying operation allows selective deletion. 26
1.5 System overview showing local and cloud components. 29

2.1 Methods to secure data deletion. The cryptographic deletion method presented in this thesis follows the highlighted path. 33
2.2 Simplified wrapped key and chain of wrapped keys. 35
2.3 IBM 4767 PCIe HSM with secure enclosure [IBM]. 39
2.4 General HSM/TPM operation. Keys never leave the secure enclosure. 40
2.5 TPM internal components. Non-migratable keys can only be used inside the TPM. 41
2.6 Deterministic (b) compared to randomized (c) encryption. . . 52

3.1 Overview of the reference application. 57
3.2 Construction of a new Key-Cascade with tree height 2 and node size 4. 58

3.3 Structure of a single Key-Cascade with height 2 and a node size of 4. Nodes 3 and 4 are not shown. 60
3.4 Key-Cascade objects and their storage location. 63
3.5 Representation of a Key-Cascade's tree structure showing only the nodes and IDs. Height $h = 2$, node size $S_n = 4$. 65
3.6 The path to the object key for Object 7, found through ID calculations. 67
3.7 Cascaded re-keying results in cryptographic deletion of Object 7. 69

4.1 Introducing the ID enumeration representation compared to the tree representation. Both figures represent the same Key-Cascade. 77
4.2 Cascade growing with regular ID enumeration. 77
4.3 Cascade growing with alternative 2D ID enumeration. 78
4.4 In the implementation, a chain of two keys is used as root key for the Key-Cascade. Root key and nodes are handled by the Key-Cascade code, deletable key and en/decryption of the root key are provided by "key sources" (see Section 4.3). . . . 80
4.5 Key source interaction: initialize a new root key with the `initialize_root_key` operation. 83
4.6 Key source interaction: decrypt the current root key with the `decrypt_current_key` operation. 84
4.7 Key source interaction: securely delete and replace the deletable key with the `encrypt_next_key` operation. 84

5.1 Architecture of the SDOS prototype showing the three operational modes. 93
5.2 Account, container, and object hierarchy in Swift. 96
5.3 Mapping of SDOS data structures to Swift Object Store. . . . 97
5.4 Sequence of operations for storing a new object. 99
5.5 Thread synchronization during writing. 104
5.6 Thread synchronization during cascaded re-keying. 105

5.7 Parallel processing methods with distributed data structures and key source access. 107
5.8 Pseudo-object API compared to requests on regular and encrypted objects. Showing object / information retrieval. . . . 109
5.9 Layout of the evaluation setup for SDOS. 113
5.10 Measurement results comparing a direct client-to-server connection (Configuration C) to the API proxy (Configuration B3). 116
5.11 Measurement results comparing a direct client-to-server connection (Configuration C) to the API proxy (Configuration B3). Results show the average response-time of the 95th percentile of fastest requests. 118
5.12 Measurement results showing the added cost of encryption and Key-Cascade (Configurations B1, 2, 3) for read and write operations. 120
5.13 Measurement results showing the cost of the secure delete operation. The node cache was used and the regular delete operation is shown for comparison. 122

6.1 A high-level overview of the MCM components. The three custom components are Bluebox, SDOS, and the task runner. 129
6.2 Internal components of the Bluebox user interface. 130
6.3 Screenshot of the container view in Bluebox. 131
6.4 Screenshot of the container creation dialog in Bluebox showing the options for SDOS. 136
6.5 Screenshot of the object view in Bluebox. Information retrieved from the pseudo-object API is shown on the right. . . 137
6.6 Screenshot of the Key-Cascade visualization in Bluebox. An animation shows the steps involved in cascaded re-keying. . . 139
6.7 The process of extracting and replicating metadata from objects.140
6.8 Screenshot of the task execution view in Bluebox. The request message and two replies from a task runner are shown. 141

6.9 Screenshot of the analytics view in Bluebox. Template charts are filled with query result data. 142
6.10 A high-level overview of the MCM components. The three red lines show possible deployment configurations. 147

List of Tables

1.1 Storage and computational overhead comparison for preliminary approaches. Using an object store with 16,777,216 objects. 28

3.1 Storage and computational overhead comparison for preliminary approaches from Section 1.5 and Key-Cascade. Using an object store with 16,777,216 objects and the Key-Cascade from Example 2. 66

5.1 Number of object keys for different Key-Cascade geometries. Node sizes in Bytes are for AES256 encryption keys. 122

Definitions

1.1 Re-keying operation 26
3.1 Cascaded Re-Keying 69