Flexible Secure Cloud Storage

Dissertation submitted for the degree of

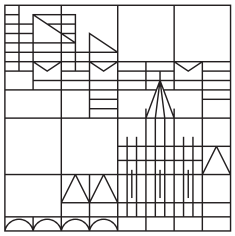
Doctor of Engineering (Dr.-Ing.)

Presented by

Sebastian Graf

at the

Universität Konstanz



Faculty of Sciences

Department of Computer and Information Science

Date of the oral examination: 29.1.2014
First supervisor: Prof. Dr. Marcel Waldvogel
Second supervisor: Prof. Dr. Hannes Hartenstein

Abstract

Our life without Internet-based services is hard to imagine: We search for information with Google, share thoughts on Facebook, buy at Amazon and store our pictures on Flickr. Many of these Internet-based services focus on easy exchange of information, providing comfortable and ubiquitous storage and sharing. Relieved from hardware purchases, software bug fixes and infrastructure maintenance, users as well as companies use these cloud-based stores either for free or at low-cost. The price is the implicit grant of full access to all their sensitive data.

The stored data naturally represents a huge pool of easily accessible and alluring information for cloud providers. Customer questions like "Who accesses my information?" (representing the aspect of confidentiality), "Who altered my data?" (requiring accountability), "Is my data still intact?" (focusing on integrity) or "What happens if the cloud is unavailable?" can rarely be answered in an obligingly and honest way. Answering these questions is challenging since security measures seldom cover all security aims at once. Furthermore, the cloud is used with all kinds of data, wishing their unique characteristics to be respected.

Each of these questions above is transformed closer to an answer in this thesis resulting in an architecture jointly satisfying all the denoted security aims. A versatile key management offers flexible group shares by providing fine-grained access on end-to-end encrypted data. The keys furthermore enable time-based access on versioned storage and are provisioned over the cloud itself without harming confidentiality. Versioning of the data protects accountability in storage. This is tailored to the remote location offering auto-configured checks, constant reconstruction and evened out transfer rates of change sets. The versioning is provided by a sophisticated bucket structure. Hierarchically ordered data provides recursive integrity checks and atomic operations covering multiple buckets. Providing automatic protection of integrity and accountability, the resulting bucket arrangement is implemented by data containers offering storage of all kinds of data. Results show that the storage of blocks, files and even XML in its structural representation becomes possible. The result is a conceptually simple, transparent, yet powerful architecture to bring data securely and efficient to the cloud. The extensibility of the architecture is proven by taking advantage of photo sharing websites as No-SQL stores to shake up the closed market of expensive No-SQL cloud storage providers.

Besides these contributions guarding security on a technical level, this thesis provides an outlook exceeding the area of computer science. The architecture is interpreted from the legal point of view not only increasing confidence in the techniques developed. The resulting mapping offers a bridge between computer scientist and legal experts to exchange knowledge about necessary measures. The need for this cooperation increases as intransparent, maybe even illegal, access to Internet-stored data seem to become the favorite pass-time of governments around the world.

Zusammenfassung

Ein Leben ohne das Internet ist schwer vorstellbar: Wir suchen nach Informationen mit Google, tauschen Gedanken über Facebook aus, kaufen bei Amazon und speichern unsere Bilder bei Flickr. Viele dieser Dienste kümmern sich um den einfachen Informationsaustausch, was in komfortablem und weit verbreitetem, gemeinsam nutzbaren Speicher resultiert. Da Hardwarekäufe, Behebung von Softwarefehlern und die Wartung der Infrastruktur obsolet werden, verwenden Endnutzer ebenso wie Firmen kostenlose oder preiswerte cloudbasierte Speicher. Der Preis ist dabei das implizite Zugriffsrecht auf alle sensiblen Daten.

Natürlich stellen die gespeicherten Daten eine grosse Masse an einfach erreichbaren und verführerischen Informationen für die Anbieter von Cloudspeichern dar. Fragen von Kunden wie beispielsweise "Wer hat Zugriff auf meine Informationen?" (was den Aspekt der Vertraulichkeit abdeckt), "Wer hat meine Daten verändert?" (was die Nachvollziehbarkeit hinterfragt), "Sind meine Daten noch intakt?" (was die Integrität in den Vordergrund rückt) oder "Was passiert wenn die Cloud nicht erreichbar ist?" können nur schlecht in einer verbindlichen und ehrlichen Weise beantwortet werden. Die Beantwortung dieser Fragen ist schwierig, da Sicherheitsmechanismen selten mehrere Sicherheitsaspekte auf einmal abdecken. Desweiteren werden verschiedene Arten von Daten in der Cloud gespeichert, wobei jede dieser Arten eigene Charakteristika aufweist die es zu berücksichtigen gilt.

Jede dieser oben stehenden Fragen wird in dieser Arbeit anhand einer Architektur behandelt, welche alle beschriebenen Sicherheitsaspekte erfüllt. Ein vielseitiges Schlüsselmanagement ermöglicht flexible Freigaben für Gruppen und stellt fein-granularen Zugriff auf Ende-zu-Ende-verschlüsselte Daten zur Verfügung. Die Schlüssel ermöglichen ausserdem zeitbeschränkten Zugriff auf versionierte Speicher und werden in der Cloud selbst vorgehalten ohne die Vertrauenswürdigkeit zu verletzen. Die Versionierung der Daten schützt die Nachvollziehbarkeit im Speicher. Zugeschnitten auf die entfernt liegende Speicherung bietet die Versionierung selbst-konfigurierte Überprüfung der Daten, konstante Rekonstruktion und gleichmässige Transferraten von Änderungen an. Die Versionierung wird bereitgestellt durch eine ausgeklügelte Bucketstruktur. Hierarchisch angeordnete Daten erlauben rekursive Integritätschecks und atomare Operationen auf mehreren Buckets. Das resultierende Bucketgefüge stellt automatischen Schutz von Integrität und Nachvollziehbarkeit zur Verfügung und ist implementiert mit verschiedenen Datencontainern was die Speicherung von allen Arten von Daten ermöglicht. Ergebnisse zeigen die Machbarkeit der Speicherung von Blöcken, Dateien und XML unter Berücksichtigung seiner Struktur. Das Resultat ist eine konzeptionell einfache, transparente aber mächtige Architektur, welche Daten sicher und effizient in der Cloud speichert. Die Erweiterbarkeit der Architektur zeigt, dass man Photowebseiten als No-SQL Datenbanken nutzen kann um den geschlossenen Markt von teuren No-SQL Speichern in der Cloud aufzubrechen.

Neben den Beiträgen, welche sich um Sicherheit auf einem technischen Level kümmern, stellt diese Arbeit einen Ausblick bereit, welche das Gebiet der Informatik verlässt. Die Architektur wird von einem rechtlichen Standpunkt beleuchtet, was nicht nur

das Vertrauen in die entwickelten Techniken stärkt. Die resultierende Zuordnung baut eine Brücke zwischen Informatikern und Rechtsexperten um Wissen über notwendige Massnahmen austauschen zu können. Die Notwendigkeit dieser Kooperation ergibt sich aus undurchsichtigen, wahrscheinlich sogar illegalen, Zugriffen auf Daten im Internet, was momentan die Lieblingsbeschäftigungen von Regierungen auf der ganzen Welt darstellt.

Contents

All things are difficult before they are easy.

Thomas Fuller - English Historian / 1608 - 1661

1 Preface

1.1 Summary

The quote at the beginning motivates the simplifying of complex approaches, algorithms and thoughts. The main ideas should be recognizable for everyone.

So, before diving in the technical depths of algorithms, data structures, evaluations and discussions, the focus is very briefly described:

Imagine the possibility to store information forever and make them accessible from everywhere. Thanks to the Internet, this possibility recently got a name: cloud storage. Cloud storage can be considered as internet-based disks being always available. They offer the ability to share any kind of information. The benefits of internet-based storage come at a price: The loss of any physical possession makes it hard to control who accesses the data. One example is the problem of guaranteed deletion of cloud-based data. Even if the user deletes data, the data can (and often does) still physically exist in the infrastructure of the provider. Practically, the data owner is not the user any more. The cloud storage provider has full access to the data. If the provider removes data, it becomes inaccessible for the user if not stored elsewhere. As a result, no guarantee about accesses or changes can be made.

In this thesis, a solution to the problem of confidential, traceable and consistent storage in the cloud is presented. An own defined architecture, mapped to the cloud, offers integrity checks and storage of different versions. As a result, no one except the user as well as granted groups and persons can make use of the data. Modifications on the data are not only easily detectable but also traceable. To enable collaboration, an adaptive key management provides sharing of encrypted information. The resulting architecture is a pleading for a tight interaction of measures: Only the satisfaction of different security aims guards the data physically distributed in the Internet.

In a nutshell, by applying the methods developed in this thesis, users are able to store data securely in the internet.

1.2 Structure of this thesis

The structure of this thesis is depicted by Figure 1.1.

An introduction outlines the scope of this thesis in Chapter 2. Starting with a brief motivation, the claims of this thesis are described. To give a more detailed overview

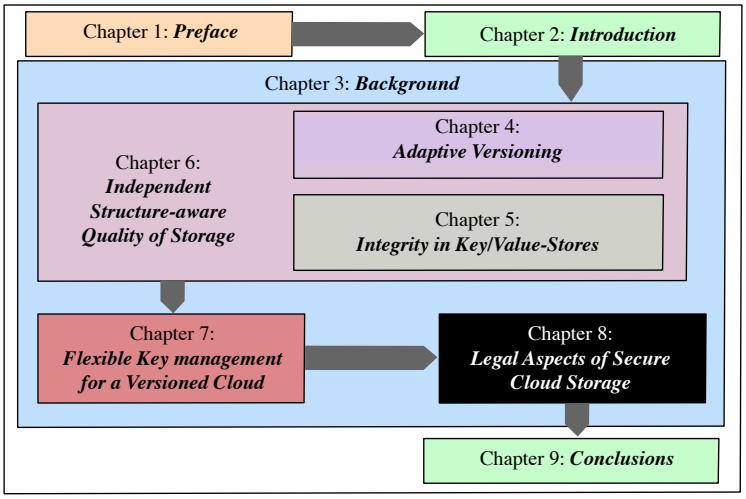


Figure 1.1: Structure of this Thesis

of the developments leading to this work, contributions consisting of publications and projects are listed.

Chapter 3 establishes the basis for the upcoming chapters. Besides the idea of cloud services in common and cloud storage in particular, current classifications are discussed. These classifications map different levels of cloud services to common security requirements. This mapping results in a position statement on the research field. Five main security requirements, namely, availability, integrity, confidentiality, accountability and assurance, are identified. These security aims need to be satisfied in the cloud and act as leitmotif in the rest of the thesis.

Chapter 4 presents an adapted versioning approach applicable on remote data. This versioning is combined with a mechanism to guard data stored in No-SQL databases described in Chapter 5. Buckets, representing the values in stored key/value-pairs, are ordered hierarchically. The resulting bucket order combines efficient integrity checks with the described versioning approach. Chapter 6 describes and evaluates the resulting architecture. Different mappings of input data with distinct characteristics make use of the versioned, integrity-protected bucket architecture. The resulting architecture relies on an abstract backend. Different cloud infrastructures including photo sharing websites become thereby accessible.

The buckets are symmetrically encrypted. A flexible key management handles the keys as described in Chapter 7. This key management not only respects the versioning of the data. It can take advantage even of untrusted cloud storage to propagate changes in the key set.

All developed technical measures are evaluated regarding common legal aspects in Chapter 8. This evaluation represents one aspect of assurance beyond technical possibilities. The conclusion in Chapter 9 summarizes the approaches, maps them to the claims and gives an outlook to future work.

There's no way that company exists in a year.

Tom Siebel - CEO of C3 / 1952 -

2 Introduction

Contents

Tom Siebel, founder of Siebel CRM Systems, made this statement in 2001 about Salesforce.com. Salesforce.com is one of the main cloud service providers nowadays. Ironically, Siebel CRM Systems itself does not exist anymore. Salesforce.com in the meanwhile represents one of the leading platforms hosting cloud-based web applications. Companies having their home base in the IT-environment like Amazon push cloud services from a developers platform to end-user products. The abstraction gained by the cloud brings automatic synchronization, easy sharing and guaranteed availability. The promise of flexibility, scalability, omnipresence and availability generates a paradigm shift from self-hosted services to globally hosted services. Such a service runs on massive infrastructure of global players like Google, Amazon, Microsoft, and Apple. Persisting data in the cloud offers new possibilities of synchronization and sharing. The price for storing data in the cloud is the physical loss. The unknown location of the stored data, ignoring country borders including regulations, naturally generates concerns about its security. Recent information about access of security agencies to the intranet of cloud providers let this anxiety grow further.

2.1 Cloud Storage and Security

Cloud storage enables simple synchronization of data among multiple devices. Its omnipresence in the Internet is interpreted as all-time availability of personal data. Fast Internet connections are both, durable and affordable. Combined with the increasing computing power of even hand-held devices, applications relying on synchronization become standard today.

Sharing information is the next, logical step extending the synchronization of information. Suitable authorization mechanisms enable thereby collaborative workflows in the cloud. These workflows make use of the permanent available, location-independent access to the data.

Cloud-based services offer much higher and guaranteed availability than local services. User-hosted infrastructure hardly reaches the uptime of a cloud service. The availability starts at 99%. The "numbers of nines" from the site of the cloud storage provider defines the exact percentage representing the decimals.

This availability is achieved by mirroring services over multiple sites. By abstracting services, hardware and software-platforms become encapsulated and independent. Resources of cloud services appear to be infinite: Services are adjustable to the customer's needs even at runtime: Storage is flexibly adapted offering billing by actual usage only. This paradigm is known as pay-as-you-go representing one of the main characteristics of cloud services.

These benefits come at a cost:

- The access to cloud-based data is characterized by its storage over the Internet. The resulting distance to the cloud increases the costs of its usage massively as described by Chen and Radu [43].
- The world-wide distribution of the data makes the stored data susceptible to any world-wide executable attacks. The hosting companies themselves furthermore possibly access the data internally. Current incidents, in particular the uncontrolled access to such data from the site of governments, intensify these concerns.
- The scalability of cloud-based services results in different billing models, ranging from flat rates to pay-as-you-go. Security measures might generate an overhead by consuming more space and requesting additional data. Security in the cloud thereby might result in additional costs.

The nested nature of cloud storage increases the need for well-established technical measures protecting the data. These security measures must compensate the lost of physical control. Additionally, the applied security mechanisms must adhere cloud storage benefits, namely collaboration and synchronization by minimizing the costs.

2.2 Contributions

Adaptive Versioning: Modern storage systems provide mechanisms for backup, scrubbing, versioning, and defragmentation; but they are provided separately and require complex tuning and configuration, depending on system load and patterns. Chapter 4 shows all this can be achieved in a single simple, elegant and powerful mechanism by generalizing versioning, especially suited for cloud storage. One single parameter offers flexible adjustments, tunable to the characteristics of the cloud storage providers in use.

Integrity in Key/Value-Stores: Most cloud storage APIs are based on stateless REST [53], requiring multiple, independent operations to be performed for all but the simplest operations. Even if protecting integrity on multiple buckets would not already fail in the transfer, ACID conformity in the eventually consistent

data model is out of question. By adapting techniques from log-structured file systems, Chapter 5 presents hierarchical integrity checks, fine-granular versioning and COW, resulting in ACID operations even over REST. The resulting bucket arrangement extends well-established mechanisms from the area of log-structured file systems offering hierarchical integrity checks and fine-granular versioning.

Securing all Data in No-SQL Stores: No-SQL stores as used for most cloud storage do not provide interfaces directly usable by end users or application not specifically written for the cloud. A single security layer working efficiently with a wide variety of interface layers provides seamlessly integration of applications not specifically tuned for the cloud. Chapter 6.1 presents a mapping of all kinds of data to No-SQL stores providing automatic protection of confidentiality, integrity and accountability in amorphous buckets. Encapsulating data in stackable data containers results in infinite flexibility as shown by storing and benchmarking four example data kinds.

Photo Sharing Websites as Complimentary Cloud Storage: Professional cloud storages is mostly accessible as No-SQL stores billed by traffic, request count and storage amount while application-dependent cloud storage like photo sharing websites are frequently free for end-users. In Chapter 6.2, the vast storage capabilities of Facebook, Picasa and Flickr are exploited as No-SQL stores by mapping data into images transparently as part of a container system. When access speed is not critical, these services provide excellent bang for the buck while retaining characteristics of professional cloud storage, including ubiquitous access, high availability, and adjustable resources.

Flexible Key Management for a versioned Cloud: Collaboration, synchronization and ubiquitous access represent the major goals for storing data in the cloud. To take advantage of the confidentiality of encryption for a group of users, the key management problem needs to be solved and match the high availability of the data. In Chapter 7, a versatile, group-based key management is presented offering methods for time-based access using the versioning of the data. The key management is separated into an on-demand updating mechanism for adapting the keys and an always-on distribution mechanism for propagating existing keys in a confidential way.

Legal Aspects on Secure Cloud Storage: Recent developments in the political landscape show the need for a common language bridging technical measures and legal regulations. Chapter 8 maps German laws and security techniques to commonly well-defined security aims. The resulting mapping embraces a joint terminology for computer and legal experts and ties thereby disjoint areas with distinct vocabulary together.

2.2.1 Practical Results

The results in this thesis are generated mostly by own developed open source software. The software is tested, documented and freely available. These projects are described in detail (sorted chronologically).

- **jSCSI**
  jSCSI represents the first entirely Java implementation of the iSCSI standard [80]. The initiator was implemented as a master's project in 2007 [116]. jSCSI was extended with multithreading support on the initiator's side [39, 58], a target [50], a storage pool system [82] and a direct binding to cloud storage systems. jSCSI today is used by several companies. Freely available under the 3-clause BSD-License at `http://jscsi.org`, jSCSI is the iSCSI-interface for Treetank.

- **Perfidix**
  Perfidix is originated from the need to generate benchmarks of Java source code in a reliable way. Created in 2007 [79], Perfidix is inspired by the usage of jUnit and TestNG. Java methods are annotated. Their execution is measured by different meters like time, storage consumption, threads or individual counters. Perfidix comes with a fully functional Eclipse integration named Perclipse. Both software projects are freely accessible under the 3-clause BSD-License at `http://perfidix.org`. All Java-based evaluations are relying on Perfidix.

- **Treetank**
  Idefix, the former name of Treetank, represented the Phd-project of Marc Kramis. Developed as native, versioned XML storage [60], the project was renamed into Treetank in 2009. The project evolves into a storage system not only satisfying XML as input format. It also offer interfaces to store data as RESTful services, files, XML as well as iSCSI blocks [56, 97]. Treetank is freely available under the BSD 3-clause license at `http://treetank.org`. The current status of Treetank offers bindings to different cloud storage providers. It includes a combination of the described security measures. Treetank thereby represents the core contribution of this thesis.

- **JAX-RX**
  Together with the Database & Information Systems Group (DBIS) of the University of Konstanz, JAX-RX was developed as REST interface layer. JAX-RX offers RESTful access to tree-based storage [62]. Used as prototyping platform for various publications [63, 65], JAX-RX still remains open source representing the REST interface to Treetank.

- **jClouds**
  jClouds is an open source binding for accessing cloud resources in Java over common interfaces. This API was extended by a master's thesis to use photo sharing websites as cloud storage backends [64, 89]. This extension is freely available under the Apache License at `http://github.com/disy/jclouds`.

2.2.2 Publications

Besides the practical outcome, several milestones in this thesis were published in chronological order.

1. Marc Kramis, Volker Wildi, Bastian Lemke, Sebastian Graf, Halldór Janetzko and Marcel Waldvogel. jSCSI - A Java iSCSI Initiator. Presented at the Jazoon. 2007
   http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-84424

2. Marc Kramis, Alexander Onea and Sebastian Graf. Perfidix: A Generic Java Benchmarking Tool. Presented at the Jazoon. 2007
   http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-84446

3. Sebastian Graf, Marc Kramis and Marcel Waldvogel. Distributing XML with Focus on Parallel Evaluation. Presented at the DBISP2P. 2008
   http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-84487

4. Sebastian Graf, Patrice Brend'amour and Marcel Waldvogel. jSCSI 2.0: Multithreaded Low-Level Distributed Block Access. Tech Report
   2009 http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-84511

5. Sebastian Graf. Treetank, a native XML storage. Tech Report. 2009
   http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-100664

6. Sebastian Graf, Lukas Lewandowski and Marcel Waldvogel. Integrity Assurance for RESTful XML. Presented at WISM. 2010
   http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-123507

7. Sebastian Graf, Lukas Lewandowski and Christian Grün. JAX-RX - Unified REST Access to XML Resources. Tech Report. 2010
   http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-120511

8. Sebastian Graf, Vyacheslav Zholudev, Lukas Lewandowski and Marcel Waldvogel. Hecate, Managing Authorization with RESTful XML. WS-REST. 2011
   http://nbn-resolving.de/urn:nbn:de:bsz:352-126237

9. Sebastian Graf, Sebastian Belle and Marcel Waldvogel. Rolling Boles, Optimal XML Structure Integrity for Updating Operations. WWW, Poster. 2011
   http://nbn-resolving.de/urn:nbn:de:bsz:352-126226

10. Sebastian Graf, Lukas Lewandowski, Johannes Lichtenberger, Marc Kramis and Marcel Waldvogel. Treetank, Designing a Versioned XML Storage. XMLPrague, Poster. 2011
    http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-126912

11. Sebastian Graf. A Secure Cloud Gateway based upon XML and Web Services. ECOWS, PhD Symposium. 2011
    http://nbn-resolving.de/urn:nbn:de:bsz:352-154112

12. Sebastian Graf, Jörg Eisele, Marcel Waldvogel and Marc Strittmatter. A Legal and Technical Perspective on Secure Cloud Storage. DFN-Forum Kommunikationstechnologien. 2012
http://nbn-resolving.de/urn:nbn:de:bsz:352-192389

13. Sebastian Graf, Patrick Lang, Stefan Hohenadel and Marcel Waldvogel. Versatile Key Management for Secure Cloud Storage. DISCCO. 2012
http://nbn-resolving.de/urn:nbn:de:bsz:352-200971

14. Sebastian Graf, Wolfgang Miller and Marcel Waldvogel. Utilizing Photo Sharing Websites for Cloud Storage. Tech Report. 2013
http://nbn-resolving.de/urn:nbn:de:bsz:352-234273

15. Sebastian Graf, Andreas Rain and Marcel Waldvogel. "You can find my CV on LinkedIn..." - Privacy-Aware Distributed Social Networking for Research Facilities. Tech Report. 2013
http://nbn-resolving.de/urn:nbn:de:bsz:352-212815

[...]the part of a picture, scene, or design that forms a setting for the main figures or objects, or appears furthest from the viewer.

Definition of Background, Oxford Dictionaries

3 Background

Contents

The overall picture of secure cloud storage seems overwhelming by looking at the research activity and existing products. The chapter represents a trade-off between a high-level overview and recent approaches. This high-level overview starts with a clear definition of the term "security". The approaches described in this thesis map this definition. Research approaches are described and compared relying on the defined security aims. The techniques, described in the next chapters, include, if appropriate, additional related background. The definition of cloud storage models and security aims is published as joint work under the title "A Legal and Technical Perspective on Secure Cloud Storage" in DFN-Forum Kommunikationstechnologien, 2012 [59]. The contribution in the paper and re-used in this chapter includes all technical definitions.

3.1 Security Requirements

"CIA" [48] represents the most common definition of security referring to the security goals confidentiality, integrity and availability. Mapped on storage, these goals are concisely covered by the following definitions.

- Confidential data handling prohibits the unauthorized disclosure of any information. Data needs to be protected against internal as well as external attackers. Protection mechanisms furthermore cover the data while in transit. One technical measure to gain confidentiality is encrypting the data before sending it in the cloud.

- Integrity guards the status of the remote stored data against unauthorized or unintended modifications. Continuos bit-checking of remotely stored data needs to rely on fast and scalable mechanisms. Checksums are a straightforward approach to guard integrity. Measures guarding integrity offers even recovery possibilities by applying erasure codes.

- Availability guarantees the access to the data. Availability on the server-side is hard to be assured from a users' perspective. The status of the cloud as well as the connectivity stays out of focus of a common customer. Measures to increase availability include mirroring the data in multiple clouds as well as local caching.

This definition is extended in the literature by accountability and assurance [81, 107].

- Accountability defines the traceability of actions occurring on the data. In secure data storage, versioning is one way to achieve accountability. Atomic modifications result in disjoint versions making actions on the data traceable.

- Assurance summarizes the achievement of the other goals. It is therefore hard to achieve from a technical perspective. Assurance extends technical measures e.g. with policy and society-based attributes. Since these aspects are flexibly defined, a throughout satisfaction of assurance is hardly possible.

These security goals represent the basic design goals for secure cloud storage. The evaluation of current products and of current research approaches refers thereby to these design goals.

3.1.1 Combination of Security Measures?

Security is commonly achieved by only satisfying either confidentiality or integrity. Reliable, secure data storage must adhere to all the security requirements for gaining assurance. Stoneburner [107] recommends the awareness of all requirements when establishing security in a system. This idea is widely reflected in the literature [93, 103].

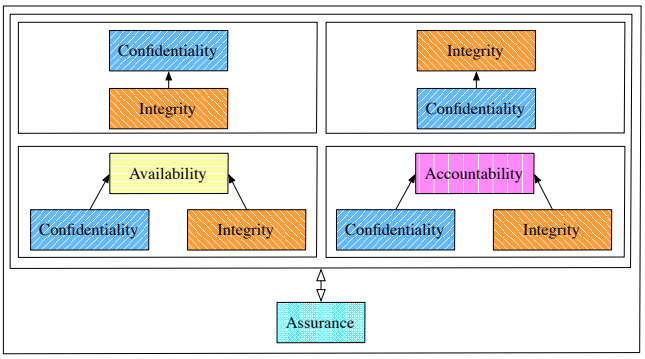


Figure 3.1: Dependency of Security Goals after Stoneburner[107]

According to Stoneburner, each of the security measures depends on each other as presented in Figure 3.1: Confidentiality without integrity cannot be guaranteed. The data might be modified in an unauthorized way e.g. by injecting false informations into the dataset. On the other hand, confidentiality itself protects integrity mechanisms on the data. In turn, availability and accountability both rely on confidentiality and integrity. Unauthorized access or modifications might result in inaccessible and unusable data. Assurance relies on confidentiality, integrity, availability and accountability representing additional, non-technical aspects. Examples for mechanisms guaranteeing assurance are policy-based accesses, legal regulations as well as social awareness.

Suitable techniques for the cloud must satisfy all security aims. These mechanisms must not only respect the remote location. The technical possibility for cloud storage scenarios must be considered as well.

3.2 Security and Cloud Storage

Any cloud service is categorized as "Software as a Service" (SaaS), "Platform as a Service" (PaaS) or "Infrastructure as a Service " (IaaS) [83, 87].

SaaS includes directly usable services for costumers in the cloud. These services are accessible with either defined protocols, mainly HTTP, or local applications. Most web applications are examples of SaaS e.g. Google Docs, Dropbox [5] as well as all directly accessible cloud storage like Amazon S3 (AWS S3) [3].

PaaS enables customers to deploy own applications using the cloud as platform for own services (e.g. complex web applications). Using pre-defined programming languages and documented APIs for the platform, developers upload their applications to PaaS servers. Examples of this service are Google App Engine [10] and Amazon Beanstalk [2].

IaaS provides costumers with fully or partly operational operating systems. This includes storage and applications as well as own deployed services in this system. One example is Amazon EC2 [1].

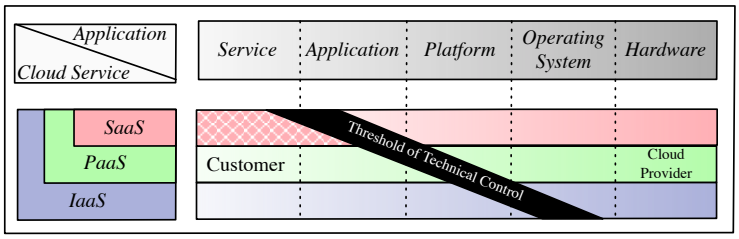


Figure 3.2: Definition of *Threshold of Technical Control*

Figure 3.2 maps these categories to the applicability of technical measures. Cloud providers and customers have different abilities to interact with the service. The classification leans on the *NIST guidelines on Security and Privacy in Public Cloud Computing* [71]. Each service maps to an item of the execution stack. The stack consists of services, applications, platforms, operating systems and hardware. The higher the offered service is, the more control over the application is lost to the cloud provider. The *Threshold of Technical Control* denotes this loss of control: IaaS for example delegates full control of all application layers above the operation system to the user. In contrast, the technical management of the operating system and the hardware is under the exclusive control of the cloud provider.

Cloud storage commonly fits the SaaS level. The range of services cover storage protocols like iSCSI [101], storage of files (as provided by Dropbox [5], Wuala [22], Google Drive [12], Microsoft Skydrive [14]), or professional No-SQL databases (like AWS S3 [3], Google Cloud Storage [11] or Microsoft Azure [15]). The stored units themselves are always built as resources accessed by REST [47]. Consequentially, the least common denominator of these storage types is a simple key/value representation. Key/Value stores can be seen as persisted tables representing a major part of the No-SQL community. Referenced by any key, which must be unique and not null, binary large objects (BLOBs) are persisted. The resulting tuple is called bucket in this thesis, derived from the description in multiple products. The handling of buckets in the cloud only covers storage and retrieval in this thesis. All further interaction with the data in the cloud must be seen as untrusted and exceeds furthermore the SaaS level. The practicability of the buckets is nevertheless given by the stacking of SaaS applications: One example is Dropbox, built entirely upon Amazon S3. By combining techniques to protect buckets, a morphing architecture is built, establishing security measures for multiple data types. The *Threshold of Technical Control* enforces thereby the measures to be established before pushing data in the cloud.

3.2.1 Security Challenges

The focus lies on cloud storage and therefore on SaaS only. Buckets are used only for storage and retrieval. Respecting the *Threshold of Technical Control*, protecting measures on the data must be established before the data is deployed in the cloud. Consequently, the hatched red part in Figure 3.2 represents the area of interest. The following measures protect data stored in the cloud [76].

Encryption and Cloud Storage Use Cases: Before uploading in the cloud, the data has to be encrypted to gain confidentiality. Encryption offered on the server-side or in closed source clients weakens confidential data handling. The user is entirely unaware of intermediate accesses. Therefore, the data has to be transparently encrypted before transfer. Applied encryption should not hamper synchronization or collaboration. Sophisticated methods to collaborate with encrypted data still patronize cloud storage use cases.

Tracing actions on remote data: Accountability is achieved easily by versioning the data. Although cloud storage providers offer versioning as a feature, this server-side feature might not be trusted respecting the *Threshold of Technical Control*. The *Threshold of Technical Control* motivates the protection of the data on the client. Furthermore, server-side versioning represents a provider-specific functionality. Using different clouds enforces independence from cloud storage providers. The versioning approach must be tailored to the remote location when applied by the client. Characteristics to be respected are bandwidth and the consumed storage.

Data-Independent Integrity Checks: Integrity is hard to achieve in the cloud. The distance to the cloud restricts instant checks of the integrity. Server-side checks do not satisfy the aim to secure the data appropriately. These operations are performed behind the *Threshold of Technical Control*. The client must nevertheless be able to localize errors. Structures like trees or directed acyclic graphs offer scalable fault finding operations. Equipping the uploaded data with erasure codes offers the ability to restore damaged data.

Independence from Cloud Storage Providers: Using a single cloud binds the availability of the data to the availability of the hosting cloud storage provider. Connection losses, temporal unavailability of the infrastructure and internal removals generate the need to buffer the data. Using a cloud-of-clouds or mirroring the data locally increases the availability of the data.

These challenges match the security goals defined in Section 3.1. Chapters 4-7 describe approaches to protect the data.

3.3 Related Work

The rise of cloud services in the last years generates common concerns about security. The related work is distinguished into existing products and solutions on the one hand and research approaches guarding security of cloud-stored data on the other hand. These techniques might be extended by specific additional related work in the particular chapters representing the measures developed in this thesis.

3.3.1 Existing Products

Due to the variety of different cloud storage, the analysis of existing products focuses on secure cloud storage only. The Fraunhofer Institute for Secure Information Technology (SIT) published a report in 2012 [34]. This report describes existing solutions for secure file-based cloud storage: Evaluating common cloud storage providers, the SIT identified five out of seven clients mirroring the data locally. Local stores ensure availability when the connectivity to the remote data is interrupted. Three out of these five clients offer server-side versioning as well: Dropbox [5], Wuala [22] and TeamDrive [20]. Not

yet available at the publication date of this report was Google Drive [12], offering local mirroring and versioning as well. No security measures are applied directly in Dropbox and Google Drive. Wuala and TeamDrive offer client-based encryption of the data. Integrity-guarding techniques like checksums are not available in any of the products evaluated. Neither offers any client the ability to reconstruct unavailable data. The applied security measures to gain confidentiality by Wuala and Teamdrive are furthermore vulnerable. Violating the Kerckhoff's Principle [75], the provided security measures are not transparent based on the closed architectures of these systems.

Additional security layers such as EncFS [7] and TrueCrypt [21] offer the possibility to gain confidentiality independent of the location of the data. EncFS represents a file system in user space. Using EncFs needs additional adaptions on the operating system. Boxcryptor [4] builds on top of EncFS. Representing a dedicated solution for cloud storage, the implementation is unfortunately closed source as well. Furthermore, Boxcryptor does not protect integrity and availability even though EncFS guards the data also against unauthorized modifications. Truecrypt generates an encrypted container. Depending on this container, small changes in single files might result in a complete rewrite-operation. The rewritten container needs to be transferred in the cloud afterwards.

SpiderOak [19] represents a classic SaaS client focussing on the satisfaction of all security requirements: Locally cached, all former versions are held. Remotely lost data can thereby be reconstructed. Furthermore, parts have already been released to the open source community. Spideroak has the ability to work with user-owned servers. Strong security measures become obsolete in this scenario. Further examples relying on own servers are Sparkleshare [18] and Owncloud [16]. Since these approaches run on own-hosted servers, no security measures need to established, although Owncloud optionally encrypts the data on the server.

Custom-built systems like Duplicity [6] enables skilled users to create encrypted and checksummed archives. These archives are directly storable on various remote stores including Amazon S3.

Lower-level data like blocks rely on dedicated storage protocols like e.g. iSCSI [101]. Suitable specific block stores in public clouds are rarely protected. The overlaying file systems are able to care at least about the confidentiality. Integrity as well as availability are achieved by additional techniques like summarizing multiple remote volumes with the help of RAIDs.

Table 3.1 presents an overview of the described products. Security in the context of cloud storage focus mainly on the protection of confidentiality. Existing products mostly rely on the file system to protect availability, integrity and accountability. Other clients store data on own and therefore trusted infrastructure instead of public clouds. This generates the need to host servers. All of these approaches either store files or work on REST. None of the clients satisfy lower-level storage protocols like iSCSI nor specific data like images or databases. The remote location is fixed in all clients. At the moment, no independent clients exist accessing multiple public clouds at one time. Challenges ongoing with mirroring data in multiple clouds thereby represent an active field of

Table 3.1: Products providing Secure Cloud Storage

Kind	Name	Link	Access	License	Backend	Versioning	Integrity Checks	Encryption
Third-Party	Dropbox	[5]	REST & File	Closed	proprietary	✓	⊠	⊠
	Wuala	[22]	REST & File	Closed	proprietary	✓	⊠	✓
	Teamdrive	[20]	File	Closed	proprietary & own	✓	⊠	✓
	Google Drive	[12]	REST & File	Closed	proprietary	✓	⊠	⊠
	Spideroak	[19]	File	Partly Open	proprietary	✓	✓	✓
	Sparkleshare	[18]	File	Open	proprietary & own	✓	⊠	⊠
	Owncloud	[16]	REST & File	Open	own	✓	⊠	(✓)
	Duplicity	[6]	File	Open	own	✓	✓	✓
Layer	EncFS	[7]	File	Open	Any	⊠	⊠	✓
	Truecrypt	[21]	File	Open	Any	⊠	⊠	✓
	Boxcryptor	[4]	File	Closed	Any	⊠	⊠	✓

research. Such challenges include the usage of quorum systems to identify invalid data.

3.3.2 Research Approaches

Secure cloud storage represents a hot area in research:

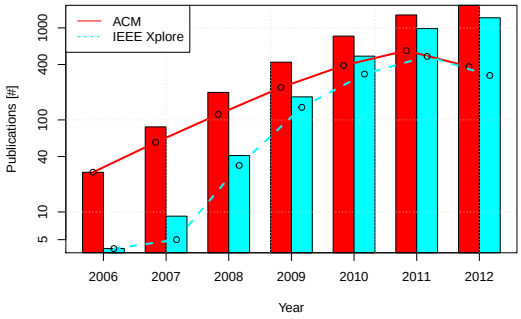


Figure 3.3: Number of Publications per Year covering "Cloud, Stor*, Sec*"

Figure 3.3 shows the number of publications containing the terms "cloud, stor*, sec*" in the title or the keywords in April 2013. The bars represent the cumulated amount of publications. The lines stand for the new publications per year. As denoted by Figure 3.3, the amount of publications increases dramatically in the last years. The focus is thereby laid on security of bucket-based cloud storage only. In this area, multiple surveys tailor the identified security goals in Section 3.1 to cloud storage [24, 66, 108, 110]. These surveys act as starting point for an overview over current research in this area.

3.3.2.1 Maintaining Accessibility

When advertising cloud storage, availability stands as major argument pushing data in the cloud. "Numbers of nine" denotes the decimals after 99%, representing the guaranteed availability by the cloud storage providers. The availability of the data is nevertheless dictated by these companies. Ways to overcome this dependability are local caching of the data and/or the usage of multiple clouds as proposed by Cachin et al. [41]. Similar to the idea of RAID, several approaches distribute the data in disjoint clouds [23, 40].

The "Proof of Retrievability" (POR) [36] or "Proof of Data Possession" (PDP) [27] generates knowledge about the integrity and thereby indirectly about the availability. These approaches focus on cloud storage only. Similar techniques also exist in the area of P2P storage [42]. Message Authentication Codes are computed using chunks of the data as described in detail in Section 3.3.2.2. Byzantine Quorum Systems [85] and striping enables the usage of multiple clouds [30, 35, 90]. Specific applications like database systems [25] use the resulting cloud-of-clouds approach to increase their availability.

Other approaches rely on local mirrors and focus on the consistent handling of the data. The data must be consistent in a collaborative environment combined with an untrusted cloud storage provider. These approaches [26, 51, 84, 106] use local mirrors of the data and versioning. The focus of these techniques are secure, concurrent, synchronized exchanges of new status over an untrusted cloud.

3.3.2.2 Integrity Checks on remote Data

Most approaches guarding availability care about integrity of the data as well: Inconsistent data automatically harms the access to its correct status as denoted by Figure 3.1.

Related to the cloud-of-clouds approaches, distributing the data needs robustness against Byzantine Errors. Data is therefore equipped with checksums and probes. The status of the data must be checked continuously. The remote location makes incessantly checks hard to perform. The POR [36] tackles this problem. A Message Authentication Code (MAC) combined with an Error Correction Code (ECC) is applied on the buckets. The MAC detects large errors and is relying on units in the buckets. The ECC protects the bucket against small errors. The number of units for the MAC and the size of the ECC gives an assumption about a possible successful retrieval. This assumption is provided as probability to successfully access data. This technique can be combined with a cloud-of-clouds approach [35]. PDPs [27] represent a similar approach offering a probability of possession using sampling. Recomputing the MACs as well as performing checks on data needs computing capabilities in the cloud. The usage of plain SaaS stores becomes therefore impossible.

Current approaches working with single clouds are used mainly for synchronization [51, 84]. Checksums are combined with encryption to guard data [119]. These approaches use the versioned data by generating a chain of hashes. Other approaches [114] combine sampling with erasure codes similar to the POR but are working on single clouds only. Focusing on concurrent access, some approaches [105] use optimistic, time-stamped

writes. Combinations of integrity checks with probabilistic tests on remote data result in higher-level architectures [73, 115]. Examples for these architectures are cloud-based file systems [74, 106, 111] or database systems [25]. These approaches use the idea of Merkle-Trees [88]. The folder structure leverages from the tree structure in combination with remote integrity checks. Optionally, the task of integrity checks can be delegated to untrusted cloud components [92]. In this scenario, encrypting and the computation of checksums are performed by different cloud services.

The usage of multiple clouds [23, 40, 90] needs sophisticated integrity checks guarding the data against single, faulty clouds. Mirrored data in a cloud-of-clouds [30] relies on Byzantine Quorum Systems [85]. By storing data in $n = 3f + 1$ clouds, f clouds failures can be compensated. Despite relying on remote computations, these approaches work with buckets only. Furthermore, no pre-processing such as sampling is necessary. All data receives a version number ensuring accountability additionally to the availability.

3.3.2.3 Guarding Confidentiality

Guarding confidentiality is performed straight-forward: Before uploaded in the cloud, the data should be encrypted. The applied encryption should be transparent according to the Kerckhoff's Principle [75].

One active area of confidentiality in the cloud covers remote operations on encrypted data. Homomorphic Encryption (HES) [54], as primary example, is not applicable in practice for larger queries [49, 91]. The complexity of de-/ and encryption as well as the size of the cipher text grow massively with the number of operations performed on the encrypted data. Instead, some approaches use searchable encryption approaches. This technique uses pre-defined queries representing views on dedicated buckets. Such a view represents an encrypted index structure [73, 74].

Most approaches guarding availability and integrity are not explicitly mentioning encryption. Nevertheless, these techniques can be directly extended [106, 111, 114, 119]. Approaches distributing the data in a cloud-of-clouds often distribute the key material as well [30, 40]. The distribution of key material relies on common secret sharing paradigms [104].

Focusing on collaborative use cases, the challenge is to provide a suitable key management. The key management should make use of the scalability and availability of the cloud [40]. Challenges are especially the key distribution and flexible access to versatile groups [26]. Consequently , these approaches should not only satisfy integrity and availability. Accountability must include adaptable access rights mapped to different versions.

3.3.2.4 Accountability

Several different models guarantee an accountable cloud service. Examples include logging and monitoring [120], establishing procedural approaches [96], combinations of sampling, replaying modifications and time-stamping [68] or establishing an entire life cycle using all of these measures [77].

Focusing on cloud storage only, modifications must be traceable by a user. Some approaches guard integrity by versioning hashes. These approaches care about accountability as well [51, 84]. Other approaches put the data directly under version control including adjacent metadata [30, 105, 106].

3.3.2.5 Combining Security Measures

Table 3.2 summarizes the described approaches.

Table 3.2: Research Approaches for Secure Cloud Storage

Name	Ref	Year	integrity	availability	confidentiality	accountability	Level	Cloud-Type	Main Contrib.
Bluesky	[111]	2012	✓		✓		SaaS	Single	Encrypted block stored in buckets equipped with checksums
μLibCloud	[90]	2012	✓	✓			SaaS	Multi	ECC-guarded writes on multiple clouds
Iris	[106]	2012	✓	✓	✓	✓	SaaS	Single	Block-based integrity using on tree-ordered MACs
Survey	[24]	2012					SaaS	Single + Multi	Evaluation of recent approaches
Survey	[34]	2012					SaaS		Evaluation of existing products
CloudProof	[26]	2011		✓	✓		PaaS	Single	Chained hashes over encrypted blocks
DIaas	[92]	2011	✓				PaaS	Single	Integrity-Checks performed on untrusted services
CS2	[74]	2011	✓		✓		PaaS	Single	PDP and queries over encrypted data
MCDB	[25]	2011	✓	✓			PaaS	Multi	Deploying DBMS in a cloud-of-clouds
DepSky	[30, 85]	2011	✓	✓	✓	✓	SaaS	Multi	Usage of multiple clouds including error detection and encryption
Survey	[98]	2011					IaaS		Possible attacks and evaluation of methods
Depot	[84]	2011	✓	✓		✓	SaaS	Single	Weak consistent access using versioned-chained hashes
RACS	[23]	2010	✓	✓			SaaS	Multi	RAID-alike storing of data in multiple clouds
ICStore	[40]	2010	✓	✓	✓		SaaS	Multi	RAID-alike storing of data in multiple clouds
TrustStore	[119]	2010	✓		✓		SaaS + Server	Single	File-to-bucket mapping including checksums and encryption
SPROC	[51]	2010	✓	✓		✓	SaaS + Server	Single	Weak consistent access using versioned-chained hashes
	[73]	2010	✓		✓			Single	High-level architecture
SecCloud	[115]	2010	✓				SaaS + Server	Single	Merkle-Tree using integrity checks
Survey	[110]	2010					Multiple		Access rights in heterogeneous environments
Venus	[105]	2010	✓			✓	SaaS + Server	Single	Optimistic writes offering multi-user access
Survey	[108]	2010					Multiple		Flaws and possible attacks
	[49]	2010			✓		SaaS	Multi	Evaluation of FHE
	[114]	2009	✓		✓		SaaS + Server	Single	Sampling & ECCs
HAIL	[35, 36]	2009	✓	✓			SaaS + Server	Multi	PORs & striping
Survey	[41]	2009					SaaS	Multi	Combination of multiple techniques
PDP	[27]	2007	✓				SaaS + Server	Single	Sampling & MACs

Each approach is mapped to the security aims described in Section 3.1. Most approaches satisfy more than one security aim. They focus on concrete usages of the cloud like collaboration.

The level of usage denotes the applicability of an approach. Some of the approaches

are in need of computation power in the cloud to perform remote actions on the data. No approaches relying on PaaS levels are applicable for plain cloud storage only. These approaches are in need of remote computational resources. Computational handling of data in the cloud must be seen as untrusted and thereby stays out of focus of this thesis.

Another difference can be made by using multiple or single clouds. Approaches relying on single clouds focus mostly on the synchronization between multiple clients. All participating parties must establish security.

Multi-cloud approaches aim to secure the data itself. Their goal is to make it impossible for the provider to analyze the data or to reduce its availability. Consequently, in addition to availability, remote integrity checks become necessary.

The techniques developed in this thesis enable collaborative work by not trusting the storing cloud storage provider. The main idea is not to reinvent the wheel. Instead, building a morphing architecture on top of a cloud-of-clouds approach extends the existing approaches. Relying on buckets only, the morphing data is equipped with security measures locally and stored in cloud-based No-SQL structures afterwards. The applied security techniques support remote checks and fill some of the gaps in the pre-existing work.

- Accountability describes the need to track changes on data. Re-hashing old status with current modifications [51, 84] or putting the data under version control [30, 105, 106] satisfies accountability. Versioning offers the ability to replay actions on the data. Since cloud storage is billed partly based on the amount of space consumed, old versions should be removed from the system. The removal needs normally a modification of the versions relying on each other. Rewriting results in additional cost. This thesis introduces an adaptive versioning. The versioning offers easy removal of older versions without the need to rewrite. Furthermore, the versioning is extremely robust against the loss of single versions and described, analyzed and evaluated in Chapter 4.

- To guard integrity, buckets containing the data are structured in a directed acyclic graph (DAG) inspired by ZFS [33]. The resulting graph is used as a Merkle-Tree [88] similar to current approaches [25, 74, 106, 111]. Every bucket becomes immutable immediately after its creation, resulting in an append-only data store. Relying on the flexible versioning, old buckets may be nevertheless removed optionally. This DAG supports applied remote integrity checks like PDPs[27] and PORs[36] and is described in detail in Chapter 5.

- The denoted bucket layer uses Blobs. Storing arbitrary data, it offers the possibility to map different data types. Uniform security measures become thereby applicable for different data ranging from low-level data like blocks to high-level resources accessible by REST. Furthermore, the buckets are serialized through an interface [13]. This interface enables variable distribution mechanisms [30] providing availability. A detailed mapping of data to different storage backends is described in Chapter 6.

- Key management has some interesting twists when applied to a versioned storage system. This thesis analyzes these options and introduces new insights and simplifications in Chapter 7. The actual encryption of the buckets is not new.

When it comes to privacy and accountability, people always demand the former for themselves and the latter for everyone else.

David Brin - American Author and Scientist / 1950 -

4 Adaptive Versioning

Contents

The tension between privacy and accountability is well covered in the quote by David Brin. On the one hand, unexpected and questionable events should be reproducible and attributable. On the other hand, personal actions should not become public. Versioning data commonly protects accountability and is thereby essential in modern storage systems. Redundant data covering current and former versions of the data does not only increase robustness against data loss, it also provides the ability to play back operations occurring on the data tracing failures in the storage. Applied to cloud-stored data, the following problems need to be solved.

1. Scrubbing for getting knowledge about errors is expensive when applied remotely.
2. Checking data by current versioning approaches results in peak loads in transfer and storage.
3. Different access patterns might result in fragmentation of the data highly influencing the number of change sets stored and accessed.
4. Unlimited change sets are expensive when stored in the cloud.

The proposed sliding versioning represents a simple yet powerful solution for these challenges by striping unmodified content over a fixed number of change sets. Storing redundant data piggyback to modifications, checking the current status bases on a fixed number of change sets. Versioned data is evened out enabling constant data in transfer and storage.

The size of change sets a version consists of, denoted as sliding window, is defined as a tunable and intuitive parameter. Balancing even random accesses, the sliding window offers flexible choices between more data stored in change sets and more change sets accessed for reconstruction

Actions on data should become traceable, repeatable and revokable. To protect privacy, additional security measures must interact with the versioning. A versioning-aware key management as one example for such an interaction is later presented in Chapter 7.

4.1 Terminology

The nature of versioning is quite abstract. For a better understanding, common terms are defined to describe and evaluate the presented approach, called sliding versioning. Figure 4.1 shows the different components.

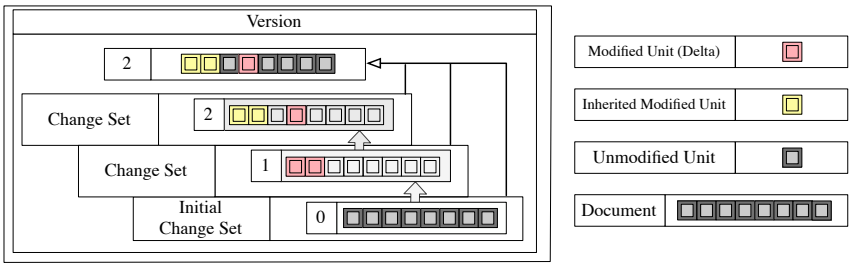


Figure 4.1: Terminology of Versioning Components

A document denotes the complete status of any system to be versioned. Documents can be any data divisible in elements. Single elements are called units. A document consists of multiple units. The actions occurring on the units are modifications, deletions and insertions. Actions applied on a unit marks the unit as modified representing a delta. Deltas are marked as red units in Figure 4.1. A set of deltas on multiple units ends up in a change set. Deltas can be inherited between change sets depending on the versioning approach. Additionally, change set holds references to the last related change set for reconstruction. In the example of Figure 4.1, the first two units are modified in change set 1. Change set 2 consists of these inherited modified units represented by yellow elements. The fourth unit is modified in change set 2. Multiple change set represents a version. A version contains all preceding change sets. In the example, version 2 contains the initial change set and the modifications resulting in the change sets 1 and 2. Versions

thereby offer a view to the full representation of a document. All appeared change sets are included in a version.

4.2 Background

Even though versioning naturally satisfies the need of accountability in storage scenarios, the following attributes must be respected for its usage in the cloud.

1. The access to the data is limited by the bandwidth of the internet. This bottleneck to the storage must thereby be partially offset by efficient read and write accesses.
2. The accountability directly relies on the robustness of the server-stored data against unexpected modifications. The loss of a single change set should therefore not result in unusable version.
3. Cloud storage is expensive. Billed on the storage consumed, pay-as-you-go must be taken into account. One possibility is to offer the removal of unnecessary versions.

4.2.1 Existing Approaches

Some cloud storage providers as well as cloud storage clients already offer versioning of the data. Nevertheless, the applied versioning approaches are, with respect to pay-as-you-go billing, not optimal regarding data transfer and storage consumption. Even though, cloud storage providers like Amazon offer server side versioning. Entire buckets are versioned as fresh copies called full versions in the rest of this thesis. Consequently, all modified data must be entirely uploaded, as the entire bucket is billed by the numbers of versions representing.

On the client side, modified data is mostly handled as full versions as well. Dropbox, as one example, treats files irrespective of their specific type always as binary. This results in the fresh upload of modified files even if only minor changes occurred.

Open source alternatives partly make use of existing versioning systems. For example Sparkleshare [18] uses git [9] as backend and transfers only change sets. This behavior reduces the number of transferred bytes drastically. The removal of persisted change sets, in order to reduce consumed storage, becomes complicated. Either this removal is impossible or it results in computational expensive, time-consuming rewrite operations of the entire repository. Furthermore, the remote repository has a fragile structure. The loss of mandatory change sets might result in an unusable remote storage.

4.2.2 Description and Mapping to Cloud Infrastructures

Handling versions must be aware of deltas. Deltas might be applied on all units of the entire document simultaneously. One example is the initial change set for inserting documents in a versioning system. Such a mirroring of all units independent of occurring modifications is called full version. The handling of consecutive versions as full versions, shown in Figure 4.2, is straight-forward. Figure 4.2 shows deltas written by a sequential

modification pattern. Additionally to the deltas, all units are written in each change set including modified, inherited modified and unmodified units. A version is reconstructed by access one single change set only. The price for the instant access is the consumed space. The storage is consumed linear to the number of versions instead of the number of deltas.

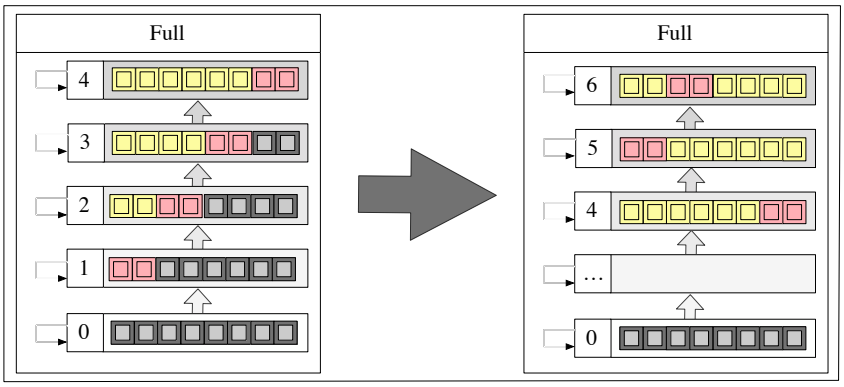


Figure 4.2: Writing data by Full Versions

Figure 4.3 represents versioning writing deltas only. These approaches consume space proportional to the modifications only. Common representatives are the incremental and differential versioning.

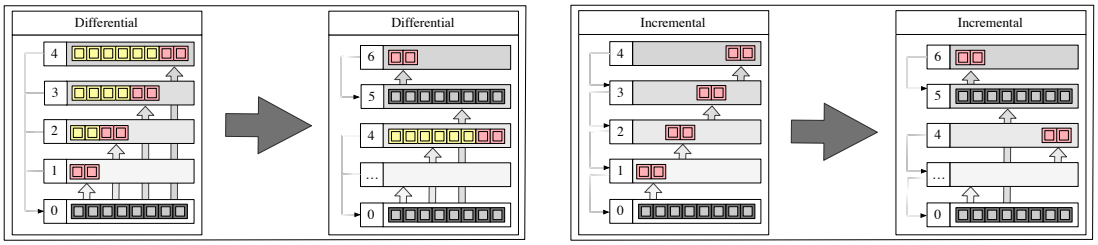
Differential handling of deltas assumes that all modifications rely on the initial change set. The left side of Figure 4.3a represents four consecutive change sets including the initial change set. Units in the document are sequentially modified. Differential versioning puts inherited modifications to each change set.

Relying on the last full version for all succeeding versions, differential versioning reconstructs a version in an optimal way: Reconstruction is only in need of the related initial change set and the change set representing the current version. The obvious drawback is the redundant copy of inherited modifications represented by the yellow units in Figure 4.3a. The generation of differential versioning-change set might lead to the generation of full versions represented by version 4 in this example. To overcome this degeneration of change sets, full versions are created. The write of such a full version results in less redundant units to be written in the following change sets. On the right side of Figure 4.3a, version 6 relies on the full version of version 5. As a consequence, less units are written compared to versions 2 to 4.

Differential versioned units in the cloud enable the client to access a version by using two change sets. Since each change set might be representing a bucket, differential versioning allows the retrieval of a version with the help of two requests. Requests, additionally to the traffic, are included in some billing models. The access to a version is therefore cheap. The traffic generated for differential writes is, however, hardly predictable: Depending on the amount of modified and inherited modified units, full versions are created in the worst case. Full versions, to overcome the degeneration of

differential versioning, result in the necessity to transfer an entire document. Such a transfer creates even more requests and consums even more bandwidth.

Furthermore, robustness relies on the full versions. If this change set is damaged or lost, all further versions might be broken as well.



(a) Differential Versioning **(b)** Incremental Versioning

Figure 4.3: Writing Data by conventional Versioning

Incremental versioning, on the other hand, represents the counterpart. Figure 4.3b shows the sequential modification pattern using incremental versioning. Versions written by the incremental approach only rely on the previous change set, minimizing the amount of data written: Only the modified units are written. Inherited modified units are not included. In contrast to the differential versioning, incremental versioning requires all former change sets to access a version. This effort is shown as black arrows in Figure 4.3b. Accessing version 4 requires the access to the related five change sets. To reduce the need to retrieve a growing number of change sets for accessing a version, incremental versioning approaches inject full versions. In Figure 4.3b, all change sets up to version 4 are needed for reconstruction. After injecting a full version representing version 5, reconstructing version 6 only needs two change sets.

Incremental versioning minimizes the amount of write operations: Only the deltas are transferred. However, the reconstruction of a version can get expensive. Reconstructing a version requires all previous change sets up to the most recent full version. The number of necessary change sets included in each version grows, originating from the last full version. The dependency of a version to its change sets makes incremental versioning also vulnerable to the loss of single change sets. Since the access to a version needs all related change sets, the loss of one change set results at worst in the loss of all versions containing this change set. Additional written full versions increase the robustness and the retrieval effort but consume more traffic, requests and storage.

Both versioning approaches need to inject full versions. Full versions are injected to overcome the necessity to retrieve a large number of change sets for reconstruction. Differential versioning needs intermediate full versions to moderate the write of former modifications in addition to the current deltas.

The dependency of versions on its change sets decreases the robustness. Combined with unpredictable reads and writes, it is questionable that current versioning approaches act as optimal solution for robust, remote-stored data.

4.2.3 Contribution: Evaluation of the Sliding Versioning

A new versioning technique, named sliding versioning, was invented by Marc Kramis in 2009 [78]. This versioning approach was presented only in a descriptive way, focusing on a patent application, without deep regard on implementability or performance.

Mapped to the needs of secure cloud storages, the sliding versioning generates the following benefits.

1. The distance to the cloud needs robust and efficient uploading and downloading operations. Sliding versioning writes not only deltas. It also trails full versions over a fixed number of versions. Requests as well as traffic is thereby reduced compared to intermediate full version injections.
   → Sliding versioning is compared to incremental versioning and differential versioning. The focus of this evaluation lies on the amount of units to be transferred. This part of the contribution is given by Sections 4.3 and 4.4.

2. The stored data must be robust against the damage of single change sets. Sliding versioning trails full versions over several change sets. A high robustness against the inaccessibility of single change sets is given.
   → The gained robustness of sliding versioning is described and evaluated. The loss of single change sets is compensated with less impact to related versions. This part of the contribution is given by Section 4.5.

4.3 Sliding Versioning

The main idea of sliding versioning is to trail a full version over multiple change sets. The approach shifts units manually to the current change set if necessary. Sliding versioning initializes therefore a new change set with units not recently written. A window parameter called sliding window defines a fixed number of change sets in which an unit must be written. Consequently, the sliding window denotes the number of change sets a full version is distributed on.

Sliding versioning relies thereby on a fixed amount of change sets to access a version. Only a minimal amount of units is written in each change set additionally to the occurring deltas.

Algorithm 1 shows the reconstruction of a version. The algorithm furthermore creates a fresh change set for upcoming modifications (represented by *currentChangeset*). *version* contains the entire document while *currentChangeset* denotes the start point for new deltas. *inputChangesets* is of size of the sliding window. The reconstruction of the version uses the first occurrence of an unit in *inputChangesets*, leaned on the incremental versioning. Sliding versioning furthermore writes the units of the last change set manually to *currentChangeset*. As a result, a version stays always accessible by accessing at most sliding window-change sets. *currentChangeset* as a result contains all units not modified in the last change sets.

Algorithm 1: Sliding Versioning

```
1  /* Former change sets sorted from newest to oldest of size Sliding
       Window                                                          */
   Input: Change Set[] inputChangesets
2  /* Reconstructed current version                                   */
   Output: Version version
3  /* Change set representing upcoming modifications                  */
4  currentChangeset ← ∅;
5  /* Iterating through all units of a doc...                         */
6  foreach unit ∈ units do
7      /* ...and iterate through all formers change sets.             */
8      foreach changeset ∈ inputChangesets do
9          /* For each unit and each change set, set unit if present and
               not yet set.                                            */
10         if unit ∈ changeset ∧ unit ∉ version then
11             set(unit → version);
12             if changeset = inputChangesets.last then
13                 set(unit → currentChangeset);
14 set(currentChangeset → version);
15 return version
```

Figure 4.4 represents the sliding versioning in practice. The sequential modification pattern from Figures 4.2 and 4.3 is used. Deltas are written adjacent to the incremental versioning, represented by the red units. If not touched in sliding window-versions, sliding versioning writes all not-yet modified units manually. The resulting trailed full version is shown as blue units in Figure 4.4 using a sliding window of size 2. Over time, sliding versioning stripes the trailed full version over multiple change sets when writing data sequentially. The result is a balanced load in the change sets shown by the right side of Figure 4.4. The write effort is constant in change sets 4 and 6. As a result, there is no need to write full versions to prohibit a degeneration of the change sets.

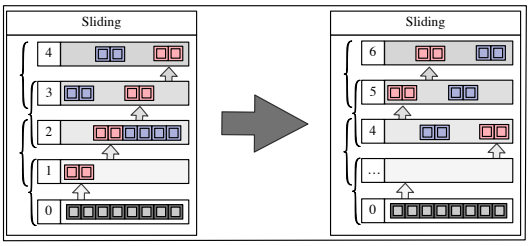


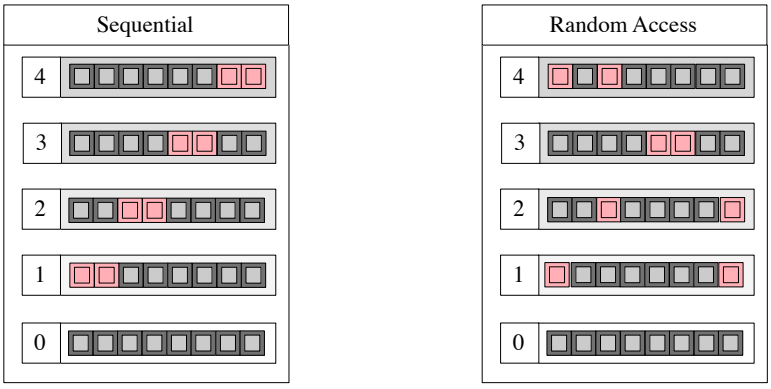
Figure 4.4: Writing Data by Sliding Versioning

Losing a change set does not result in the loss of multiple versions. The robustness argues for an application to cloud storages. If a change set is lost, other adjacent versions are not necessarily harmed. The size of the sliding window defines the amount of protected versions. Additionally, sliding versioning offers predictable loads of the change sets. The price is the transfer of more units even if a manual injection of full versions is not necessary. A fixed number of change sets is sufficient for reconstruction shown by the following analysis.

4.4 Theoretical Analysis

A theoretical analysis compares the sliding versioning with the incremental versioning and the differential versioning. This analysis evaluates the percentage of units in change sets, representing the write effort. Additionally, the percentage, that an unit becomes accessible by accessing a fixed number of change sets, is evaluated. This percentage represents the read effort. Starting from a modification rate $m \mid 1 \geq m > 0$, each versioning approach writes units in change sets named $v \mid v \in \mathbb{N}_0$. For the sake of simplicity, it is assumed that m represents the same modification rate in all change sets.

4.4.0.1 Access Patterns



(a) Sequential Access **(b)** Random Access

Figure 4.5: Definition of different Access Patterns

Figure 4.5 represents a modification rate of $m = 0.25$. Sequential access and random access define two different access patterns. Figure 4.5a denotes the sequential access pattern. Each unit is touched only once before all other units in the same document are touched. The order of occurrence is irrelevant for the upcoming analysis. It is only important that all units are accessed before any unit is touched twice.

Taking v change sets into account, sequential access touches m units in each version. Equation 4.2 represents the probability to access any unit:

$$a(v) = m * v \tag{4.1}$$

$$t_{seq}(v) = \begin{cases} a(v) & a(v) < 1 \\ 1 & \text{otherwise} \end{cases} \tag{4.2}$$

The sequential access scales linear to m. As a result, the probability to access an unit is m multiplied by the number of versions accessed (v) as shown in Eq. 4.1. The maximum is restricted by 1 resulting in Eq. 4.2.

Random access is defined as touches of units independent from former deltas. This results in possible multiple touches of already accessed units. Examples are the first, the third and the last unit accessed in Figure 4.5b. As a result, not all units are necessarily touched like the second, the fourth and the seventh unit. A probability defines the access on any unit. $1 - m$ defines the probability not to access an unit in a change set. $(1 - m)^v$ defines randomly chosen units not touched in the last v change sets. The reverse probability denotes the probability of a random access on any unit in multiple change sets. Equation 4.3 represents this probability:

$$t_{ran}(v) = 1 - (1 - m)^v \tag{4.3}$$

Mixing up the access patterns blurs the border between sequential and random access. Equations 4.2 and 4.3 are interpreted as two functions. Combining these functions results in a family of curves defining the mixed access pattern. $r \mid 1 \geq r \geq 0$ implies the amount of "randomness" in an access. The resulting access pattern represents a mixture of sequential and random access denoted by Equation 4.4:

$$t_{all}(v) = t_{seq}(v) - r * (t_{seq}(v) - t_{ran}(v)) \tag{4.4}$$

Taking the sequential access as a baseline, the randomized part results in a subtraction of values relying on the randomness of the access. For $r = 0$, $t_{all}(v)$ denotes the access probability for sequential access. If all units are randomly accessed as defined by $r = 1$, $t_{all}(v)$ and $t_{ran}(v)$ generate the same probabilities.

4.4.0.2 Hotspot Analysis

In t_{all}, m applies on all units equally. In some scenarios, some units are accessed in each version by purpose. Hotspots represent these units. Such units rely on no weighted probability. Weighted probabilities do not influence the overall access rate. m is fixed in each change set, independent of any weighted randomness among the units. Additional access patterns satisfying such weighted randomness stay out of focus of this thesis.

The hotspot analysis only focuses on a subset of units accessed in all change sets with a probability of 100%. These units are represented by $f \mid 0 \leq f \leq m$.

Figure 4.6 shows again an access rate of $m = 0.25$. A fixed amount $f = 0.125$ is accessed in all change sets represented by the first, partly-green unit. The rest of the access occurs sequentially.

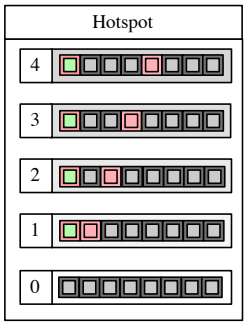


Figure 4.6: Definition of Hotspots

The probability of an access is now relying on a combination of m and f. The units always accessed must be excluded from the units possibly accessed. As a result, the exclusion of the hotspot results in m^* represented by Equation 4.5:

$$m^* = \frac{m - f}{1 - f} \tag{4.5}$$

The modification rate is adapted by f and afterwards normalized. f is applicable in Equations 4.2 and 4.3 by replacing m with m^*. An resulting adaption of Equation 4.4 is shown in Equation 4.6.

$$t^*_{all}(v) = (t_{seq}(v) - r * (t_{seq}(v) - t_{ran}(v))) * (1 - f) + f \tag{4.6}$$

The access rate $t^*_{all}(v)$ represents the starting point of the following analysis. The following parameters influence the access in a version.

- $m \mid 1 \geq m > 0$ implies the relative amount of deltas.
- $v \mid v \in \mathbb{N}_0$ denotes the number of change sets.
- $r \mid 1 \geq r \geq 0$ is the relative amount of randomness in the access patterns.
- $f \mid m \geq f \geq 0$ represents the amount of hotspot units.

4.4.1 Analysis of Writes

Conventional versioning relies on either incremental versioning or differential versioning. Incremental versioning only writes deltas, being optimal with respect to the amount of data written. The analysis of writes focuses thereby on differential versioning only. The write effort of the differential versioning is defined by the written units per change set, denoted as $w_{conv}(v)$. $w_{conv}(v)$ represents a mapping of $t^*_{all}(v)$ to writes: Every access on an unit implies a modification of this unit. A distinction of cases must be applied since a full version is always written as initial change set. The resulting write effort is shown in Equation 4.7.

$$w_{conv}(v) = \begin{cases} 1 & v = 0 \\ t^*_{all}(v) & v \geq 1 \end{cases} \tag{4.7}$$

Figure 4.7 plots $w_{conv}(v)$ with different values for $m = [0.01, 0.1, 0.5]$, $r = [0, 0.5, 1]$ and $f = [0, 0.5 * m, m]$.

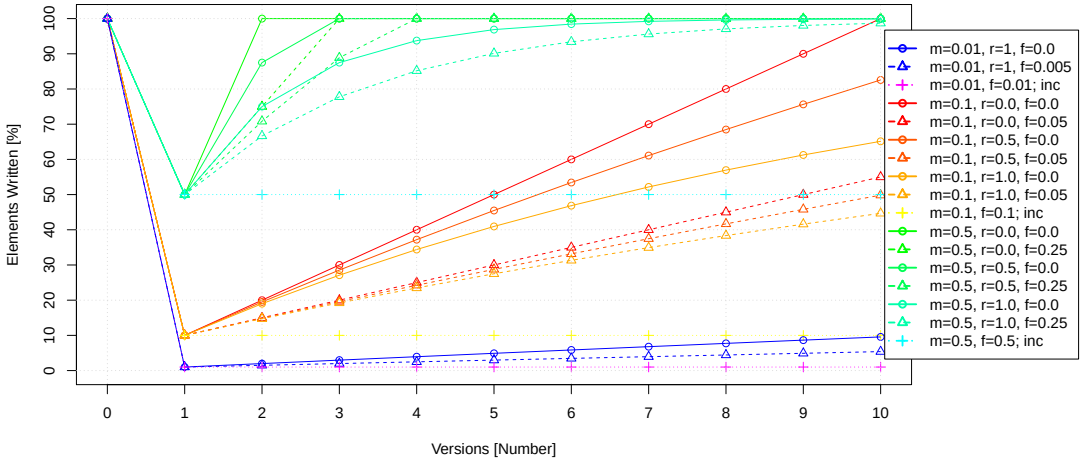


Figure 4.7: Write Effort for different Versioning Approaches

The y-axis represents $w_{conv}(v)$, the x-axis represents v. $w_{conv}(v)$ increases in every change set. The effort contains the current modified units and the inherited modified units.

m has the highest impact on the amount of data written.

Adding randomness by increasing r flattens the curve. The lower the randomness is $(r \to 0)$, the straighter the curves are. Random modifications therefore never guarantee the writing of all units. Some units might never be touched even in an infinite number of change sets.

f forces the same units to be written in each change set. The parameter f thereby flattens the curve. $f = m$ means, that all modifications take place in the hotspot. In this scenario, the curve is equal to incremental versioning since the differential versioning always modifies the same units only. The constants $y = [0.01, 0.1, 0.5]$ in Figure 4.7 denote the writings of hotspots only.

The curves relying on Equation 4.7 and shown in Figure 4.7 allows the direct comparison against the sliding versioning using different values for m, f and r.

4.4.1.1 Sliding Writes

The sliding versioning behaves similar to the incremental versioning. Additionally, sliding versioning includes not recently modified units. The sliding window, as additional

parameter $s \mid s > 1$, defines "recently". This versioning approach inserts units in the current change set which are not included in the last s change sets. The number of units not modified by s change sets is retrievable from Equation 4.7: The case $s = v$ represents the units written in s versions. The reverse probability $1 - w_{conv}(s)$ defines the units not written in s versions. This amount is trailed over s versions. Equation 4.8 represents the resulting write effort of the sliding versioning for v versions.

$$w_{slid}(v) = m + \underbrace{\frac{1 - w_{conv}(s)}{s}}_{o_{slid}(s)} \tag{4.8}$$

m defines the baseline of the workload. A constant number of units is written additionally to the deltas. This constant overhead reflects the not-recently modified units. These units are manually transferred to the current change set.

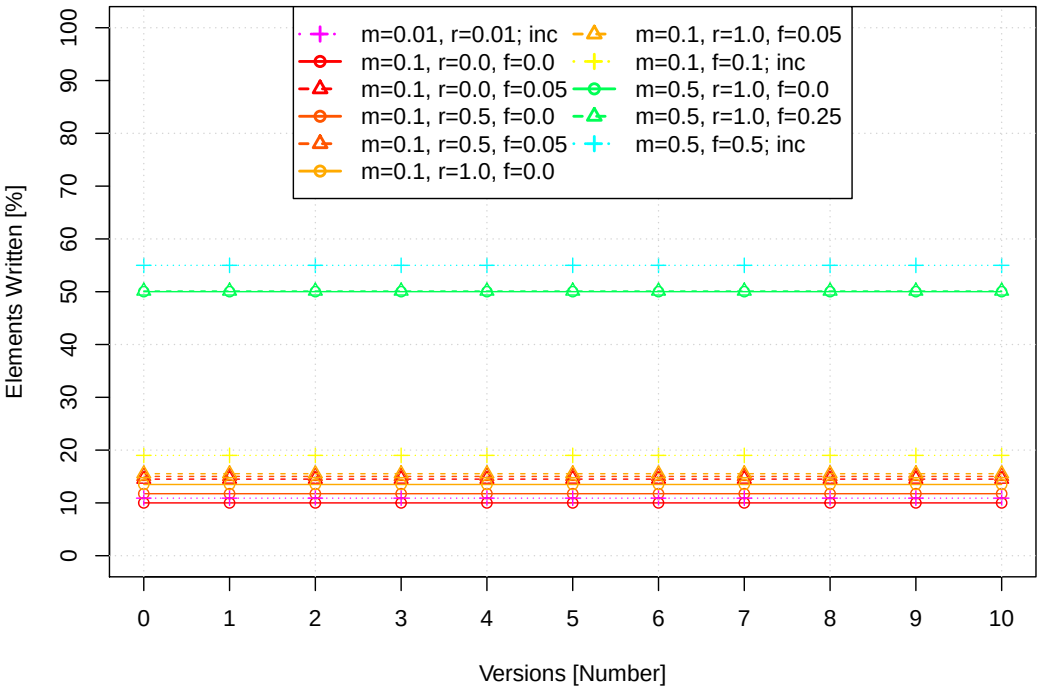


Figure 4.8: Write Effort for Sliding Versioning with a Sliding Window of $s = 10$

The write effort scales constantly in opposite to differential versioning. Figure 4.8 shows different parameters for $m = [0.01, 0.1, 0.5]$, $r = [0, 0.5, 1]$ and $f = [0, 0.5 * m, m]$. The sliding window is set to $s = 10$.

High modification rates make the overhead of the sliding versioning obsolete. $m =$

$0.5, f = [0, 0.25]$ scales similar to the work effort of incremental versioning. Because of the high modification rate, all units are written in at least two change sets. For $m = 0.5, f = 0.5$, the manual transfer needs to cover $1 - m$ units trailed over 10 change sets. The overhead constitutes of $o_{slid}(v) = 0.05$ per change set.

The family of curves relying on $m = 0.1$ scales around $w_{slid}(v) = 0.1$. The impact of r and $f = [0, 0.05]$ is negligible. For $m = 0.1, f = 0.1$, the overhead becomes visible. $o_{slid}(s)$ needs to cover 90% of the units for these parameters. The writing of the hotspot results in $w_{slid}(v) = 0.19$. Lower modification rates like $m = 0.01$ make f and r irrelevant. $o_{slid}(v)$ is in these cases significantly higher than m. $s = 10$ defines a similar scaling as applying $m \approx 0.1$.

Equations 4.7 and 4.8 allow a comparison of the write efforts between sliding versioning and differential versioning.

4.4.1.2 Comparing the Write Efforts

$w_{slid}(v)$ and $w_{conv}(v)$ allows the computation of a summarized work effort for $s = v$ change sets. The comparison relies on Riemann Integrals. Equation 4.9 represents the integral:

$$W(v) = \sum_{j=0}^{v} w(j) \tag{4.9}$$

Incremental versioning is excluded since its write effort is always optimal. By focusing on v change sets, the integrals allow the comparison of differential versioning and sliding versioning. $W(v)$ is resolvable as geometric series for both versioning approaches.

Evaluating differential versioning, the distinction of cases from Equation 4.7 must be respected when resolving the sum formula. Equation 4.10 represents the cumulated effort $W_{conv}(v)$ for differential versioning:

$$W_{conv}(v) = \quad 1 + \textstyle\sum_{j=1}^{v} t^*_{all}(v) \tag{4.10}$$

An adjacent series is computed respecting Equation 4.8 for sliding versioning. The resulting sum $W_{slid}(v)$ is represented by Equation 4.11:

$$W_{slid}(v) = \quad \textstyle\sum_{j=0}^{v} w_{slid}(j) \tag{4.11}$$

The difference between $W_{slid}(v)$ and $W_{conv}(v)$ compares the write effort with the help of $W_{diff}(v)$. Equation 4.12 shows the result of the subtraction:

$$W_{diff}(v) = \quad W_{conv}(v) - W_{slid}(v) \tag{4.12}$$

$$W_{norm}(v) = \quad \frac{W_{diff}}{W_{conv}(v) + W_{slid}(v)} \tag{4.13}$$

Any value of $W_{norm}(v) > 0$ points to less units written by the sliding versioning. The figures from Figures 4.9 to 4.17 show different values for $W_{norm}(v)$ depending on different parameters. Incrementing values for v represent the x-axis. The y-axis stands for the different modification rates ranging from $m = [0.0001, \ldots, 0.0991]$. The contour lines map different values for $W_{norm}(v)$. Values for $s = [5, 10, 50]$ and $r = [0, 0.5, 1.0]$ result in different figures. f is neglected since this parameter is not having any impact: If the hotspot covers all modified elements ($f = m$), a constant scaling is never achievable by the sliding versioning. For $f < m$, the scaling only influences the gradient, becoming irrelevant for small values of $m \to 0$.

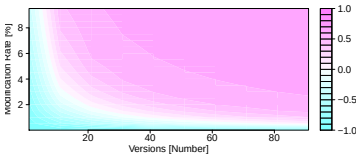


Figure 4.9: Write Effort of Sliding Versioning and Differential Versioning with $s = 5; r = 0.0$

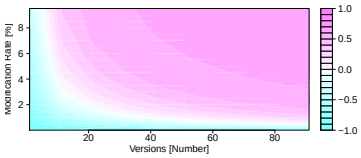


Figure 4.10: Write Effort of Sliding Versioning and Differential Versioning with $s = 5; r = 0.5$

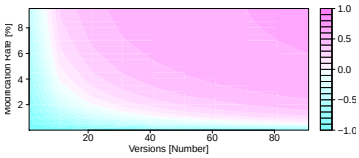


Figure 4.11: Write Effort of Sliding Versioning and Differential Versioning with $s = 5; r = 1.0$

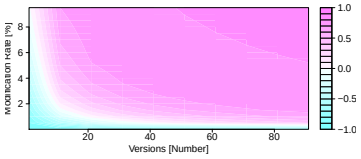


Figure 4.12: Write Effort of Sliding Versioning and Differential Versioning with $s = 10; r = 0.0$

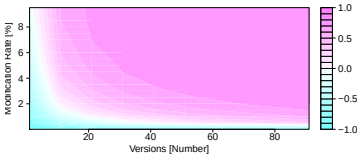


Figure 4.13: Write Effort of Sliding Versioning and Differential Versioning with $s = 10; r = 0.5$

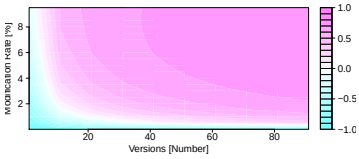


Figure 4.14: Write Effort of Sliding Versioning and Differential Versioning with $s = 10; r = 1.0$

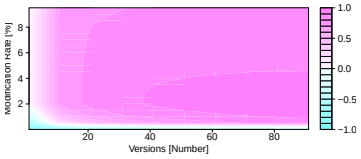


Figure 4.15: Write Effort of Sliding Versioning and Differential Versioning with $s = 50; r = 0.0$

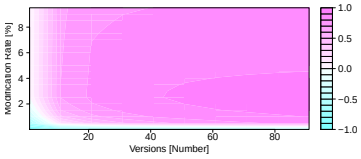


Figure 4.16: Write Effort of Sliding Versioning and Differential Versioning with $s = 50; r = 0.5$

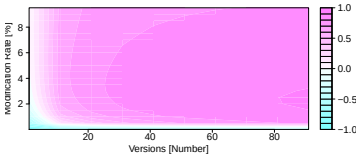


Figure 4.17: Write Effort of Sliding Versioning and Differential Versioning with $s = 50; r = 1.0$

m influences the performance of sliding versioning and differential versioning most. For large values of m, sliding versioning behaves similar to incremental versioning outperforming differential versioning. The difference between differential versioning and

sliding versioning is not linear and relies on s and r.

For small values for m, sliding versioning performs worse than differential versioning since the overhead to be written comes at a cost. The scaling improves for larger values for s. Nevertheless, differential versioning always writes less units than sliding versioning. Equations 4.14, 4.15 and 4.16 show the write effort for $m \to 0$:

$$\lim_{m \to 0} W_{conv}(v) = \quad 1 \tag{4.14}$$

$$\lim_{m \to 0} W_{slid}(v) = \quad \frac{v}{s} \tag{4.15}$$

$$\lim_{m \to 0} W_{diff}(v) = \quad 1 - \frac{v}{s} \tag{4.16}$$

$$\lim_{m \to 0} s = \quad v \tag{4.17}$$

The size of the sliding window has to be extended to cover v change sets to guarantee a performance as good as differential versioning, denoted by Equation 4.17. According to Equation 4.14, the differential versioning converges to 1. Such a convergence is only achievable by the sliding versioning with a sliding window as huge as possible.

The constant write effort of the sliding versioning needs additional units, transferred and stored. Despite traditional versioning approaches, no intermediate full versions are needed. That makes the size of the change sets more predictable. For lower modification rates, the sliding versioning performs worse. Countermeasures are a combination of versioning approaches or a choice for a larger sliding window. The sliding versioning furthermore offers a guaranteed number of change sets a version consists of.

4.4.2 Analysis of Reads

The probability to access any delta in $V \in v$ change sets denotes the read effort. Differential versioning needs at most two change sets for reconstruction. The focus of the analysis of the read effort lies therefore on the incremental versioning. Units are only written as deltas. The evaluation of reads is leaned on the access probability defined by Equation 4.6. The probability to retrieve an unit written by incremental versioning relies on $t^*_{all}(v)$. The resulting read effort is defined in Equation 4.18:

$$p_{conv}(v) = \begin{cases} 1 & v = V \\ t^*_{all}(v) & v < V \end{cases} \tag{4.18}$$

The access on a delta is influenced by the concrete units modified. In the worst case, the delta is located in the most recent initial change set. The worst case is thereby defined by accessing V change sets.

Figure 4.18 shows the distribution of the read probabilities. The initial change set is defined as $V = 10$. At most 10 change sets must be accessed to hit any unit with a probability of 100%. $m = [0.01, 0.1, 0.5]$, $r = [0, 0.5, 1]$ and $f = [0, 0.5 * m, m]$ are applied on Equation 4.18. The different parameters result in different probabilities:

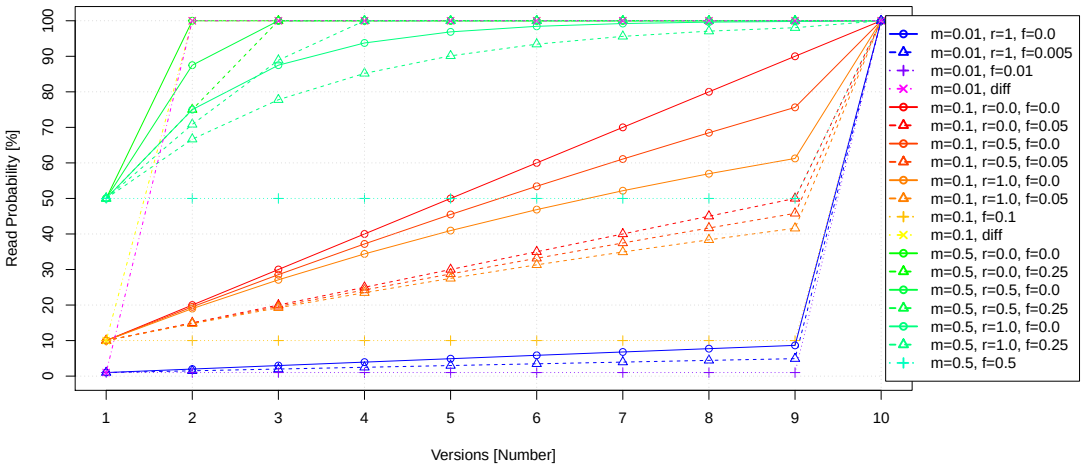


Figure 4.18: Read Effort for different Versioning Approaches

Similar to the analysis of the writes, m has the highest impact. The more units are modified, the higher the retrieval probability is.

Hotspots force the incremental versioning to always write partially the same units. Increasing values for f therefore flattens the curves significantly. If only the hotspot is written ($m = f$), the probability is not increasing for $v \to V$.

Sequential access ends up in the modification of different units in the document. Less disjoint units are accessed when modifying units randomly. Consequently, random modifications flatten the curves even further.

Figure 4.18 represents the read effort for the incremental versioning. The sliding versioning is analyzed in a similar way offering the ability to directly compare the read effort.

4.4.2.1 Sliding Reads

Sliding versioning modifies all units using at most $s \leq V$ change sets. Units are transferred manually to guarantee a constant access. $o_{slid}(v)$ in Equation 4.8 represents this overhead distributed over s versions. Equation 4.19 represents the read access $b(v)$. Since $b(v)$ might exceed 1, a distinction of cases restricts the read probability to 100% resulting in Eq. 4.20:

$$b(v) = (m + o_{slid}(s)) * v \tag{4.19}$$

$$p_{slid}(v) = \begin{cases} b(v) & b(v) < 1 \\ 1 & \text{otherwise} \end{cases} \tag{4.20}$$

$p_{slid}(v)$ defines thereby the gradient of the probability curve. The constants from

Figure 4.8 are summed up over s change sets reaching a probability of 100% automatically.

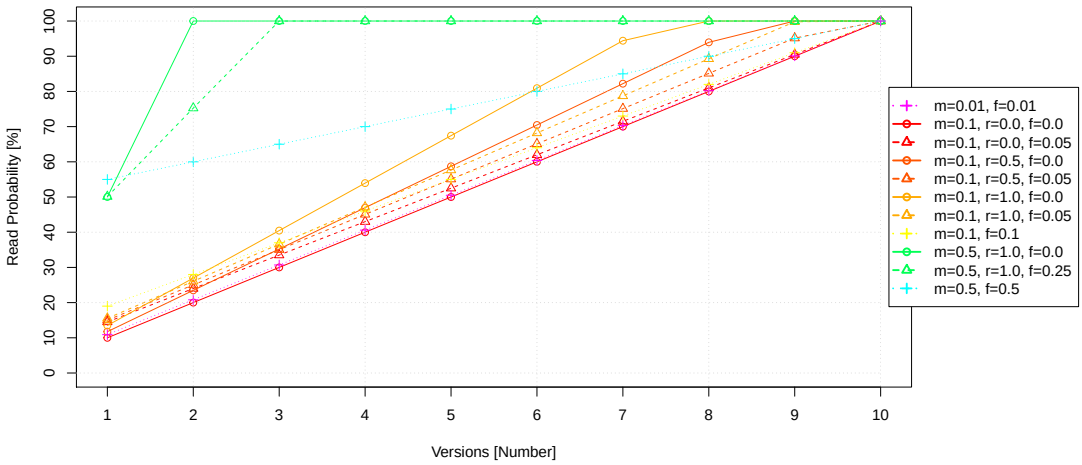


Figure 4.19: Read Effort for Sliding Versioning with a Sliding Window of s = 10

Figure 4.19 plots the read probability for the sliding versioning. $m = [0.01, 0.1, 0.5]$, $r = [0, 0.5, 1]$ and $f = [0, 0.5 * m, m]$ are the input parameters for the curves in Figure 4.19. $s = V$ is fixed to 10 and guarantees a direct comparison with Figure 4.18.

At least two change sets need to be accessed. A higher modification rate ends up in a higher access probability. Smaller modification rates like $m \leq \frac{1}{s}$ result at most in the need of s change sets. s represents an upper bound for the retrieval. In all cases, the access on any unit is guaranteed by accessing 10 change sets. Furthermore, the probability always grows with an increasing number of v.

Random access modifications tend to a higher retrieval probability. $o_{slid}(s)$ covers more units in these cases. The curves with $m = 0.1 \wedge r > 0 \wedge f = 0$ need less than s change sets for reconstruction. Increasing suitable hotspots extenuates this behavior. Only writing hotspots (e.g. $m = 0.5, f = 0.5$) trails the remaining units $(1 - f)$ over s change sets. Hotspots influence thereby negatively the read probability.

$p_{slid}(v)$ as well as $p_{conv}(v)$ allows a comparison of the read effort. $V = s$ offers a foundation for this evaluation.

4.4.2.2 Comparing the Read Probabilities

$p_{slid}(v)$ and $p_{conv}(v)$ are directly comparable over the sum of v change sets. Equation 4.21 denotes a Riemann Integral using these access probabilities.

$$P(v) = \sum_{j=0}^{v} p(j) \tag{4.21}$$

In opposite to the analysis of the write effort, the aim is to maximize $P(v)$. A higher probability results in a better chance to retrieve a desired unit.

The resulting geometric series for incremental versioning is defined as $P_{conv}(v)$ in Equation 4.22

$$P_{conv}(v) = \sum_{j=1}^{v} p_{conv}(j) \tag{4.22}$$

Similar to $p_{conv}(v)$, p_{slid} represents the counterpart for the sliding versioning. All units are accessible in at most $s \leq v$ change sets. Equation 4.23 defines the resulting Riemann Integral.

$$P_{slid}(v) = \sum_{j=1}^{v} p_{slid}(j) \tag{4.23}$$

v is independent from s and offers an observation of multiple change sets. f is indirectly included in $p_{slid}(v)$ through $o_{slid}(s)$. The comparison takes place adjacent to Equation 4.13. Equation 4.24 shows the subtraction and Equation 4.25 the normalized difference.

$$P_{diff}(v) = P_{conv}(v) - P_{slid}(v) \tag{4.24}$$

$$P_{norm}(v) = \frac{P_{diff}}{P_{conv}(v)+P_{slid}(v)} \tag{4.25}$$

For all $P_{diff}(v) \leq 0$, the access probability of the sliding versioning is higher than of the incremental versioning. Figure 4.20 to Figure 4.28 plot again different values of r and s.

Incremental versioning includes an initial change set represented by a full version for reconstruction. $V = s$ thereby allows a direct comparison of the approaches. The access probability of the sliding versioning is significantly better by observing fewer versions.

The performance improves for small modification rates ($m \to 0$). For $m \to 1, v \to \infty$, the difference between sliding versioning and incremental versioning vanishes.

s defines the threshold of the number of necessary change sets. For $v > s$, the normalized difference decreases fast. Combined with r, a larger amount of different units written improves the retrievability. More randomness therefore results in better retrieval probability of the sliding versioning.

In all scenarios, sliding versioning always generates a better retrieval probability than incremental versioning. For a larger amount of change sets, the benefit of the guaranteed access by sliding versioning disappears. The correct choice of s denotes a trade-off between the writing of unmodified units and the guaranteed access of the change sets in a version.

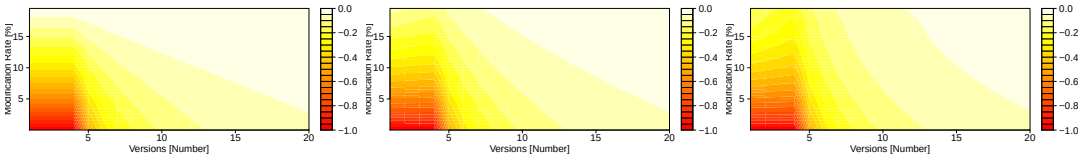


Figure 4.20: Read Effort of Sliding Versioning and Differential Versioning with $s = 5; r = 0.0$

Figure 4.21: Read Effort of Sliding Versioning and Differential Versioning with $s = 5; r = 0.5$

Figure 4.22: Read Effort of Sliding Versioning and Differential Versioning with $s = 5; r = 1.0$

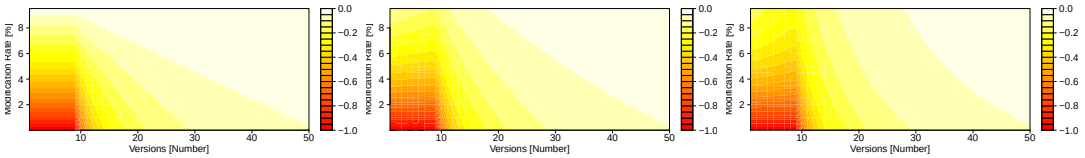


Figure 4.23: Read Effort of Sliding Versioning and Differential Versioning with $s = 10; r = 0.0$

Figure 4.24: Read Effort of Sliding Versioning and Differential Versioning with $s = 10; r = 0.5$

Figure 4.25: Read Effort of Sliding Versioning and Differential Versioning with $s = 10; r = 1.0$

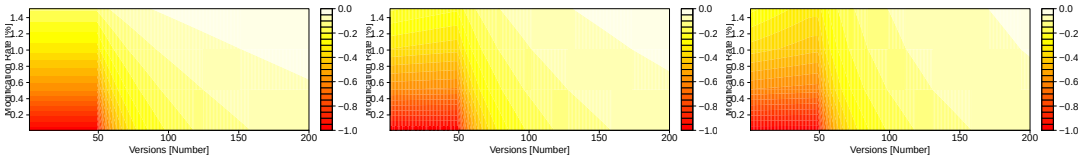


Figure 4.26: Read Effort of Sliding Versioning and Differential Versioning with $s = 50; r = 0.0$

Figure 4.27: Read Effort of Sliding Versioning and Differential Versioning with $s = 50; r = 0.5$

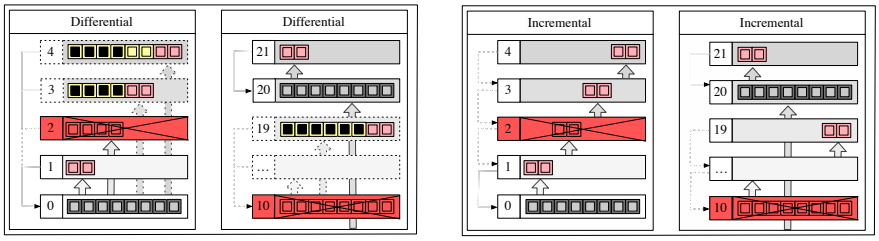
Figure 4.28: Read Effort of Sliding Versioning and Differential Versioning with $s = 50; r = 1.0$

4.5 Increasing Robustness

Loosing single change sets harms the robustness of versioning approaches. The impact of a loss differs, depending on the versioning approach and the concrete change set damaged. Figure 4.29 shows the impact of lost incremental and differential change sets.

Differential versioning is able to cope with the loss of change sets regarding the reconstruction. The left side of Figure 4.29a shows the loss of change set 2. If the change set is entirely damaged, the version 2 itself becomes unretrievable. The retrieval of following versions is not harmed. They rely directly on the initial change set, represented by the full version 0, and the own change set.

Change sets in differential versioning include inherited modified units. The generation of new versions might be hampered by loosing change sets. The modifications



(a) Loosing a Differential Change Set

(b) Loosing an Incremental Change Set

Figure 4.29: Robustness of Incremental and Differential Versioning

occurring in change set 2 are included in the following change sets. These writes are represented by units with a black core in Figure 4.29a. If the change set 2 is lost before the creation of the change sets 3 and 4, these change sets do not include related inherited modified units. The deltas representing version 2 are lost. If the loss occurs after the creation, repairing version 2 becomes cheap. All deltas of version 2 are included in the change sets 3 and 4.

A loss of an initial change set represents the highest vulnerability in differential versioning. The right side of Figure 4.29a shows the loss of the full version stored as change set 10. The reconstruction of a version relies on the current change set and the related initial change set. All consecutive change sets must be restored if the full version is missing. The recovery of the change sets depends on all change sets between the preceding and following intact full versions.

The probability to generate such damage relies on the number of the injected full versions. The impact of a loss behaves anti-proportional to the number of full versions. The more full versions are injected, the more likely damage on a full version occurs. Less full versions result in an higher impact of damaged full versions.

Damaging incremental change sets hampers the reconstruction as denoted in Figure 4.29b. The complete version becomes irretrievable by loosing the related change set. Change set 2 on the left side of Figure 4.29b is lost. Accessing versions containing this change set are not aware of its deltas. Further deltas might furthermore be incorrect.

Modifications made by incremental versioning are not harmed. Incremental change sets reflect the modifications only. Creating the change sets 3 and 4 stays independent from the lost change set 2 in the example of Figure 4.29b

The retrievability in incremental versioning is most harmed by the loss of initial change sets. All change sets start from the last correct full version. The access to a version needs all deltas until the next correct full version is reached. Change set 10 on the right side of Figure 4.29b is lost. The generation of new versions stays possible. All change sets between 19 and 11 are needed to reconstruct the lost full version. The number of injected full versions influences the probability of such damage. The more full versions are present, the higher the probability of a damaged full version is. The less full versions are injected, the higher the damage of a lost full version is.

Sliding versioning relies not on deltas combined with full versions. The versioning approach trails full versions over sliding window-change sets. As a result, the sliding versioning is more robust to the loss of single change sets. Figure 4.30 shows the same loss of change sets as Figure 4.29.

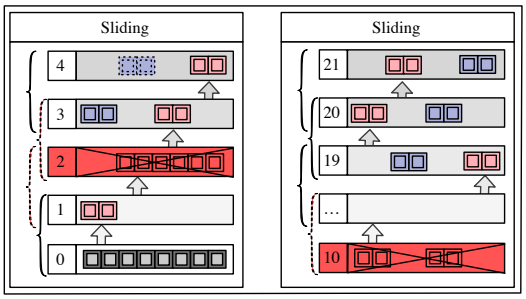


Figure 4.30: Robustness of Sliding Versioning

The loss of a change set hampers the access in three versions in the example of Figure 4.30. The lost of change set 2 restricts the access in versions 1 and 3. Sliding versioning compensates a loss of any change set in the range of s preceding and upcoming change sets.

Damages force the sliding versioning to retrieve its units from a former valid change set. The loss of change set 2 is compensated by the copy of the lilac elements from change set 0. This retrieval needs again at most s preceding change sets.

Since full versions are missing in sliding versioning, the overhead-data is trailed. The right side of Figure 4.30 shows the loss of a balanced change set. The access in version 10 does not differ from accessing version 2.

Sliding version generates balanced change sets over the time. The robustness compensating the loss of change sets benefits from this balancing. Compared to incremental versioning or differential versioning, the reconstruction effort relies on the sliding window. The overhead defined by trailed, injected full versions increases thereby the robustness of sliding versioning.

4.6 Conclusions

The need to trace actions occurring on the data naturally results in the usage of versioning approaches. Especially for cloud storages, sliding versioning is predestinated since a version relies always on only a constant number of change sets. Therefore, the following problems are solved when applying sliding versioning.

1. Scrubbing automatically protects sliding window change sets by accessing any version. The sliding window increases robustness by repeating and reorganizing active data into consecutive change sets.

2. Sliding versioning results in evened out write demands working towards balanced transfer and storage. The resulting balancing of data not only increases robustness but also improves the performance.

3. Independent from access patterns, change sets are always balanced by writing unmodified data piggyback on top of deltas. Evaluated against random and sequential access and including hotspots, the sliding versioning proves autonomy against different kinds of modifications.

4. The sliding window provides a guarantee about the amount of change sets representing single versions. The choice between fewer requests transferring more data or multiple, low-traffic requests is intuitively settable by the sliding window.

The sliding versioning solves many issues appearing by applying versioning to remote storages. Increasing robustness, guaranteed retrievability, and an upper bound for the load in buckets persuade the combination of sliding versioning with additional security measures to guard accountability in cloud storage.

A liar needs a good memory.

Quintillian - Roman Rhetorician / 35 - 96

5 Integrity in Key/Value-Stores

Contents

Lies, here generalized to deliberate and accidental errors, not only vanish in the enormous mass of information stored in the cloud. Indirectly, cloud storage supports continuous "lying".

1. Data in the cloud is handled by data models providing eventual consistency only. The huge mass of stored information needs relaxed storage requirements and promotes the corruption of data.

2. Stateless communication with the cloud relying on REST [53] further weakens consistent data handling. Acting as main mechanism to push data in the cloud, complex updates covering multiple resources are directly not supported.

3. Finding errors in cloud-stored data is expensive. The cloud restricts instant access by its location and billing models. Integrity checks on the storage node itself do not satisfy the end-to-end principle [99] and are therefore unusable to protect data against phantom accesses and storage failures.

The distant location not only constricts continuous awareness of the status of the data. ACID conformity combined with REST, the latter acting as interface for most APIs to professional No-SQL stores, is hard to guarantee since integrity can be violated

even before the data reaches stable storage. This chapter describes an interconnected bucket structure allowing COW on buckets not only enabling ACID on RESTful resources but also offering continuous awareness of the integrity relying on hierarchal integrity checks.

5.1 Background

Cloud storage providers motivate the usage of their products by advertising always-on availability to original uploaded data. Integrity is hard to achieve with technical means only. The CERN published a study in 2007 evaluating the failures on their storage infrastructure [95]. Integrity tests on their local infrastructure revealed bit rots on block level. Single bits are incorrectly persisted on the disks. These silent corruptions stay normally unrecognized. The errors occurred by common usages anywhere in the I/O chain.

Extending the classic I/O chain to cloud storage systems adds new levels of complexity. Integrity problems naturally become even more challenging. The end-to-end argument already shows the need for end-to-end integrity for storing data [99]. The preparation of data transfer, the transfer itself and the processing of the arrived data must therefore be guarded additionally to the local I/O path. Cloud storage providers are aware of the high vulnerabilities in this processing chain: Amazon, as one example, guarantees high accessibility but no integrity in their Service Level Agreement (SLA).

> **10. Disclaimers**
>
> We [...] make no representations or warranties [...], including any warranty that the service offerings or third party content will be uninterrupted, error free or free of harmful components, or that any content, including your content or the third part content, will be secure or not otherwise lost or damaged.
>
> `http://aws.amazon.com/agreement/`, accessed november, 25th, 2013

SLAs have to be interpreted from a legal perspective. The purpose of a SLA is to protect a company against any kind of recourse claims. The terms and sentences in SLAs are often standardized independent from the service offered. Amazon nevertheless might be aware that a throughout guarantee for integrity is technically not possible.

The trust put into the cloud is one of the main arguments for its usage. It is questionable if users are aware of the relaxed definition of integrity as put forth by the cloud storage providers. The technical control ends, as described in Section 3.2, by pushing data to the cloud. Users thereafter have no possibility to obtain continuous updates about location and status of the data.

Recent research focuses on guarding integrity on remote storage. DepSky [30, 85] and HAIL [35, 36] try to extend the *Threshold of Technical Control*. They apply a combination of probing, mirroring, and checksumming as described in Section 3. Proofs of data possession, proofs of retrievability and quorum-based data access provide as much awareness as possible of the data.

These approaches guard the data while stored in the cloud. Many common local storage failures are defined as "phantom writes". These writes behave as if they were to provide data persistence without actually doing so. In fact, the storing operation never succeeds. The push of data to the cloud would never or only partly occur and the client would be left uninformed. Misdirected reads and writes might access incorrect buckets and jeopardize the integrity.

Cloud storage is commonly accessed through REST [53]. Taking HTTP [52] as base, REST defines stateless access to resources accessible with the help of a uniform resource locator (URL). Its stateless paradigm forbids REST to directly access multiple resources at one time. To enable such multi-resource access, these resources must be summarized under one resource. When applied to the handling of buckets, a structure covering multiple buckets at one time must be developed. Such a structure extends common integrity guarding approaches described in Section 3.

5.1.1 Contribution: Hierarchical Bucket Order

Modern file systems, such as the "Zettabyte Filesystem" (ZFS) [33], implement measures against a vast number of errors. The I/O chain ranging from local data access to storing the information remotely motivates their mapping to the cloud. Section 5.2.1 explains the protecting paging mechanism from ZFS acting as blueprint for the implemented architecture.

The following approach is built upon Marc Kramis' research about versioned XML storage [60]. A mechanism to serialize XML into pages is adapted. This approach arranges buckets similarly to this paging mechanism. The resulting bucket structure supports remote integrity checks. A hierarchical structure binds buckets together. The resulting architecture guards data against remote bit rots, misdirected reads/writes and accidental failures. The own contribution thereby consists of solutions for the following requirements.

1. Complex operations often cover multiple buckets at one time:
   → Integrity must cover multiple buckets as well. This includes access to multiple buckets at the same time. Relaxed storage models must be respected.

2. Read-only access to previous versions must be cheap and versatile. The number of buckets required to access a version and the overhead of the versioning must remain manageable:
   → Sliding versioning combines integrity with availability.

3. Requests to the cloud should be atomic. Create/delete-requests are preferred over the modification of existing buckets. Relaxed storing models in the cloud need to

be respected.
→ Relying on REST, PUT and DELETE requests are preferable. This preference results in an append-only storage. Even modified data results in the generation of new buckets.

5.2 Ordering Buckets in Treetank

The cloud storage community focuses mainly on the availability of the stored data. Additional security measures covering complex, atomic multi-bucket accesses stay out of focus so far. The same applies for combining security approaches e.g. checksums with an adaptive versioning mechanism. When the location is untrusted, integrity is commonly applied on file level e.g. by Plutus [72] or Sirius [55]. Integrity-checked architectures still represent an active field of research in the file system community. ZFS, a prime example, represents one of the first implementations of a file system focusing on integrity.

Due to the complex architecture of ZFS, only the parts guarding integrity will be discussed. The idea of hierarchical buckets before upload is inherited from ZFS. An architecture, developed by Marc Kramis in context of versioned XML storage [60], acts as template. The resulting bucket ordering is influenced by the following architectural considerations.

- Copy-on-write (COW) results from the hierarchy of buckets. The hierarchy is serialized bottom-up. The access to any bucket in the hierarchy occurs always via the root. Modifications covering multiple buckets become thereby atomic from a client's perspective.
- Versioning is easy applicable on the bucket structure. Read-only access to former versions combines integrity with availability. Old states can replace incorrect modifications.
- Independent checksums on single buckets are not satisfactory to prohibit phantom accesses. In the hierarchy of buckets, checksums are stored in related parent buckets. This mechanism guarantees protection against bit rots including faulty and dislocated writes.

5.2.1 Integrity Checks Inherited from ZFS

The bucket structure leans on the block arrangement of ZFS. A brief summary of the integrity-guarding measures in ZFS is therefore provided. Figure 5.1 shows a hierarchical order of blocks in ZFS including example logical block addresses. This arrangement satisfies the considerations denoted above.

All blocks are numbered preorder starting with the Uberblock assigned as logical block address 1. Hierarchical ordered blocks are referenced over pointers. The pointers are represented by logical block addresses of children blocks. Retrieving a leaf block costs therefore $\mathcal{O}(log(n))$ accesses[1]. The benefits of this hierarchy are tree-aware hashes

[1] n represents the number of blocks in the tree.

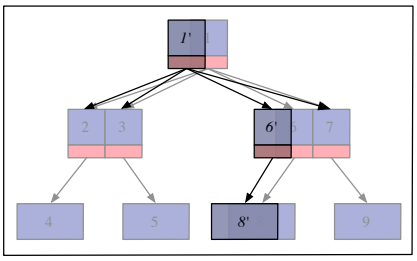


Figure 5.1: COW, Checksums and Versioning in ZFS

leaned on Merkle-Trees [88]. Merkle-Trees protect the integrity of entire subtrees by their root nodes. Parent nodes store the checksums of blocks, denoted by the red areas in Figure 5.1. The Uberblock, as root node representing an exception, stores its own hash in addition to the hashes of its children. Checking the hash of the Uberblock checks thereby its own integrity and preserves the integrity of the entire block tree.

All data is stored in leaves only. The intermediate blocks only reference the leaves and store checksums. The tree is always modified bottom-up. Modifications are atomic in the tree satisfying the COW-principle. COW as a result is widely used in all types of log-structured file systems.

The data in the logical block 8 is modified resulting in a new block $8'$ in Figure 5.1. Since the checksums must be adapted, block 6 must be modified. The new block $6'$ stores a pointer to the new block $8'$ and includes its new hash. The final write of the new Uberblock $1'$ represents an atomic operation. It is arguable that the checksum of this block guards its own content as well. The write of one single block is always atomic. Since the Uberblock is written last, the modifications are either entirely accessible or inaccessible. COW naturally enables snapshots by keeping old blocks. Blocks 2 and 3 have two parents in this case since they stayed untouched in the most recent version. The creation of snapshots is therefore an instant operation in ZFS and many other file systems relying on COW and the log-structured file system paradigm. The cheap versioning and the free protection of blocks relying on the tree structure motivate the mapping of the block hierarchy to the storage of buckets.

5.2.2 Bucket Hierarchy

The hierarchical block structure is mapped to buckets. The resulting bucket order ends up as backbone of the storage system Treetank. Figure 5.2 shows the bucket architecture:

The architecture defines a fixed set of different bucket types. All buckets are identifiable in the No-SQL store with the help of a key called bucket identifier. The orange elements in all buckets in Figure 5.2 denote the bucket identifiers. Storing a bucket identifier of another bucket represents a reference. Additionally, the different buckets contain attributes according to their purpose in the structure.

Uber Bucket: The overall root bucket is called uber bucket in analogy to the Uberblock.

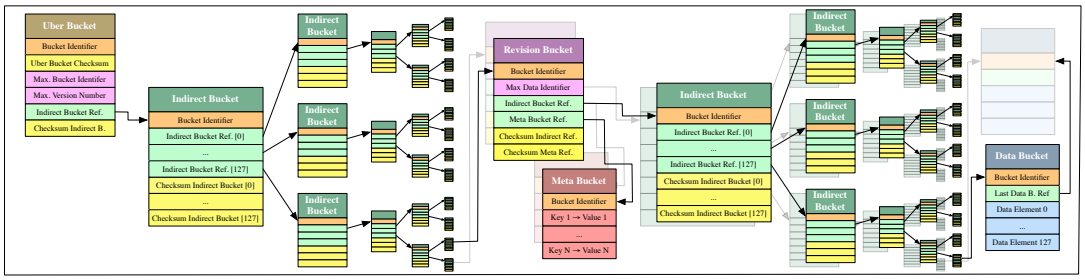


Figure 5.2: Hierarchically Ordered Buckets in Treetank

The uber bucket manages the bucket identifiers and the revision number denoted by the pink elements. Both counters are incremented by creating a bucket or a new version. The uber bucket contains exceptionally its own checksum as well. In all other buckets, the checksum is stored in parent buckets. Each bucket can be referenced from various parent buckets if related subtrees stay unmodified. A reference stores a tuple consisting of the bucket identifier of the indirect bucket acting as children including its checksum. References are always shown as green elements in all buckets in Figure 5.2. Checksums are represented as yellow elements in the buckets. Summarized, the uber buckets always act as entry point for each access.

Indirect Bucket: Indirect buckets store the bucket identifiers of other indirect buckets, data buckets or revision buckets acting as children. The purpose of the indirect buckets is multiplying the fanout of the tree. Each indirect bucket offers $2^7 = 128$ pointers to the lower levels. 5 layers of indirect buckets addresses up to $2^{7^5} = 34359738368$ data buckets or revision buckets. Additionally, the checksums of linked buckets are persisted in each indirect bucket.

Revision Bucket: Revision buckets represent single revisions. Besides the linking to the indirect bucket including its checksum, a meta bucket is referenced. An additional, incrementing counter offers semantic access to the data. This counter is called data identifier and references to units stored in the data buckets. The access to any data bucket starts at a given revision bucket.

Meta Bucket: Meta buckets optionally store application-dependent information as key/value pairs. Each revision bucket holds one reference to a meta bucket. Examples for meta information are mappings for common tag names while storing XML and paths to inlaying files while mapping a file system. Implemented mappings are described in detail in Chapter 6.

Data Bucket: Similar to ZFS, the leaves called data buckets store all data. The using application defines atomic data units called data elements in the rest of this thesis. The data elements constitute the main content of the data buckets. Different implementations of data elements represent application-dependent mappings.

Table 5.1: Offset Thresholds per Level in Indirect Buckets in Bucket Architecture

Data Identifier	Level				
	0	1	2	3	4
$0 \dots 2^7 - 1$	0	0	0	0	$0 \dots 127$
$2^7 \dots 2^{14} - 1$	0	0	0	$1 \dots 127$	$0 \dots 127$
$2^{14} \dots 2^{21} - 1$	0	0	$1 \dots 127$	$1 \dots 127$	$0 \dots 127$
$2^{21} \dots 2^{28} - 1$	0	$1 \dots 127$	$1 \dots 127$	$1 \dots 127$	$0 \dots 127$
$2^{28} \dots 2^{35} - 1$	$1 \dots 127$	$1 \dots 127$	$1 \dots 127$	$1 \dots 127$	$0 \dots 127$

Chapter 6 describes example mappings in detail. Each data element receives an own identifier called data identifier. The data identifier is managed by the revision buckets. Each data bucket stores a fixed range of data elements, 128 in the example of Figure 5.2. The data identifiers identify the data elements including the storing data buckets. For example, the first data bucket stores all elements with data identifiers = $[0 \dots 127]$. The next data bucket stores all elements with data identifiers = $[128 \dots 255]$.

The bucket identifiers and the data identifiers are independent from each other. The access to a data element with the help of a data identifier needs the traversal of 5 layers of indirect buckets to access the related data bucket. Each indirect bucket stores 128 referenced bucket identifiers. The references in the indirect buckets are accessed by offsets.

Table 5.1 shows the offsets of indirect buckets on each layer mapped on different ranges of data identifiers. On each layer, the bucket identifiers referenced under these offsets point to the next indirect bucket to be accessed.

The revision buckets need to be dereferenced as well. The revision number represents the position of all revision buckets on their level acting as data identifier-equivalent. The dereferencing of a revision bucket relies on the same offset computation.

Versioning is achieved by accessing data buckets through revision buckets. Revision buckets are accessible with the help of version numbers. Sliding versioning is applied directly on the data buckets. Each data bucket represents a change set. Pointers between data buckets offer accesses to its former change set. This additional pointer spares the dereferencing over revision buckets when reconstructing a version. Without the additional pointer, the reconstruction would be in need of the traversal of linked indirect buckets. Since non-modified buckets are referenced by multiple revision buckets, an unknown number of revision buckets including their subtrees would have to be traversed. The extra pointer between the data buckets offers scalable access to its former change set to spare such traversals.

All bucket implementations must compute their content's hash. The origin of the checksums for data buckets and meta buckets is defined by their application-dependent representations. Recursively, each indirect bucket stores related checksums adjacent to the offsets of the bucket identifiers set. The bucket identifiers and the stored checksums allow each indirect bucket to generate an own hash. Uber buckets are storing the hash of their subtree including their own hash. They represent, similar to ZFS, the only exception of a bucket storing its own checksum. The checksums are examined similar

to the checks of ZFS upon retrieval. Optionally, ZFS-like scrubbing operations can be applied by iterating through all buckets.

The binary representation of each bucket can be extended with Error Correction Codes (ECC) compensating small errors not only in storage but also in transfer. Probing and mirroring can furthermore benefit from the resulting hash structure. Applying this structure to remote-guarding mechanisms in detail stays out of focus of this work so far.

5.2.3 From DAG to Buckets, An Example

The bucket ordering ends up in a directed acyclic graph (DAG). An example DAG representing multiple revisions is shown in Figure 5.3. Each bucket receives its bucket identifier from the system. The bucket identifier is denoted by the number in the buckets in Figure 5.3. The black squares on the bottom of referencing buckets represent the pointers to underlying buckets.

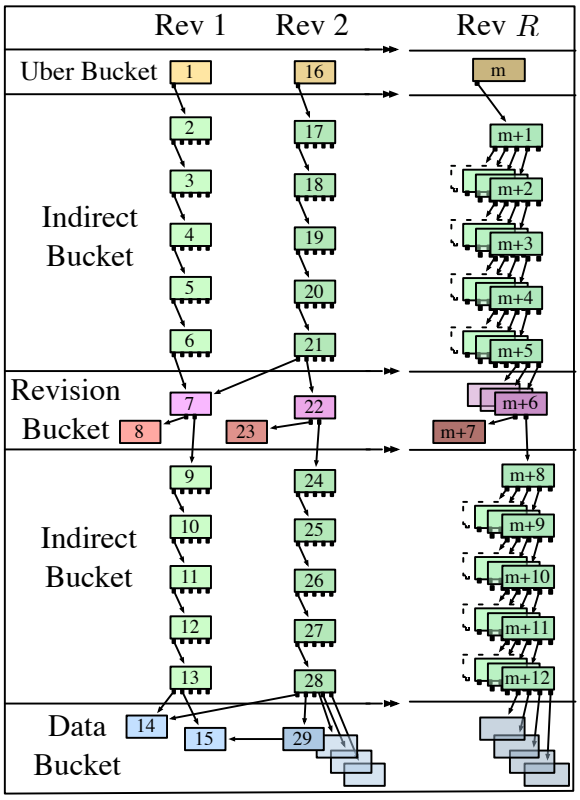


Figure 5.3: Hierarchy of Buckets representing multiple Versions

Revision Bucket 7 represents version 0. The indirect buckets $2, 3, 4, 5, 6$ reference this

revision bucket. Version 0 automatically results in offset 0 in all referencing buckets.

$w_{r=0} = 2*128$ data elements are inserted in the first version resulting in the creation of the data buckets 14 and 15. Each data element receives a data identifier in the range of $[0 \dots 255]$. By accessing any data element, indirect buckets $9, 10, 11, 12$ need to be accessed. In these indirect buckets, always the pointer stored at offset 0 is used. The data elements with data identifiers $[0 \dots 127]$ are accessible over the offset 0 in indirect bucket 13 pointing to data bucket 14. The other data elements with data identifiers $[128 \dots 255]$ are accessed over offset 1 referring to data bucket 15.

The indirect buckets $2, 3, 4, 5, 6$ and the indirect buckets $9, 10, 11, 12$ only differ related to their position in the bucket structure and thereby their purpose. The first indirect buckets offer access to the version represented by the revision bucket 7. The other indirect buckets provide access to the data elements stored in data buckets 14 and 15.

Version 1 inserts $w_{r=1} = 3*128+1$ data elements with the data identifiers $[255 \dots 640]$. Four data buckets are created in version 1. The data elements with the data identifiers $[256 \dots 640]$ are no existing in version 0. The semi-transparent, blue buckets linked by the offsets $2 - 4$ from the indirect bucket 28 store these new data elements. The data element with the data identifier 255 overrides the last data element in the data bucket 15. Sliding versioning stores change sets incrementally. The change set results in the new data bucket 29. Data bucket 29 stores only the modified data element at offset 127. The backward pointer from data bucket 29 to its former status 15 offers direct access to the entire status. Since the data elements with the data identifiers $[0 \dots 127]$ stay unmodified, the already persisted data bucket 14 is linked at offset 0 in the indirect bucket 28. Satisfying the append-only paradigm to always write new buckets, new indirect buckets become necessary to link to the data buckets. The new pointers trigger the mirroring of the indirect buckets $9 - 13$. The revision bucket 22, representing version 1, links (indirectly) to the indirect buckets $24 - 28$. The last bucket to be serialized is the uber bucket with the bucket identifier 16.

This example shows that a complex write operation covering multiple buckets still adheres the COW-principle. The uber bucket denotes the last bucket to be serialized. The most recent uber bucket always represents the main entry point to further operations.

In contract to the bottom-up serialization, the new buckets are created top-down. After writing R revisions, m defines the bucket identifier of the related uber bucket. All new indirect buckets become directly accessible from this uber bucket. They are assigned the next available bucket identifier as part of the new path to the most recent revision bucket. Related indirect buckets are mirrored from former versions. Old revision buckets are referenced over the same positions. The new revision bucket is linked by the next available offset. The same mechanism applies to new indirect buckets on the way to data buckets containing new or modified data elements.

The pointers to former, unmodified buckets are kept. Only new or modified content is serialized. Old content stays accessible over already set offsets.

Figure 5.4 shows the serialization of the buckets from Figure 5.3. The buckets are

serialized in single revisions.

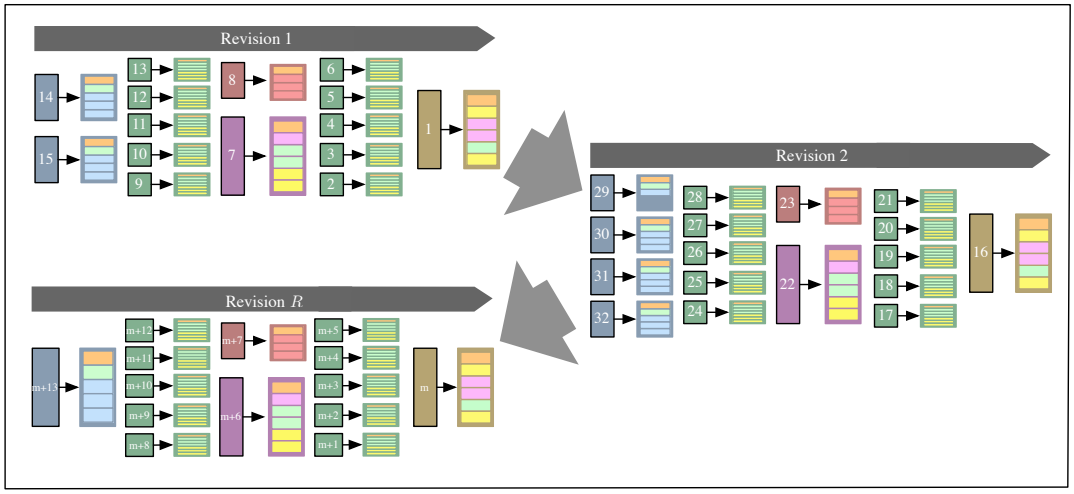


Figure 5.4: Persisted Buckets representing multiple Versions

The bucket identifier acts as key in the No-SQL store. The single buckets are serialized including their attributes explained in Figure 5.2. The attributes are transformed into their byte representations. The resulting byte representation allows optional compression and ECC application. The byte representation acts as value for the blob written to the cloud storage. The related key is given by the bucket identifier.

The usage of sliding versioning combined with this architecture achieves the following advantages.

1. The bucket structure includes hierarchical checks on the data. These checks are performed automatically upon retrieval. Checks can furthermore be manually triggered. By reconstructing the status of a bucket, the check covers multiple revisions. All checksums are persistent. Proofs of retrievability and remote-checking operations directly benefit thereby from the structure. The bucket ordering can be seen as offline extension to such established approaches.

2. All modifications result in the generation of new buckets. This creation covers also indirect buckets pointing to new data buckets. Unmodified buckets are linked by their position in the structure. Versioning is applied for the data elements storing modifications as deltas.

3. The bottom-up serialization results in atomic operations binding multiple buckets together. REST makes such operations on multiple buckets otherwise hard to achieve. The pointers accessible from the recent uber bucket always include the newest revision bucket. Since each serialization ends with the persistence of this uber bucket, COW is provided without harming the stateless paradigm of REST.

4. Different data is mapped to buckets. The access to stored data must only rely on the data identifier. Implementing applications gain automatic versioning and hierarchical integrity-checks. The resulting byte representation furthermore supports byte-modifying techniques like ECC, compression, encryption and checksums. Examples of such mappings are given by Chapter 6.

The well-established mechanisms from log-structured file systems like ZFS enable COW and introduce Merkle-Trees to cloud storage. Ordering buckets hierarchically enables sliding versioning to version buckets. Introducing backward pointers between data buckets on the same position makes additional tree traversals obsolete. Instead, retrieving only a constant number of change sets guarantees the access to any version of a bucket. The morphing nature of data elements enables the storage of different data types in Treetank.

5.3 Performance Costs

The implementation of the data elements is application-dependent. Different usages of this structure are presented in Section 6.1. The benchmarks in this chapter cover inserting, updating and retrieval operations of dummy data elements only. Each benchmark is analyzed by two different representations.

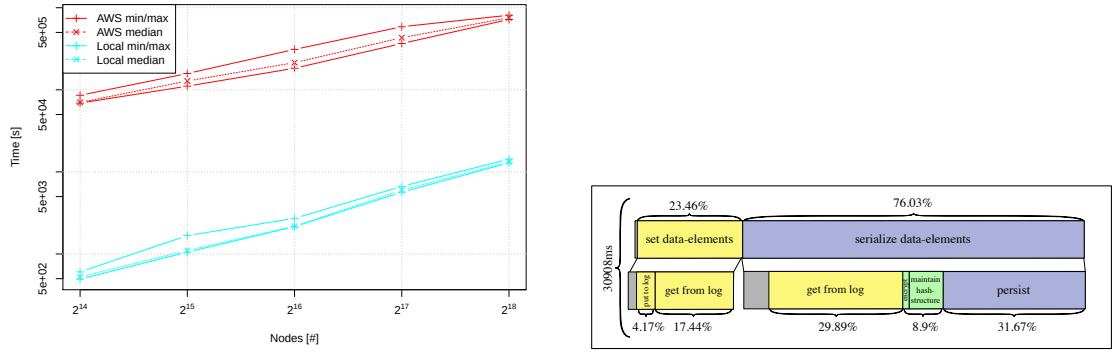
- The benchmark is performed against a local storage and the cloud. The focus of this comparison lies on the overhead of the remote location Since the cloud storage must be seen as a black box, only the entire storage process is benchmarked remotely. The impact of single operations on remote storages can not be evaluated in detail.
- Each benchmark is split in sub-operations given by their percentages on local storages. Analyzing the different components gives an overview about the costs of single operations locally.

The dummy elements store 1024 random bytes as data. All benchmarks are performed on a local solid-state disk and the Amazon S3 (AWS S3) cloud.

5.3.1 Insert

The insertion benchmark generates an incrementing number of dummy elements $n = 2^i \mid i \in [14, 15, 16, 17, 18]$. A version is created after inserting $\frac{n}{8}$ elements. Sliding versioning is applied with a sliding window of 4.

Figure 5.5a shows the performance. Serializing the data elements in a local backend consumes roughly 1% of the time compared to the access in the cloud. Inserting 2^{18} data elements in the cloud takes 13 minutes. The long insertion time makes the remote location of the cloud unusable for time-crucial applications. The performance of the local storage motivates a local mirror. Such a system allows fast insertions of data including the creation of new versions.



(a) Performance of Insert

(b) Distribution of Insert-Operation

Figure 5.5: Evaluation of inserting Data in Bucket Hierarchy

Figure 5.5b analyzes the sub-operations. 2^{18} data elements are inserted in a local backend. This operation consumes 31 seconds[1]. The plain insertion of data elements only takes 23.46% of the time. The access to the transaction log is most expensive. Not-yet serialized buckets are stored in this transaction log. These accesses are represented by the yellow elements in Figure 5.5b. Accessing this log while serializing consumes as much time as persisting the buckets (29.89% against 31.67%). The security guarding mechanisms take 8.9% of the time denoted as green part in Figure 5.5b. The security overhead decreases when using the cloud. In this case, the persisting effort will increase massively, as the access to the cloud represents the bottleneck. Establishing security measures becomes then relatively cheap.

5.3.2 Get

The get operation is evaluated by retrieving $n = 2^i \mid i \in [14, 15, 16, 17, 18]$ data elements. For preparation, 2^{15} data elements are inserted in 8 versions. Afterwards, 2^{15} data elements are modified randomly in additional 8 versions. These modifications shuffle the loads in the buckets. Sliding versioning is used with a sliding window of 4.

The units in the data buckets are randomly distributed. The retrieval includes thereby the reconstruction of entire versions of data buckets. The benchmark retrieves the data elements sequentially and randomly. The data elements are stored in a local backend and the cloud.

Figure 5.6 shows the results of this benchmark. Sequential retrieval of 2^{18} data elements from a local backend takes 6.6 seconds in average. Compared to the Amazon Cloud, the local retrieval is between 209 and 262 times faster: Retrieving sequentially 2^{18} data elements from the cloud takes 1411 seconds in average. Temporal caching of already retrieved buckets is only slightly improving this performance. The remote

[1]The difference between the insertion time of 2^{18} in the benchmark and the analysis origins from the analyzing overhead of the profiling procedure.

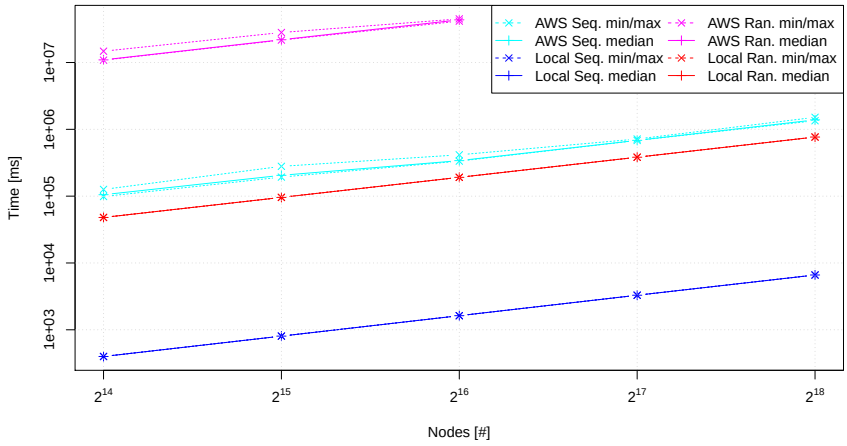
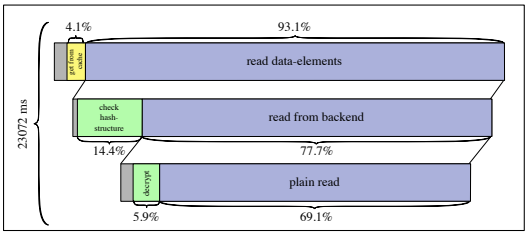


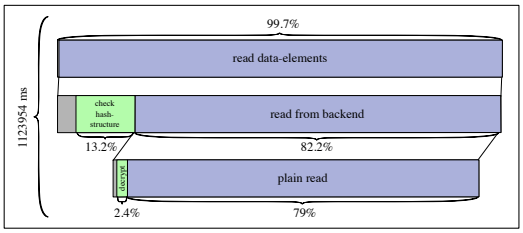
Figure 5.6: Time for Getting Data out of Bucket Hierarchy

backend needs to be mirrored entirely in a local storage.

The impact of mirroring becomes visible especially by evaluating random accesses. Accessing 2^{16} data elements randomly in the cloud takes 43370 seconds. Remote access consumes thereby between 226 and 239 times more time than accessing data locally. The caching of already retrieved buckets has no impact on randomly chosen data elements.



(a) Distribution of Get, Sequential Access



(b) Distribution of Get, Random Access

Figure 5.7: Distribution of Sub-operations for Getting Data out of Bucket Hierarchy

Figure 5.7 shows the distribution of the sub-operations on a local backend. Sequential reads from the disk consumes 69.1% of the time represented by Figure 5.7a. 4.1% of the time is spent to retrieve data from the cache. The impact of caching is near to zero by randomly accessing data elements represented by Figure 5.7b. The plain read performance increases to 79%. Consequently, the effort needed for checking the data decreases. Integrity-checks and encryption will become cheaper when applied to the cloud. The access itself represents again the bottleneck. Using a cloud backend without local mirroring makes random accesses practically impossible.

5.3.3 Update

Updating operations represent a combination of get with insert operations. The benchmark inserts 2^{18} data elements in one version. Afterwards, $n = 2^i \mid i \in [14, 15, 16, 17, 18]$ elements are replaced in 8 versions. $\frac{n}{8}$ data elements are updated per version. Buckets to be updated are either entirely sequential or chosen completely at random.

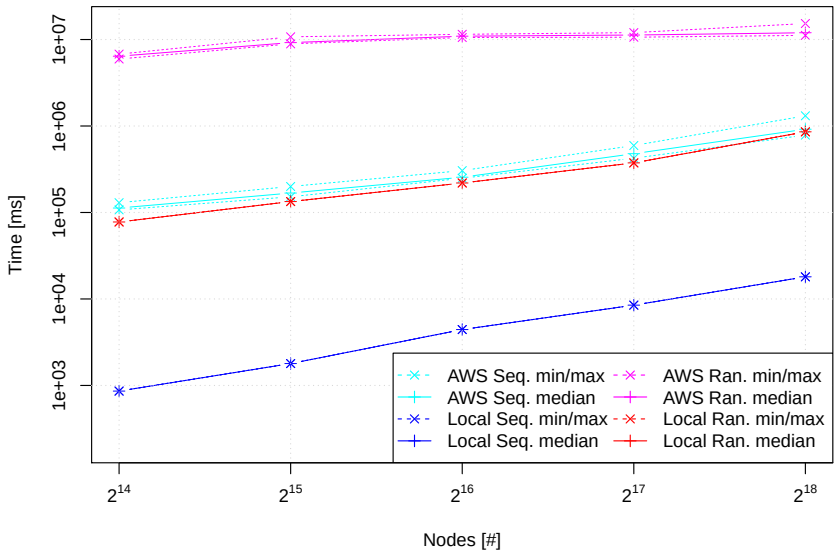
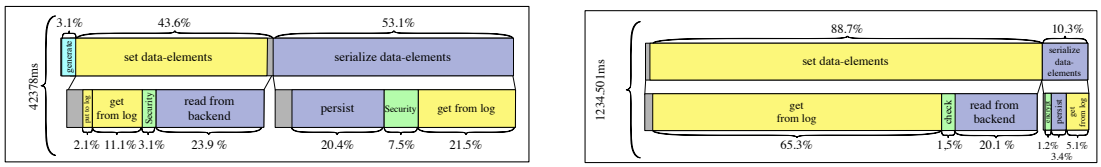


Figure 5.8: Time for Updating Data in Bucket Hierarchy

Figure 5.8 shows the scaling for updating buckets on remote and local storage. The sequential update on local storage consumes between 1.08% and 2.21% of the time compared to random updates. On remote storage, the overhead of random updates to sequential updates decreases from factor 55.74 to 12.74 with an incrementing n. The probability that a bucket is modified at least twice in one version increases with the number of elements modified. As soon as one bucket is modified in a version, it is stored in the local transaction log, enabling faster access to the same bucket for further modifications. Nevertheless, updating 2^{18} elements randomly consumes 207.67 minutes in average making remote storage without a local mirror unusable in practice.

A detailed distribution of sub-operations is shown in Figure 5.9. The sequential update allows a detailed analysis of single operations represented by Figure 5.9a. The yellow elements again cover the intermediate handling of the data by the transaction log. The serialization of the buckets consumes 53.1% of the time. Applying security mechanisms takes 7.5% of the time. This time includes the computation of the checksums and the encryption of the buckets. The distribution of sub-operations of random updates is represented by Figure 5.9b. The transaction log, again represented by the yellow parts,



(a) Distribution of Update, Sequential Access **(b)** Distribution of Update, Random Access

Figure 5.9: Distribution of Sub-operations for Updating Data in Bucket Hierarchy

buffers access to already written buckets in a version. This shows the benefit of local mirroring. The maintenance of the security measures takes only 1.2%. of the time. Especially modifications on randomly chosen buckets become cheaper when using local mirrored data.

5.4 Conclusions

Hierarchical storage structures are not cutting-edge although never applied to cloud-based No-SQL stores. Applying an hierarchical arrangement to buckets leaned on established log-structured file systems solves most problems of inconsistent data handling in the cloud.

1. COW is applied to cloud-stored buckets annulling consistent data handling. Either all modifications appear to be written or the current status stays inaccessible guaranteeing ACID in No-SQL stores.

2. Without modifying REST and its stateless paradigm, atomic multi-bucket operations are provided, simplified by the append-only log-structured operation. Instead of following the current trend, of providing hand-coded summary resources when their joint atomic operation is required, the hierarchical bucket order is used to gain flexibility by serializing the structure bottom-up.

3. The hierarchical structure provides straightforward and effortless recursive integrity checks, verified in normal retrieval operations. Together with the sliding window described in Chapter 4, this eliminates scrubbing for the active version. Each access on the data needs to traverse a defined path in the structure including checks of the buckets.

Additionally to these benefits, the structure is adapted to apply sliding versioning to even smaller elements stored in data buckets. Extending the hierarchical file system paradigm, the leaves represents change sets and are connected to each other. The linking of data buckets to former written leafs offers scalable retrieval and reconstruction of a version.

The architecture guards the data automatically. Applications are able to make use of the combined security measures gained with this structure by implementing data elements, while current approaches commonly establish security measures on specific data types only. The proposed bucket architecture establishes combined security techniques to morphing buckets.

Remote checks like HAIL [35] or DepSky [30] are orthogonal approaches and can in fact benefit from this structure. Stripping of the buckets can be applied like offered by DepSky. Since damaged substructures in the bucket hierarchy are easily identifiable, a cloud-of-clouds storage is able to compensated and restore lost buckets [23, 40].

The more we do, the more we can do.

William Hazlitt - English Writer / 1778 - 1830

6 Independent Structure-aware Quality of Storage

Contents

Professional cloud stores provided by versatile No-SQL databases become only accessible by mapping user- or application-related data to buckets. Using such cloud storage needs establishing of security measures before the mapping of data, application-dependently to vendor-specific No-SQL APIs, takes place. While professional cloud stores are billed on the base of consumed resources, multiple cloud storage providers grant complimentary access to their No-SQL storage through specialized services like photo sharing websites.

This chapter provides a paradigm shift returning the authority about any data to the user. Data is stored securely and for free. By layering the data, data containers are stored in the bucket hierarchy described in Chapter 5 providing checksums and versioning. The amorphous bucket structure is stored over an independent and flexible storage layer enabling even the use of photo sharing websites as No-SQL stores.

The autonomy of data containers combined with the independence of the storage layer enables infinite flexibility. By automatic protecting integrity, accountability and

confidentiality, the encrypted buckets are stored as files, images or on professional cloud-based No-SQL storage depending on the users' needs. Fig. 6.1 summarizes the resulting architecture as secure cloud storage gateway.

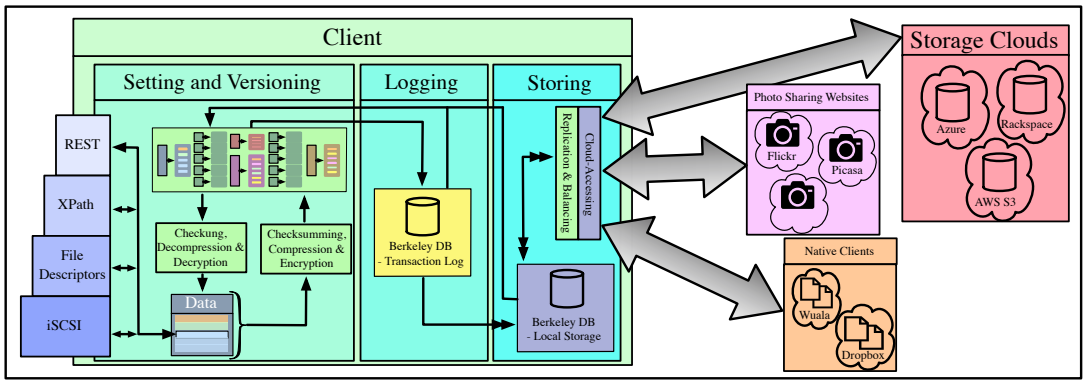


Figure 6.1: Treetank representing a Secure Cloud Storage Gateway

The components in the green "Setting and Versioning" part maps data containers like blocks, files, XML and REST to data elements. Section 6.1 describes four implementations of the data containers. Each mapping is evaluated by third-party benchmark suites showing not only the feasibility of the mappings but also the scalability of the bucket hierarchy presented in Chapter 5.

The push of data in the cloud takes place by explicitly creating a version and serializing buckets to a local and a remote backend represented by the "Logging"- and "Storing"-area of Figure 6.1.

Data to be pushed undergoes optionally additional replication and balancing steps leaned on HAIL [35] or DepSky [30]. The access to multiple clouds via a versatile interface layer using jClouds [13] enables the binding of photo sharing websites as No-SQL stores to the architecture. The transformation of the buckets into images and the handling of the resulting pictures on the photo sharing websites is described in Section 6.2.

6.0.1 Contribution: Establishing Quality of Storage

The versatility of the denoted architecture relies on the following contributions:

1. The mapping of data to buckets is always interface-dependent. To make use of the bucket arrangement described in Chapter 5, the incoming data must be mapped onto data elements summarized in data buckets:
   → The mappings of four major data containers shown in Figure 6.2, namely blocks, files, XML and REST, onto data elements are described and evaluated. Since the cloud storage is a black box, making the analysis of concrete sub-operations impossible to analyze, the mappings are evaluated from a clients perspective only.

2. To prohibit vendor lock-in, the access to cloud storage providers is abstracted. This abstraction allows storing data in different clouds. Through the proposed RAID container, even on-the-fly migration is possible.
   → One example of the proposed versatile backend architecture is the storing of buckets as images on photo sharing websites. Providing vast amounts of complimentary storage, their infrastructure inherit an alluring mass of second-tier storage usable by this architecture.

6.1 Creating Data Containers

The benefits gained by using cloud storage motivate the push of all data in the cloud. Not only files but also databases, block devices and services profit from the gained availability and all-time access. Since buckets are commonly stored in No-SQL databases, different data types must be mapped to provide key/value-relying access.

Figure 6.2 provides a classification of different abstractions layers. The defined levels range from blocks over files to specific data types like e.g. XML.

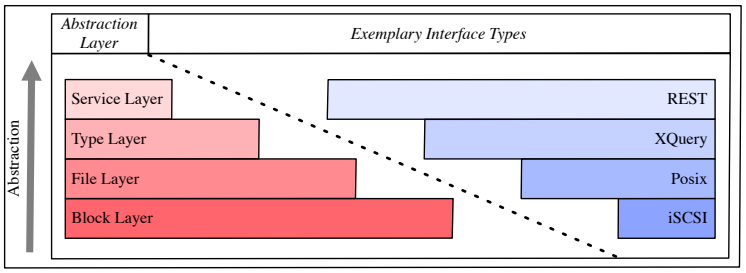


Figure 6.2: Different Layers of Data Abstraction representing Data Containers

The highest abstraction is offered by direct usable services. REST [53] as one main representative stands for resources directly accessible by any type of URL. The type of the REST resource is mainly unimportant. Nevertheless, implicit knowledge about the resource enables REST requests to be equipped with specific parameters. Data handling becomes an highly abstracted service. This layer is thereby denoted as service layer.

One example of an underlying resource for REST is XML. XML denotes an example for the type layer. Query languages such as XQuery [32] offer exploration of content and structure. Such queries can be piped through REST as parameter if knowledge about the type exists. The type layer offers still a high abstraction to the inlaying data. Content becomes accessible without in-depth knowledge about the storage.

The file layer cares about the persistence of data independent from their type. XML can be stored in the file layer represented as text. XML can also be stored in any type of XML-aware database resulting in binary-files. The file layer offers standardized access to the file system encapsulating physical layers.

The physical layer is commonly used as block device. Blocks, accessed by their logical block address, represent the lowest level of data abstraction. Besides local disks,

additional interfaces such as iSCSI [101] offer remote access to block devices.

Each of these layers offers specific interfaces to interact with the data. The different abstraction layers are stacked. Specific cloud stores exists for all the described layers. Applying commonly combined security measures becomes difficult when using these specific cloud stores. The specific characteristics and interface types must be respected.

By mapping the data containers to the amorphous, secure bucket structure described in Chapter 5, integrity, accountability, and confidentiality are gained automatically. Additionally, modifications resulted from REST requests covering multiple buckets become atomic. The mapping furthermore allows the interaction with the data by using their specific interfaces. Each layer must be mapped to buckets since the smallest denominator of cloud storage models are No-SQL stores. The key question is how to encapsulate these different levels of abstraction to use the described bucket architecture.

6.1.1 Implementation

All data stored in Treetank must be mapped on data elements. The following conditions must be satisfied for such data.

Generating Secure Hashes: The data stored in data elements must generate secure hash sums depending on its content. Relying on this hash, the hashes of the data buckets are generated representing the backbone for the bucket hierarchy described in Chapter 5.2.2.

Accessible by Data Identifiers: The data must become accessible with the help of data identifiers. The data identifier allows the computation of offsets to access the storing data bucket and referencing indirect buckets as described in Section 5.2.2.

Additionally to the data elements, meta data is optionally stored in meta buckets. Elements in the meta buckets are organized as map and not versioned. The data stored in meta buckets is optional and represents meta data of the application only. Classic examples of elements stored in meta buckets are file paths explained in Section 6.1.3 or the centralized handling of tag names in XML described in Section 6.1.4.

Referenced by data identifiers and generating secure hashes, data containers are mapped to data elements. Each access to data elements in Treetank must occur over data identifiers. The following four examples show the versatility of mapping different data containers to data elements.

6.1.2 iSCSI and Buckets

Block-wise communication protocols enable the usage of low-level, remote storage. iSCSI [101], as one major representative, bases on TCP and maps SCSI commands to remote storages. The result is an architecture consisting of mountable, delocalized block storage independent from file systems.

Blocks consist of simple byte chunks identifiable with the help of a logical block address (LBA). The LBA represents an incremented counter accessing byte chunks of a

requested length in storage. To work with Treetank, blocks are mapped to data elements. A relation between the data identifier of a data element and the LBA of an inlying block offers computational access to the data.

The mapping of blocks to buckets is used by a server implementing the iSCSI standard, called target. The own target implementation, called jSCSI [58, 80] [1] handles the iSCSI-related server/client communication. The backend of jSCSI is by default a file-based backend and extended to use jClouds directly as well as Treetank.

The mapping of blocks to data elements is represented by Figure 6.3:

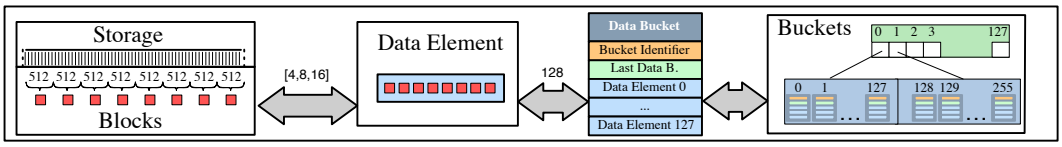


Figure 6.3: Mapping Blocks to Data Elements stored in Treetank

512 bytes are stored in one block. Blocks represent the smallest unit in storage acting as base for the proposed iSCSI mapping. 4, 8 or 16 blocks are combined in one data element. The LBA allows a direct computation of the data identifier as explained later. 128 data elements are afterwards encapsulated in one data bucket and stored in the already described bucket architecture.

Examples of the mapping of the LBA to the resulting buckets are described in Table 6.1.

Table 6.1: Mapping of $c = 8$ Blocks to Data Elements enabling iSCSI on No-SQL stores

Byte	LBA	Data Identifier	Bucket
$[0 \dots 511]$	0	0	0
$[512 \dots 1023]$	1	0	0
$\dots$	$\dots$	$\dots$	$\dots$
$[4096 \dots 4607]$	8	1	0
$\dots$	$\dots$	$\dots$	$\dots$
$[524288 \dots 524799]$	1024	128	1
$\dots$	$\dots$	$\dots$	$\dots$
$[N * 512 \dots ((N+1) * 512) - 1]$	N	$\lfloor \frac{N}{8} \rfloor$	$\lfloor \frac{N}{8*128} \rfloor$

Relying on a block size of 512 bytes per block and by clustering 8 blocks in one data element, each data bucket holds $8*128 = 1024$ blocks and therefore $8*128*512 = 524288$ bytes. The access via LBAs allows a direct mapping to each data element with the help of the data identifier as shown in Table 6.1. A clustering parameter, representing the number of blocks stored in each data element, is denoted as c in the rest of this chapter. Table 6.1 shows the relation of any LBA N to data identifiers using $c = 8$.

Each block maps a data bucket. Since $c * 128$ blocks are stored in each bucket, sequential access is only partly supported. Prefetching over Treetank is due to the linking architecture not possible since the access via LBAs is rarely predictable and highly

[1] jSCSI is freely available under http://jscsi.org.

depending on the file system applied on the block storage. The access in the storage by LBAs represents an independence of the block access to upper-level file systems. To preserve this independence, the point of creating a new version is determined in the proposed iSCSI mapping by writing a fixed amount of data.

The computed data identifier accesses its related data bucket only using pointers and offsets in linking indirect buckets. Overriding data is not possible since the append-only paradigm enforces the creation of new data buckets linked under the same offset as explained in Section 5.2.3. This mechanism prohibits fragmentation of the storage. Each overwritten block results in a new data element and therefore in a new data bucket.

The performance of the bucket architecture of Treetank is compared with a direct block/bucket mapping. A jClouds adapter is implemented relying on the mapping represented by Figure 6.4.

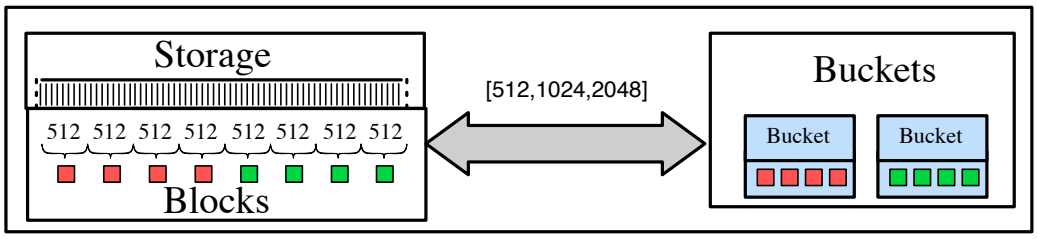


Figure 6.4: Mapping Blocks to Buckets stored in jClouds

Similar to the Treetank mapping, a variable number of blocks $c \in [512, 1024, 2048]$ is clustered in one bucket. Both mappings result in similar bucket sizes offering direct comparison of the benchmarks.

6.1.2.1 Benchmarks

The benchmark evaluating iSCSI mapped to No-SQL stores bases on 10 runs relying on the benchmarking suite bonnie++ [45]: The jSCSI target is connected from the standard Linux iSCSI initiator. The block device is mounted on a virtualized machine with 256 MB RAM. Since the file system caches access to block devices, the benchmarking data is automatically set to the double amount of RAM available. The resulting block device of 512 MB is formatted with Ext2 as file system. All benchmarks take place in a gigabit network including uplink to Amazon S3 representing an example cloud storage provider.

The evaluation use jSCSI comparing Treetank and jClouds as backend. Table 6.2 shows the applied values for c and the commit threshold of the experiments.

The results are denoted as "(AWS) [Ordered|Direct],n,m". n represents the bucket size in KiB and m stands for the commit threshold in MiB in the following Figures 6.5 - 6.9. "Ordered" describes the results of the Treetank mapping while "Direct" refers to the jClouds/jSCSI mapping. The sliding window is fixed to 4 in all benchmarks.

The number of buckets serialized is presented in Figure 6.5. The more blocks are

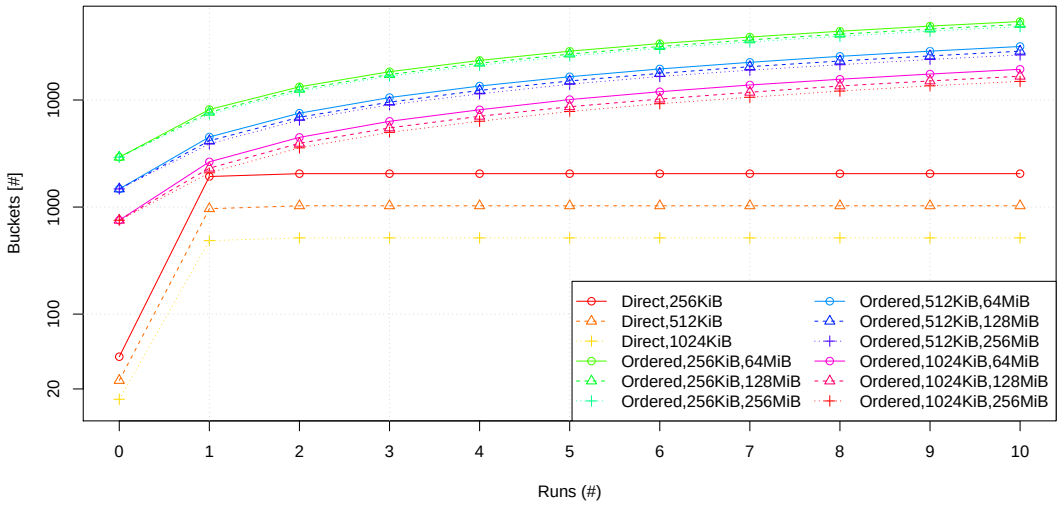


Figure 6.5: Number of Buckets storing Blocks generated by bonnie++

mapped in a bucket, the less buckets are serialized. Depending on the mapping, the number of buckets scales differently. When blocks are directly mapped to buckets, the numbers of buckets is scaling constantly. In this jSCSI/jClouds mapping, blocks might be overridden making the allocation of new buckets unnecessary. The amount of directly mapped buckets is thereby restricted by $n = 256\text{KiB} \rightarrow 2051$, $n = 512\text{KiB} \rightarrow 1027$ respectively $n = 1024\text{KiB} \rightarrow 515$.

Storing blocks in Treetank represented by the curves labeled as "Ordered", scales linear when writing data in multiple bonnie++ runs. Since no buckets are overridden, modifying existing blocks result in the creation of new buckets, satisfying the append-only paradigm. Blocks stored in the bucket architecture are physically neither deleted

Table 6.2: Bucket Sizes for the iSCSI Benchmark with bonnie++

Mapping	c	Bucket Size in Kib	Commit Threshold in MiB
Direct	512	256	-
Direct	1024	512	-
Direct	2048	1024	-
Ordered	4	256	64
Ordered	4	256	128
Ordered	4	256	256
Ordered	8	512	64
Ordered	8	512	128
Ordered	8	512	256
Ordered	16	1024	64
Ordered	16	1024	128
Ordered	16	1024	256

nor modified once serialized. The scaling bases on the number of blocks stored in each bucket and the commit rate: The more blocks are stored in a bucket, the less buckets are serialized per version. The less often a commit occurs, the lesser buckets are written. The commit rate has thereby a bigger impact than the number of blocks stored in each data element.

The performance evaluation use the same parameters presented in Table 6.2. All correlations are evaluated against a No-SQL store locally. The parameters generating the most and the fewest buckets are additionally benchmarked against Amazon S3 as cloud-based No-SQL store. Therefore, the jSCSI/jClouds mapping and the Treetank mapping are both applied in the cloud with bucket sizes of 256 KiB and 1024 KiB.

bonnie++ performs different benchmarks, all presented as boxplots showing the values of 10 consecutive runs. The first benchmark evaluates creates, reads and deletes as well as random seeks on the data measured by accesses per second. The summary is represented by Figure 6.6

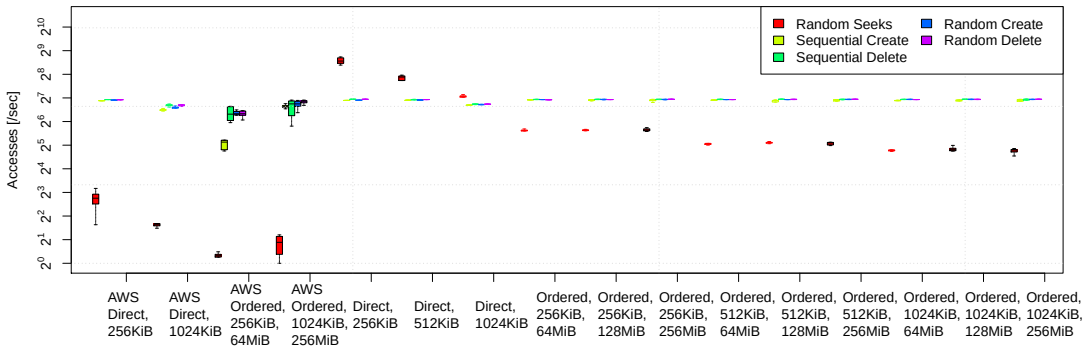


Figure 6.6: Benchmarking Creates, Deletes and Seeks with bonnie++

In this benchmark, $16 * 128$ files are created sequentially and randomly, accessed and deleted by bonnie++. The access for reads is too fast to be measured. Only results for creates, deletes and random seeks are generated.

Creates and deletes perform similar, independent from the pattern. Random seeks on the data are visibly dependent on the bucket size: The smaller the buckets are, the faster the seek is performed. The retrieval operation of a block relies on the read access on the storing bucket. The larger the bucket is, the more data must be processed to return the requested information.

Random seeks are faster when working with the direct mapping. Compared locally, the jClouds/jSCSI mapping is between 5 (when using 1024 KiB buckets) - 8 (when using 256 KiB buckets) times faster. Creates and deletes mainly rely on data stored in the inodes only. Random seeks in opposite directly access the content of files. This benchmark therefore represents an indicator about the performance loss by using the bucket architecture of Treetank. Each random access results in additional read accesses of indirect buckets. The direct mapping has the ability to access relevant blocks directly.

This impact even diminishes accessing cloud-stored buckets. The performance of creating and deleting files behaves similar in the direct mapping and the Treetank mapping. Focusing on Treetank, these operations destabilize denoted by the larger confidence intervals. The reason is again the necessity to retrieve multiple buckets for accessing the correct inode. Randomly seeking even performs worse because of the distance to the cloud: Randomly retrieving buckets just for the access on a single, inlying block makes practically a direct usage of remote buckets unusable.

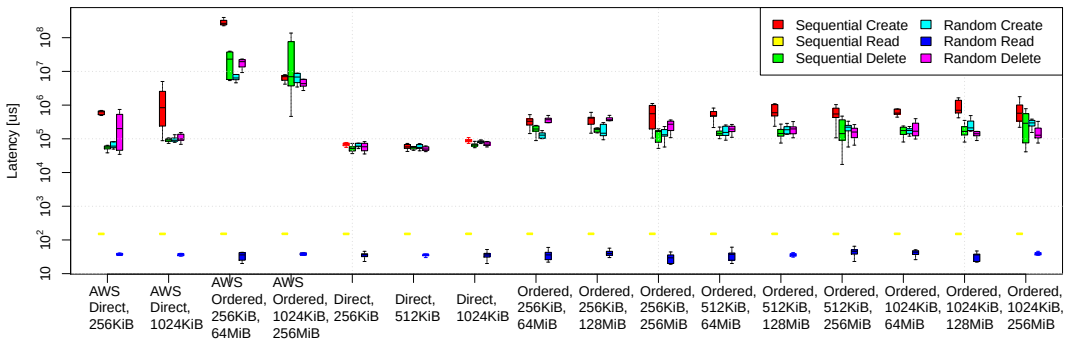


Figure 6.7: Benchmarking Latencies for Creates, Deletes and Seeks with bonnie++

Latencies for create, delete and read are represented in Figure 6.7 for random and sequential accesses. It is important to note that iSCSI is sensitive to delays when blocks become not instantly accessible. Delays result in ping requests to the target. Since jSCSI is implemented as prototype providing only fundamental iSCSI functionality, ping requests result in temporary disconnects between target and initiator. The disconnects last a couple of milliseconds and influence the latency. The direct, local mapping shows a robust performance. Using Treetank locally ends up in higher latencies, independent from bucket size and the commit threshold. The latencies occur randomly since each block access might occur as too slow for the initiator. Using the direct mapping in the cloud, the latency destabilizes the system denoted by the larger confidence intervals. The Treetank mapping in the cloud results in the highest latencies. The higher latency shows again the need for an instant access of blocks in iSCSI.

The throughput of the mappings is evaluated in Figure 6.8. The Treetank mapping and the direct mapping scale similar locally regarding char-based in- and output. Nevertheless, the Treetank mapping is more unstable than the direct mapping. Block-based access on local, directly mapped buckets transfers data with about $50\frac{Mb}{sec}$. The only exception are buckets of the size of 256 KiB, explained later in the evaluation of Figure 6.9. The throughput of the direct mapping is about 10 times faster than relying on Treetank as backend. The transfer of blocks increases slightly with larger bucket sizes since less buckets must be retrieved from storage per sequential block access. The commit threshold has no significant impact. Storing data in remote buckets, the system again destabilizes denoted by the larger confidence intervals. Storing buckets via

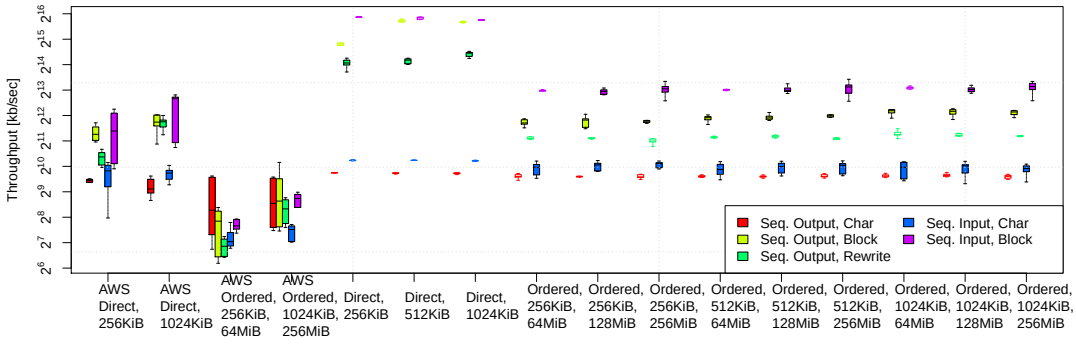


Figure 6.8: Benchmarking Throughput by Writing and Reading Chars and Blocks with bonnie++

Treetank directly with the cloud generates throughput rates with at most $0.5\frac{Mb}{sec}$. This throughput is unacceptable for block-based storage. A local mirror becomes necessary to use Treetank in practice.

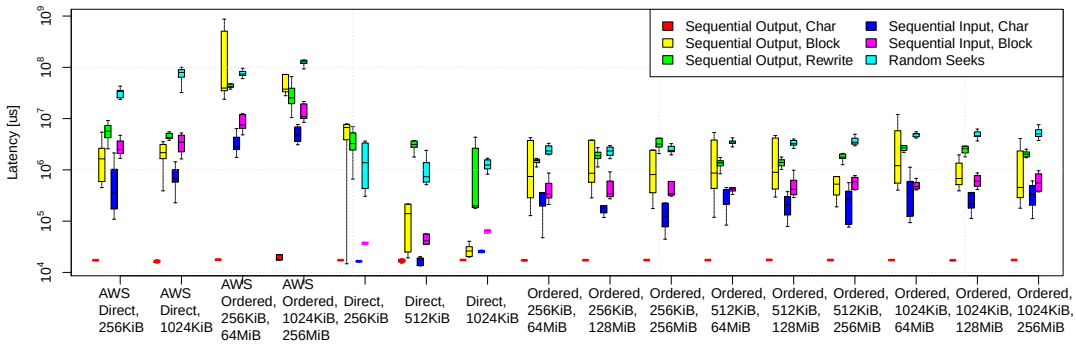


Figure 6.9: Benchmarking Latency for Writing and Reading Chars and Blocks with bonnie++

The latencies of writing chars and blocks is presented in Figure 6.9. All latencies except sequential char-based access, show unstable access by their larger confidence intervals because of the prototype status of jSCSI. Especially the direct mapping with a bucket size of 256 Kib shows the impact of ping requests to the latency. This results also in slower transfer rates in Figure 6.8. Working on remote storage by directly mapping blocks to buckets, the larger confidence intervals show again the problem of an unstable access to cloud-based data. The latency by performing random seeks consumes 60 times more time compared to a local storage. Random accesses on the blocks results in random accesses on buckets. The mapping decelerates massively the retrieval of multiple blocks.

Relying on local data only, the latency is acceptable especially when considering a full-featured target. Even the bucket architecture of Treetank offers more or less stable access independent from bucket sizes and commit thresholds.

6.1.2.2 Conclusions on the Block Mapping

Block-based communication relying on iSCSI is complex. The complexity results in a high fragility against disturbances and delays. Blocks have to become accessible instantly. The cloud-based location ongoing with the mapping results in an unfortunate combination with the prototype status of jSCSI. Delays result in disconnects and high latencies presented by Figures 6.7 and 6.9. The remote storage combined with the bucket architecture increases latencies even further. The results of the benchmarks are summarized in Table 6.3:

Table 6.3: Comparison of bonnie++ Results for the iSCSI Mapping

Mapping	Location	max. Throughput, Block: $\frac{kb}{sec}$	min. Throughput, Block: $\frac{kb}{sec}$	avg. Throughput, Block, $\frac{kb}{sec}$	max. Latency ms
direct	local	60874	9688	50771.88	34908
direct	remote	22656	958	4389.125	100000
Treetank	local	10994	2695	6220.294	34294
Treetank	remote	1141	73	331.475	871000

Char-based access performs similar with different mappings on the different locations. The focus of comparison lies thereby on block-based access only. The average throughput scales significantly worse when relying on remote data. A direct storage for blocks in the cloud is practically impossible.

Local caching of data is inefficient since no predication about the access on buckets can be made. Nevertheless, the architecture offers the ability to synchronize to the cloud asynchronously. jSCSI and Treetank ship with local backends. The open source architecture of both libraries supports the binding to additional backends. The cloud is thereby used for archiving purposes mainly. Such an usage is backed by the sliding versioning. A client holds for example double the size of the sliding window locally. All data is furthermore pushed in the cloud. A hybrid backend, relying on asynchronous pushes, would scale similar than the local backends offering secure storage of blocks in buckets.

6.1.3 Including Files

Storing files in the cloud is from a user's perspective the most common use case. Products satisfying this use case are described in detail in Table 3.1. All products act as watching service in the operating system. Bound to a defined folder, file-modifying events are communicated to a cloud client. These events include creation, deletion, modification and access of a file. The client maps the modifying events to the related files and pushes these changes in the cloud. The push takes place asynchronously representing the main difference to block-based access. Since all data is accessible in local folders, retrieval of data from the cloud only occurs while bootstrapping or restoring data.

The mapping of files to Treetank uses these watching services as well: Bound to a folder, all files in a folder are mapped to data buckets, encrypted, equipped with a checksum and pushed in the cloud. Mapping files to buckets is done straight-forward represented by Figure 6.10.

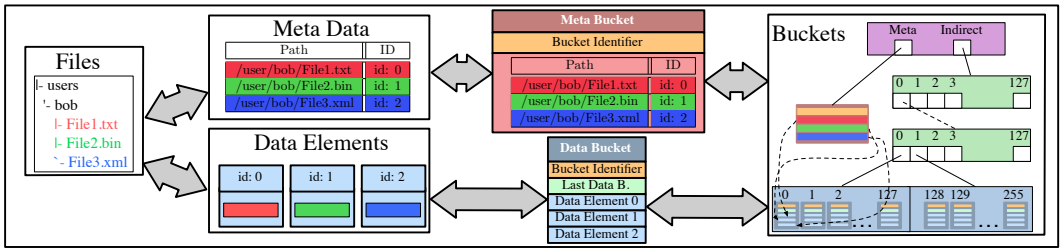


Figure 6.10: Mapping Files to Data Elements stored in Treetank

Single files are determined by their path and stored in their binary representation. The bytes are directly stored in a data element. To balance the resulting buckets, each data element is trimmed to a maximal size of inlying bytes. Optionally, a link to the next data element is set, if the size of the file to be stored is too large to fit into one data element. The data identifier of the data elements are incrementally generated and referenced over the meta buckets. A map links the path as key to the data identifier of the storing data element. In Figure 6.10, "File1.txt" is mapped to the data element with the data identifier 1 in the meta data. Access to single files via Treetank takes place over the meta bucket. Accessing this mapping, each data element is referenced represented by the dotted lines in Figure 6.10.

Events like file creation, file deletion, file modification and file access act as input for the file mapping. The first three events push data in the cloud triggered by a commit in Treetank. Binding commits directly to these events offers a persistent traceback of all occurred events on the watched folder. While creation events and modification events result in the generation of new buckets, deletion events just remove related entries from the meta bucket.

The straightforward mapping of files to buckets uses meta buckets to map file paths to data identifiers. The resulting transparent and low-overhead usage of Treetank is shown by the following results.

6.1.3.1 Benchmarks

Benchmarking the file binding to Treetank is difficult due to the asynchronous data transfer: The watching service takes events from the operating system resulting in non-blocking transfers of the data. Comparison against existing clients is furthermore hardly possible since the closed source architecture of their software prohibits customized hooks to measure storage and retrieval effort of the client. The events fired by the file system are additionally quite simple representing only a marker on the file touched. The mapping of the events to Treetank is thereby defined as follows.

- Creational events insert a new file in the described architecture. Inserts are performed straightforwardly: New data elements are allocated. The content of the file is stored in the data elements. The path is inserted as mapping into the meta bucket of the current version including the data identifier of the storing data element.

- Removal of already written data is not possible in Treetank. Removal of data would violate the append-only paradigm. Deletion events thereby simply result in the removal of the related entry from the meta bucket. Missing the reference, the retrieval of this file is not possible in future versions.

- Modifications are only pointing to a file without providing in-depth knowledge of the delta occurred. Treetank thereby does not benefit from these events treating them like creational events.

- Retrieval of a file over Treetank is only necessary upon restoration. The restoration includes thereby retrieval of the buckets and the reconstruction of the file out of the storing data elements.

The benchmarks focus on the creation and the retrieval of files only. Modification events are handled the same as creational events. Removal events only delete the related reference from the meta bucket resulting in an instant operation. All benchmarks hook directly into the client to override the asynchronous data handling. In each bench run, 100 files are written to Treetank or retrieved from Treetank. Each insertion ends up in a new version. The files thereby consist of different sized random data ranging from 256 KiB to 32 MiB.

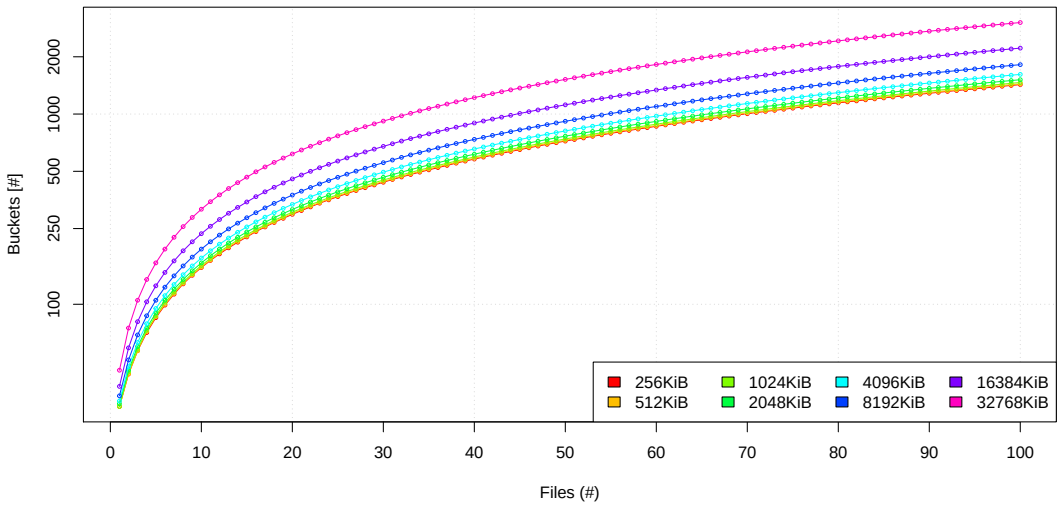


Figure 6.11: Number of Buckets storing 100 Files of different Sizes

Before evaluating the performance, the number of serialized buckets is evaluated by Figure 6.11. The number of resulting buckets scales linear with respect to the stored files. The larger the files are, the more buckets are allocated. The capped capacity of 16 KIB of the data elements forces Treetank to split the data into multiple data elements. Each insertion results in a new version including the creation of new data buckets and indirect buckets, if necessary. The scaling of the new buckets to be created by each insertion is represented by Table 6.4.

Table 6.4: Increment of Number of Buckets storing multiple Files with different Sizes in Treetank

File Size	Bucket Number (Start)	Increment (avg. increment)
256 KiB	29	14.133
512 KiB	29	14.255
1 MiB	29	14.510
2 MiB	30	15
4 MiB	31	16.010
8 MiB	33	18.031
16 MiB	37	22.061
32 MiB	45	30.122

The increment differs not only because on the different file sizes. Two insertions of the same file result in a different number of buckets allocated. The numbers of buckets to be written cover the generation of the indirect buckets and data buckets as well. The creations of indirect buckets occur thereby irregularly. If the amount of pointers in the indirect buckets exceeds 128, new indirect buckets must be generated. This results in a slightly different amount of buckets generated in this version. The 15 buckets needed for writing one 2 MiB file represent an exception. Relying on the defined maximum data size of 16 KiB per data element, exactly 128 data elements are allocated. As a result, each file results in one data bucket. Since the benchmarks inserts only 100 files, no additional indirect buckets are allocated. The increment is fixed to $14 + 1$ buckets by inserting 2 MiB files represented a standard normal version in Treetank similar to the example in Section 5.2.3. The additional bucket stores the bucket identifier of the most recent uber bucket serialized. As a result, for file sizes less than 2 MiB, not every version results in the creation of $14 + 1$ buckets. Often, new files fill up preceding data buckets up to 128 data elements. The increment of buckets thereby scales always with the file sizes: Starting from 2 MiB, one additional data bucket per version is needed for double the amount of data.

The resulting buckets from writing and reading files are stored locally as well as on Amazon S3 presented by Figures 6.12 and 6.14. Each bench run inserts 100 files of the same file size ranging from 256 KiB to 32 MiB.

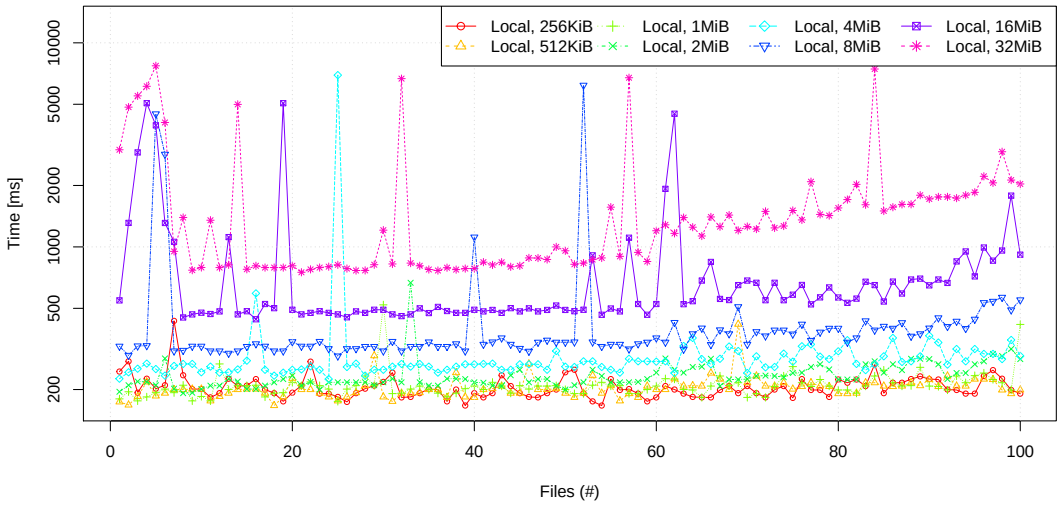


Figure 6.12: Benchmarking Writes for the File Mapping in Treetank stored locally

The caching of the operating system becomes visible by writing buckets in a local file system. Figure 6.12 shows furthermore the impact of Treetank as overhead. The peeks represent the intermediate persisting of the buckets occurring from time to time. Besides these peeks, all file sizes scales similar upon a file size of 4 MiB. The scaling afterwards is roughly linear with respect to the file size.

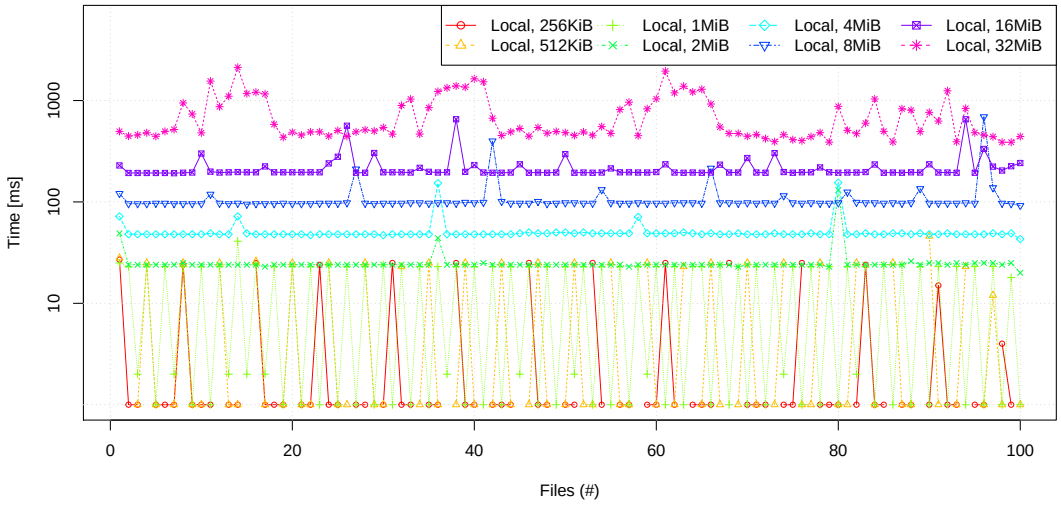


Figure 6.13: Benchmarking Reads for the File Mapping in Treetank stored locally

By writing the buckets to Amazon S3 directly, represented by Figure 6.14, the file system cache has less impact. For small file sizes up to 512 KiB, no significant difference is visible. Partly written data buckets are locally cached in the transaction log of Treetank. This behavior is equivalent to the integrity benchmarks in Sections 5.3.1 and 5.3.3. Referring to Table 6.4, two files fit into one data bucket for file sizes up to 512 KiB. Starting 1 MiB, partly new data buckets must be created for each file resulting in the gab between the curves. Focusing on larger file sizes, the curves scale depending on the size of the file inserted.

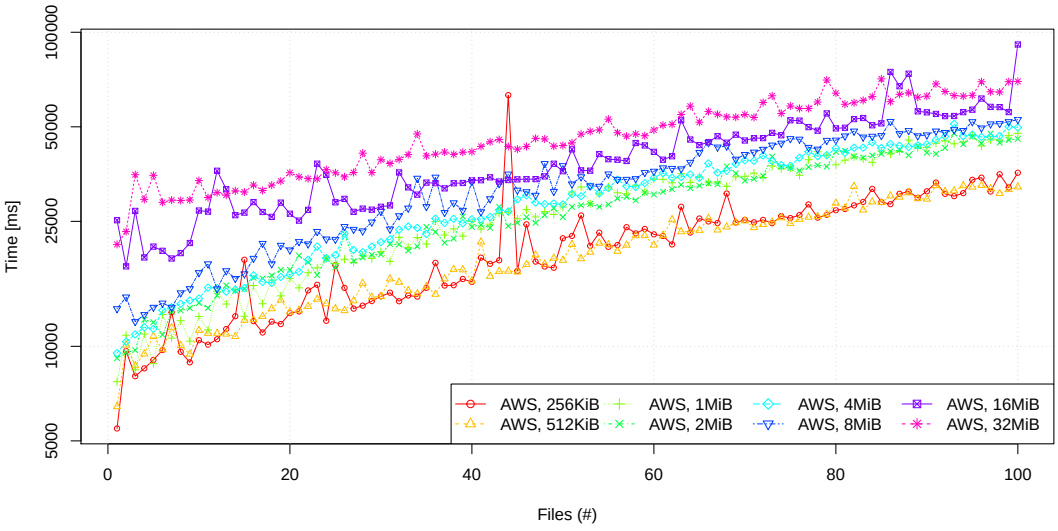


Figure 6.14: Benchmarking Writes for the File Mapping in Treetank stored on AWS S3

Reading is evaluated in Figure 6.13 and Figure 6.15. In these benchmarks, 100 files of the same file size are read from a local and a remote AWS S3 storage. The file size ranges from 256 KiB to 32 MiB.

The impact of the fixed defined maximal size of each data element again becomes visible. For larger files with a size of 2 MiB and larger, the retrieval time from a local store is more or less constant as shown in Figure 6.13. Each access to a file is in need of at least one bucket, which was not accessed by retrieving any former file. The retrieval of these files from Treetank therefore always scales constantly. The impact of the defined maximum size of data elements becomes visible by evaluating file sizes of 1 MiB and less. The bucket containing the last file also holds data for the current file to be retrieved. Therefore, the read results in partly low access times. Roughly every second read needs to access the disk when retrieving 1 MiB files. Then, a new bucket containing the next two files must be read from storage. The access becomes less regular by accessing files with a size of 512 KiB. The persistent storage is accessed all 4 files. By reading 256 KiB files, a new data bucket need to be accessed by reading 8 files.

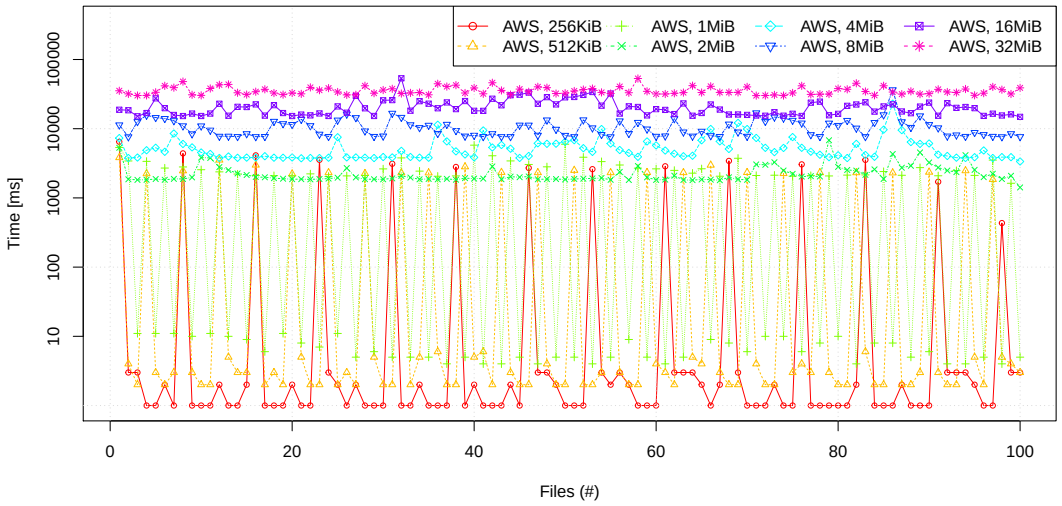


Figure 6.15: Benchmarking Reads for the File Mapping in Treetank stored on AWS S3

This same rates apply for the access on the AWS S3 store presented in Figure 6.15. For file sizes of 2 MiB and bigger, no caching applies. Each file maps to at least one data bucket. The scaling is constant for these files and respects the size of the data to be retrieved. For lower data sizes, the impact of caching again becomes clearly visible. The retrieval of 2 files results in the necessity to read 1 data bucket for a file size of 1 MiB. The rest of the scaling is equivalent to the local storage.

6.1.3.2 Conclusions on the File Mapping

The straightforward mapping of files to buckets in Treetank results in a feasible architecture. Leaned on clients like Dropbox or Google Drive, Treetank enriches the file data with checksums, encryption and versioning. The defined size of 16 KiB per data element restricts the size of data buckets to 2 MiB. Unbalanced buckets are thereby prohibited when it comes to different file sizes. The price for the gained stability is a performance difference. When it comes to small files, local caching enables faster access on files stored in the same data buckets. Since the data mapping takes place on a higher abstraction level than blocks, the mapping enables finer-granular data handling. In the file mapping, the data elements are more bound to access patterns increasing the performance. In opposite to the block-based mapping, the caching of data buckets has thereby an higher impact. From an end-user's perspective, the usage of the file mapping is preferable over the mounting of iSCSI targets mapped in Treetank.

6.1.4 Storing native XML

The versatile architecture of Treetank originates from its former usage as native, versioned XML storage, developed by Marc Kramis while doing his PhD. Even though the project left the database area, the mapping for storing nodes natively still exists. Enabling scalable structural modifications in the tree, a localized encoding is achieved by relying on the tree structure of XML [38]. Each node stores pointers to its neighborhood representing a localized encoding. The advantages, disadvantages and discussions are not the focus of this thesis but are partly described in [60]. Figure 6.16 shows the mapping of an example XML to Treetank.

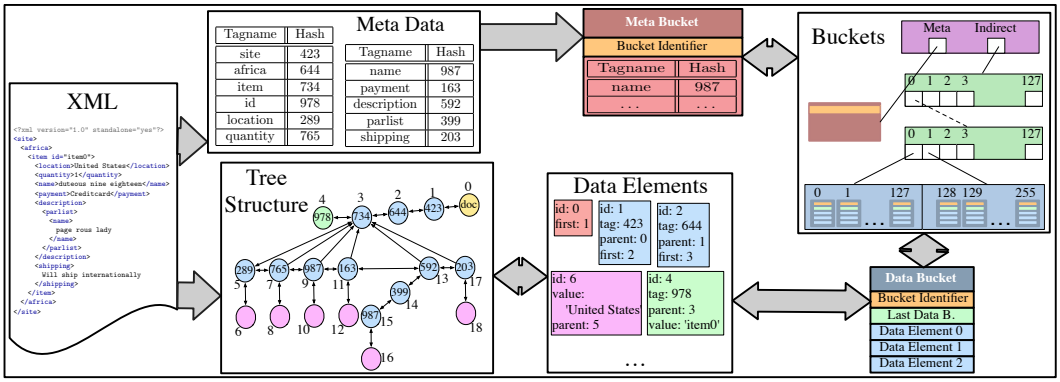


Figure 6.16: Mapping XML to Data Elements stored in Treetank

The inlying structure of a XMark [102] document is imported as a tree. The different node types are mapped to different implementations of data elements. The blue elements in the tree structure in Figure 6.16 are tags representing the backbone of the tree. These data elements store pointers to their parent, their first children and their left and right sibling. Tag names for these elements are handled in meta buckets reducing the consumed storage. The nodes with the data identifier 9 and 15 therefore point to the same hash value referencing the same tag name. The number in the elements denoted the tag names mapped in the meta bucket. The linking to their neighbors in every node relies on the data identifiers received from Treetank. Other node types are text nodes (represented by the pink nodes), attribute nodes (denoted as green nodes) and the document root node, always receiving the data identifier 0.

Any XML file can be inserted in Treetank. Besides, Treetank comes with a partly implemented, native XPath 2.0 [29] engine which stays out of focus of this work. Furthermore, Treetank comes with a binding to Saxon [17], a freely available, external XPath/XQuery processor. XML stored natively in Treetank is further accessible as RESTful resource [53] described in detail in Section 6.1.5.

6.1.4.1 Benchmarks

Benchmarking the XML mapping relies on serialization and deserialization of nodes only. Relying on a local encoding, no tree order can be derived from the data identifier of single nodes: Inserting an entire document into Treetank indeed results in nodes receiving their data identifier in pre-order at the beginning. Modifications however might insert new nodes anywhere in the structure. No reliable assumption mapping the position of nodes in buckets to their position in the tree can be made. Benchmarks are therefore only performed locally. The independence between access patterns and data identifiers, similar to the block mapping, slows down a remote retrieval of nodes from the cloud.

The benchmark of the XML mapping inserts documents into Treetank multiple times. Each insertion results in a new version. Each version is afterwards serialized. Figure 6.17 shows the results of these processes.

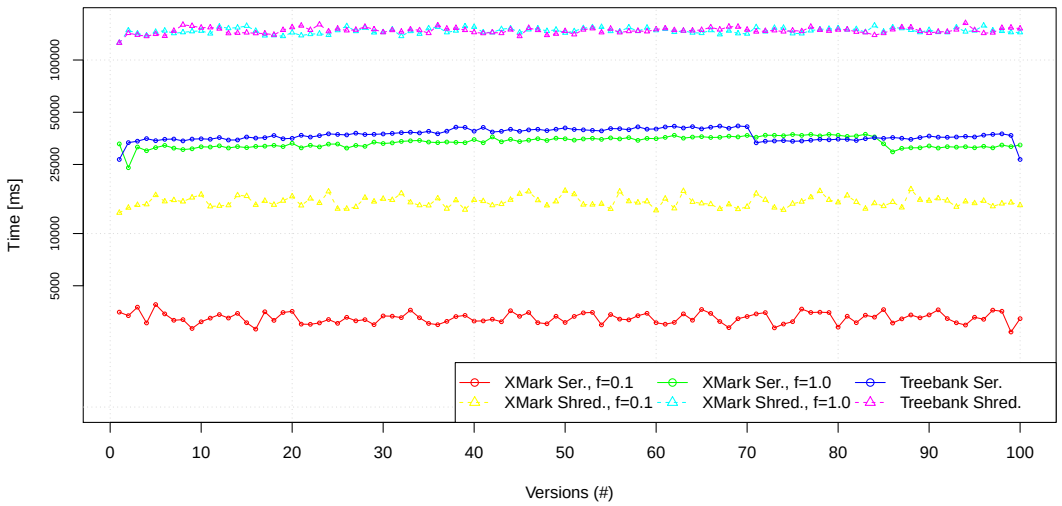


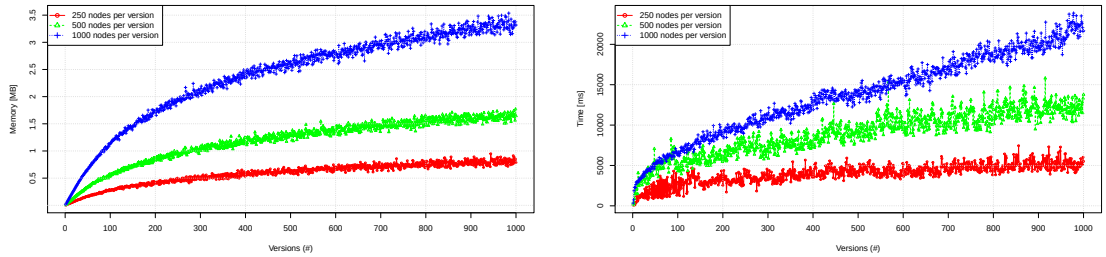
Figure 6.17: Benchmarking the XML mapping by inserting and retrieving XMark and Treebank into Treetank

Two XMark [102] documents are inserted with different document sizes. Additionally, the Treebank [86] corpus is loaded as well. XMark is a widely used XML generator, representing a state of the art benchmarking suite for XPath. The parameter $f \in [0.1, 1.0]$ results in two XML instances with the sizes 112 MB $\leftarrow f = 1.0$ and 12 MB $\leftarrow f = 0.1$. The resulting XMark contains document-centric as well as data-centric areas in the tree. Treebank [86] represents a linguistic corpus and thereby a classic document-centric document with an irregular structure and a file size of about 82 MB.

The aim is to benchmark the access on the data for all versions. Therefore, the document is deleted after each insertion and inserted again. Independent of the structure but depending on the file size, the insertion scales constantly. The same amount of

buckets is allocated in all versions. Therefore, the insertion time differs not significantly by creating a new version. The serialization scales constantly as well. The access to the data relies on the access on a version represented by a revision bucket. All revision buckets are reachable by traversing a fixed number of indirect buckets. The retrieval time is therefore independent from the version.

The adaptability of the node encoding is evaluated by manual inserting nodes represented by Figure 6.18.



(a) Random Insert of Nodes, Memory

(b) Random Insert of Nodes, Time

Figure 6.18: Evaluation of Random Insert of XML Nodes in Treetank

In this scenario, tag nodes are inserted in the tree randomly. The insertion takes place either as first children or as right sibling. After each insertion, the insertion point for the next insert is chosen randomly in the tree. To show the impact of commits, the commit threshold is set to 250, 500 and 1000 nodes. The insertion continues until 1000 versions are created.

This benchmark aims to show the adaptability of the tree structure. The position of the changes in the tree is unpredictable. Nevertheless, the local encoding allows linear insertion time with respect to the number of inserted nodes. By each insertion, only a constant number of adjacent nodes must be adapted. The independence of the data identifiers against the tree order nevertheless results in an increasing retrieval effort of the data buckets. Since the pointers of the neighborhood must be adapted, these nodes must be retrieved, modified and serialized again. The consumed memory shown in Figure 6.18a represents the amount of storage consumed by the application. This consumption scales better since already instantiated nodes are reused for adapting the neighborhood.

6.1.4.2 Conclusions on the XML Mapping

Treetank, originated from the area of native XML storage, relies on a local encoding to store XML as tree structure. The local encoding allows direct, localized and versatile modifications in the tree. The mapping to buckets including the underlying versioning mechanism supports data element-aware traceback of changes. Insertion operations scale linear with respect of the time. Retrieval and storage of new versions consumes constant time independent from the accessing version.

The XML mapping stores tree structures natively. This furthermore enables secondary, structure-depending integrity structures. Links to siblings as well as to parent nodes enable the efficient maintenance of a Merkle-Tree on the XML structure [88]. Even though Treetank contains integrity checks using its bucket architecture, these additional checksums support higher-level service layers. Related approaches protecting XML combined with REST and Merkle-Trees are developed in the context of this thesis [57, 63]. Offering protection against race conditions in REST, checking checksums on the fly still respects its stateless paradigm.

6.1.5 Mapping REST Services

Web services represent the backbone of modern distributed architectures. Requests and data is transferred mainly over HTTP [52]. REST [53] represents a simple yet elegant way to define those services. All interaction with the data takes place over HTTP verbs. The architecture of the uniform resource locator (URL) is extended optionally by parameters.

$$\underbrace{GET}_{Verb}\underbrace{http://host/data}_{Resource}\underbrace{?language=en}_{Parameter} \tag{6.1}$$

An operation ("GET" in this example) is applied on a resource accessed over an URL. The access can thereby optionally be equipped with parameters. Parameters enable operations directly applied on the resource before returning the result. The independence of representations of the resource makes use of the parameters such as the choice of language in the example of Equation 6.1. REST furthermore represents compact definitions of operations and their interaction with resources. Resources accessed by "GET" requests must never be modified. The stateless communication of HTTP is furthermore respected at all time. REST provides light-weight definitions and inherit most practical functionality directly from HTTP. Widely used, REST represents a modern opponent to heavy-loaded web services such as SOAP [37].

Treetank exposes nodes stored by the XML mapping as REST services. Figure 6.19 shows the usage of trees stored in Treetank as REST services.

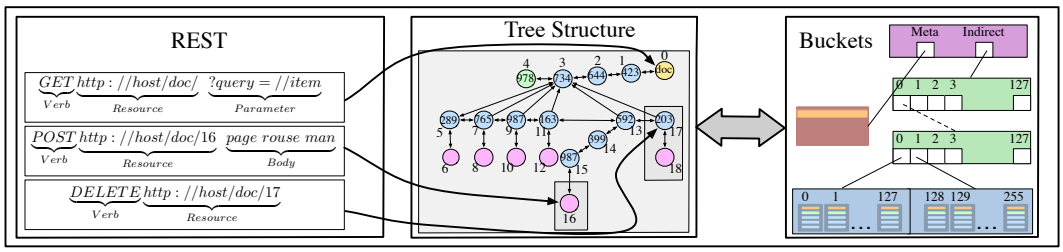


Figure 6.19: Mapping REST services to Data Elements stored in Treetank

Relying on a localized web server, REST requests become applicable on the tree. Since the data identifiers are persistent and consistent over all versions, nodes can be directly accessed as resources. The document itself is associated with the root node. Interpreting the node as resource, XPath queries can be applied as parameters. The independence of the representation furthermore allows transformations with the help of XSLT [44]. Direct changes on elements are offered by setting new content on resources. In the example of Figure 6.19, new content is posted to the text node 16. All HTTP functionalities are directly mappable to any resource represented by nodes. Deleting nodes is performed by removing the node including its subtree. In the example of Figure 6.19, the node with the data identifier 17 is deleted by a REST request.

The underlaying mapping is adjacent to the XML mapping shown in Figure 6.16. Consequently, no extra evaluation of this layer is provided.

6.1.5.1 Conclusions on the REST Mapping

This REST mapping relies directly on the XML mapping. Nodes are interpreted as direct allocable resources. Optional, structure-based checksums in the tree [57, 63] protect nodes as REST resources against concurrent accesses and race conditions. Working directly with XML as resource, convenient XML-relying operations become applicable as parameter. The direct resource allocation of single nodes by their data identifier further enables REST-based access even to sub-structures.

This intersection of REST and XML results in a reference implementation of the described REST binding called JAX-RX [62]. This project was developed with the Database and Information Systems Group at the University of Konstanz.

6.1.6 Conclusions

The mapping of data containers to data elements stored in Treetank offers policy-based enforcement of commonly defined security measures. Even though the abstraction of concrete storage operations in the cloud makes fine granular performance evaluations impossible to achieve, the benchmarks of the provided implementations evaluate the cloud as black box. The variety of benchmarks proves thereby, that the application of the mappings is practicable.

Independent from the abstraction layer, all data not only becomes storable in cloud-based No-SQL structures. Gaining protection against unauthorized data modifications and accesses, all data still stays accessible by their container-specific interfaces. The independence gained by the amorphous bucket architecture results not only in the ability to apply additional byte-modifying techniques like error correction or encryption in the serialization process. The buckets are furthermore pushed over an extensible abstraction layer in the cloud offering even the definition of own No-SQL stores relying on photo sharing websites as described in the next Section.

6.2 Defining your Cloud Provider

As shown in the previous section, Treetank has the ability to map different data containers to buckets. This flexibility is also provided at the backend. Technically, as already mentioned in Section 3.2, buckets are just binary large objects referenced by unique keys. The resulting storage structure is widely known as key/value store. Figure 6.20 shows the example buckets of Section 5.2.3 in their key/value representation.

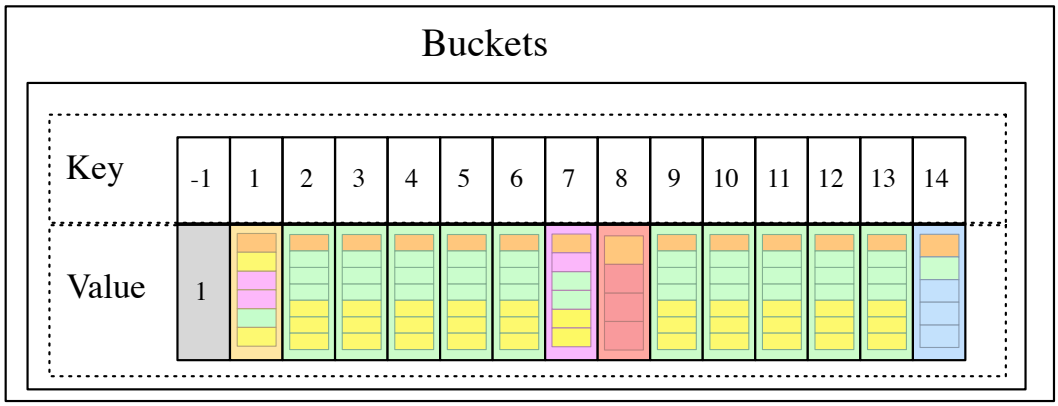


Figure 6.20: Key/Value Representation of Buckets generated by Treetank

No-SQL stores offer the ability to handle a vast amount of key/value tuples. The tuples are billed by consumed storage, HTTP requests and traffic. Specialized cloud services such as photo sharing websites are commonly free of charge. Their data is directly usable by the hosting provider and binds end-users to the products of the cloud storage provider.

Independent from the stored data, all API-based transfer normally takes place over REST. Data is interpreted as resource directly allocable with the help of HTTP.

Relying on professional No-SQL stores, each provider defines its own data handling. The results are e.g. different authorization mechanisms or URL-layouts. Table 6.5 shows examples for accessing URL layouts for Google Cloud Storage, Microsoft Azure and Amazon S3.

Table 6.5: Example REST dialects for No-SQL Stores

Provider	URL
Azure	http://<account>.blob.core.windows.net/<container>/<bucket>
AWS S3	http://<container>.s3.amazonaws.com/<bucket> http://s3.amazonaws.com/<container>/<bucket>
Google Cloud Storage	http://storage.googleapis.com/<container>/<bucket> http://<container>.storage.googleapis.com/<bucket>

The different URLs are not only domain-specific. Variants in the URLs of the cloud storage providers make cloud-specific bindings from the application site necessary. To prohibit vendor lock-in and to reduce implementation overhead, the key/value store

accessed by Treetank uses an abstraction layer. This layer translates REST requests to a Java API. The API guarantees not only independence from cloud storage providers but enables also easy balancing and storage of the data in a cloud-of-clouds.

Furthermore, the API not only provides flexible access to multiple clouds. Own cloud stores can be implemented as backend. One implementation uses service clouds such as photo sharing websites as No-SQL stores. The gained No-SQL access thereby not only becomes accessible for Treetank only: Every library using the storage provider over jClouds gains access to the complimentary stores.

The following approach bases on joint work with Wolfgang Miller and Marcel Waldvogel resulting in the technical report *Utilizing Photo Sharing Websites for Cloud Storage* [64]. Parts of the implementation rely on Wolfgang Miller's thesis focusing on the image generation. A technical report [64], including all graphics and results partly reused in this chapter, offers an in-depth evaluation of different photo sharing websites and image generators. Therefore and for reasons of brevity, the focus of this chapter is the plain usage of photo sharing websites as No-SQL stores.

6.2.1 Evaluating the Costs

The market for direct allocable No-SQL stores remains small. Complex pricing models, where a small change in usage makes a big difference in price, are common. Content-specific storages like photo sharing websites such as Yahoo's Flickr, Google's Picasa-Web, or Facebook are typically excluded as storage backends. However, they not only provide large storage space and high availability for free or cheap. Their underlying infrastructure offers similar characteristics as No-SQL stores such as scalability, availability and API-based access. Facebook for example stored 260 billion photos in 2010, representing 20 PB of data where 60 TB of new photos were uploaded in one week [28].

Figure 6.21 shows the price per GB per month comparing Facebook, Picasa[1], Flickr[2] against AWS S3[3] and Dropbox[4].

This chart represents only an approximation of the costs. Effects not included in Figure 6.21 are requests and traffic, extra billed in AWS S3. Furthermore, restrictions on photo sharing websites like fixed image sizes to make use of free storage plans are ignored. Two aspects are, even without any in-depth analysis, obvious: First, cloud services building on top of each other are billed in a way that the customer always pays. The costs for Dropbox, relying on AWS S3 as backend, are always higher than the direct usage of AWS S3 up to a quota of 500 GB. Second, photo sharing websites are massively cheaper than direct allocable data storages. Picasa and Facebook use the sharing of photos for gathering and selling information about user behavior including social interaction profiles. Summarized, the costs for storing pictures in the cloud are significantly less than using professional data storages.

[1] `https://support.google.com/picasa/answer/1224181?hl=en&ref_topic=30179`, as of September 2013

[2] `http://www.flickr.com/help/limits/`, as of September 2013

[3] `http://aws.amazon.com/s3/pricing/`, as of September 2013

[4] `https://www.dropbox.com/pricing`, as of September 2013

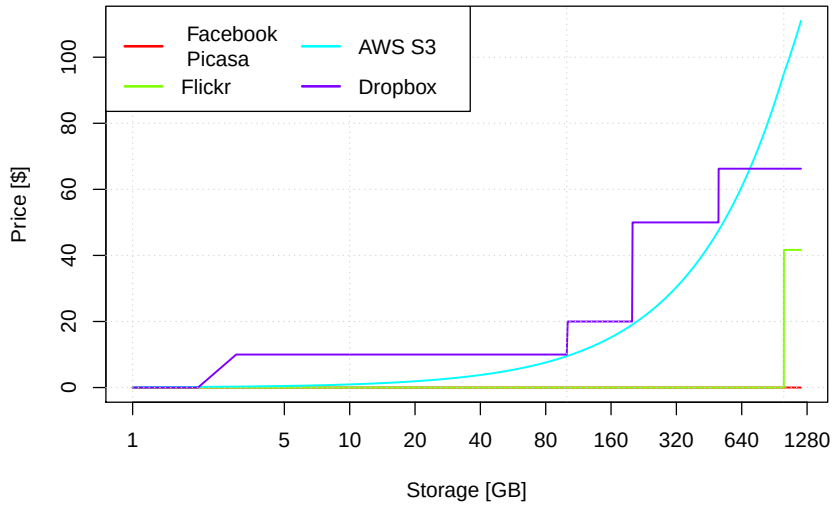


Figure 6.21: Comparison of Prices of No-SQL stores, Photo Sharing Websites and file-based Cloud Storages

6.2.2 Flexible Access to different Clouds

Enforcing abstracted access to the cloud is crucial for guaranteeing availability: Distribution approaches like DepSky need such an abstraction for non-vendor specific access to the different clouds. Treetank and its storage mechanism therefore use an entirely abstracted backend described in Figure 6.22.

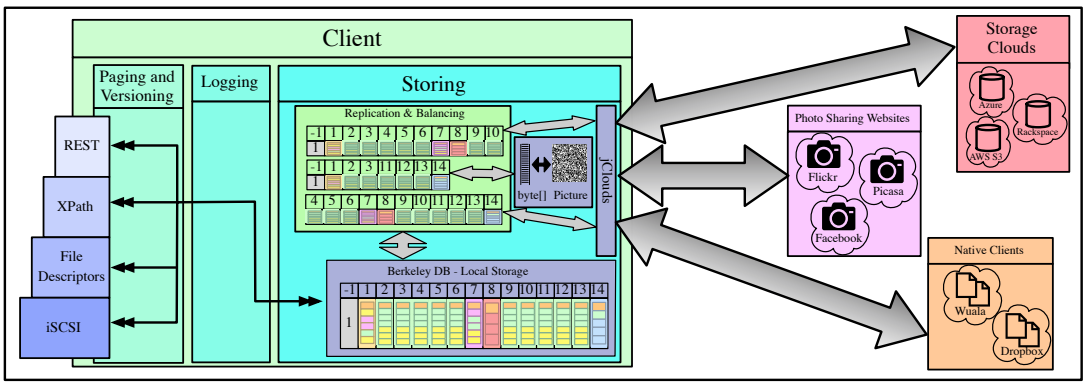


Figure 6.22: Binding to Treetank to different Cloud Stores

The key/value tuples generated by Treetank are locally persisted in a Berkeley DB (BDB) [94]. Open Source, implemented in plain Java and representing one of the first

No-SQL stores, the BDB allows flexible implementations of keys and values. Index structures allow fast access on keys. A variable number of versions leaned on the sliding window is stored locally improving read/write accesses. The evaluation of the different mappings shows the necessity for a localized mirror of the data. The iSCSI mapping for example becomes practically unusable when relying on remote stored buckets only, as presented by Section 6.1.2.1. The append-only paradigm of the storage including sliding versioning allows a flexible delegation of single versions in the cloud. Treetank offers striping of buckets in three different types of clouds presented in Figure 6.22.

Native Clients: Native clients like Dropbox or Wuala are accessed as local folders. Treetank stores the buckets as files in these folders.

Storage Clouds: Cloud-based No-SQL stores like Amazon S3 are accessed directly over REST.

Photo Sharing Websites: Photo Sharing Websites are used as No-SQL storage by encoding buckets in images. The access on the images occurs over REST.

The mapping of buckets to photo sharing websites is described in detail in Section 6.2.2.1. The idea is to generate images out of the bytes and use the APIs of the photo sharing websites to upload the buckets encoded in images.

6.2.2.1 Encapsulating Data in Images

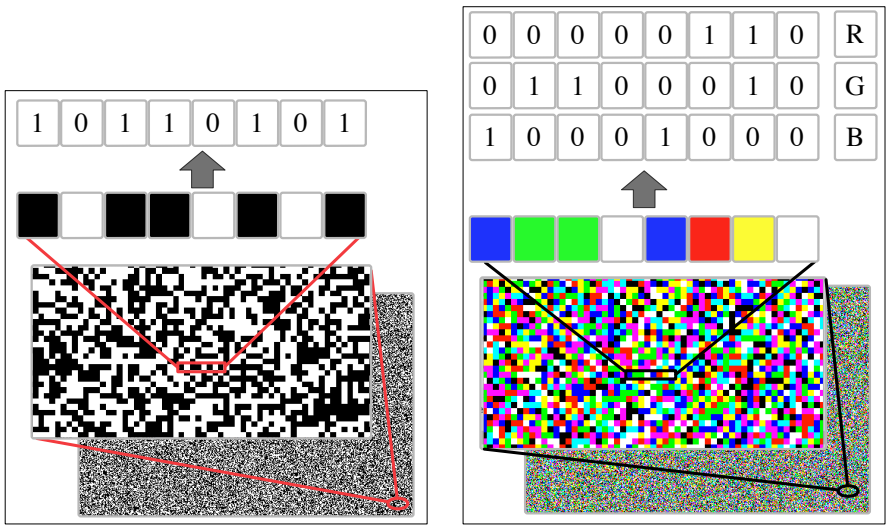
The main idea behind the usage of photo sharing websites is the generation of images out of arbitrary data. Data is interpreted binary-encoded represented only by boolean values. The values are mapped to different colors depending on the encoding. The resulting colors are applied on single pixels, representing the atomic units in an image. Figure 6.23 shows the resulting two encoding approaches.

Interpreted as single layer, Figure 6.23a maps the two extremest colors, black and white, on single pixels. Each pixel represents one bit resulting in a data rate of $\frac{1}{8}\frac{\text{byte}}{\text{pixel}}$. Since all colors are interpreted as one single data range, different areas of this range are mapped on a variable numbers of bits resulting in a extremely robust approach against color-based compression. This approach is denoted as single layered approach (SL approach) in the rest of the chapter.

Another approach using colors is represented by relying on the different layers in the RGB color space as denoted in Figure 6.23b. Each of the RGB-layers is seen as one layer eligible for storing single bits. By applying two values per layer per pixel, a data rate of $\frac{3}{8}\frac{\text{byte}}{\text{pixel}}$ is achieved. Since each component of the RGB color space is seen as one single value range, the resulting three values are mapped to each other. This approach is denoted as multi layered approach (ML approach) in the rest of the chapter.

6.2.2.2 Increasing the Data Rate

Storing additional values in each pixel increase the data rate. Both approaches are extensible by adding more values per pixel. A higher data rates is thereby paid by



(a) Encoding Booleans in Images **(b)** Encoding Boolean in RGB layers

Figure 6.23: Encoding Data into Images for using Photo Sharing Websites as No-SQL Stores

higher fragility against compression. The number of applied values per pixel (by the SL approach) or per layer (by the ML approach) is denoted as $\gamma \in [1 \dots 256]$. Equations 6.2 and 6.3 shows the amount of necessary pixels for encapsulating one byte related to the SL approach and ML approach:

$$p_{SL} = \lceil \log_{\gamma}(256) \rceil \tag{6.2}$$

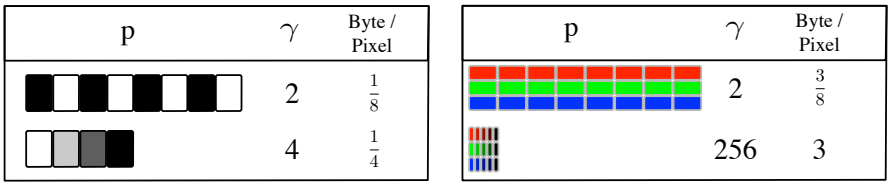
$$p_{ML} = \frac{\lceil \log_{\gamma}(256) \rceil}{3} \tag{6.3}$$

The denoted equations restrict useful choices for γ. The range of $\gamma \in [16 \dots 255]$, for example needs always 2 pixels to represent one byte by the SL approach. Choosing any value of $256 > \gamma > 16$ results in more information per pixel without any improvement of the data rate. As a result, only values for $\gamma \in [2, 3, 4, 7, 16, 256]$ are beneficial. These values offer the best data ratio for the applied number of values per pixel.

6.2.2.3 Adjusting to different Photo Sharing Websites

Three of the biggest photo sharing websites are evaluated as No-SQL stores namely Picasa (hosted by Google), Flickr (hosted by Yahoo) and Facebook. All of these photo sharing websites offer API-based access enabling automatic uploading/downloading operations. Flickr and Picasa offer access to the original uploaded images making techniques against compression errors obsolete. Data stored on Picasa and Flickr is encapsulated with the ML approach with $\gamma = 256$. 1 pixel stores 3 bytes.

Facebook compresses images in the uploading process. The applied compression falsifies the stored colors. The usage of ML approaches is thereby not possible, requiring the fall back on SL approaches.



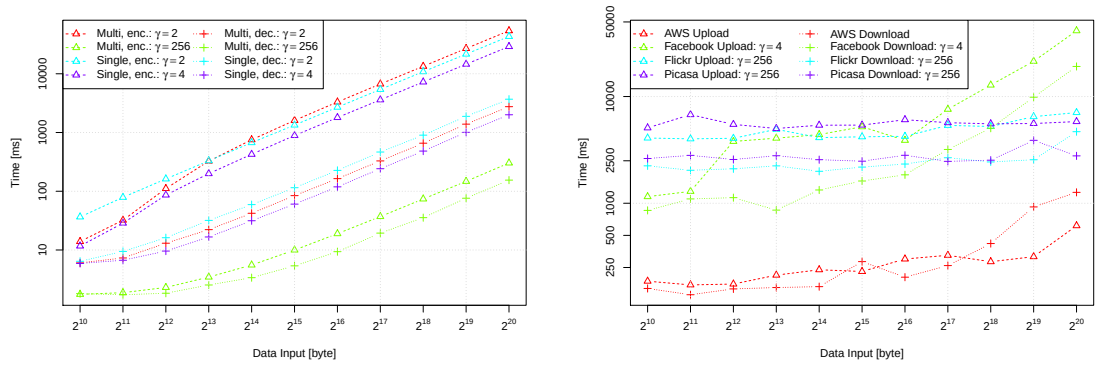
(a) SL encoders **(b)** ML encoders

Figure 6.24: Data Ratio of the Encoding of Data in Images

Figure 6.24 represents the both encoding approaches. The first encoders in Figures 6.24a and 6.24b with $\gamma = 2$ refer to the examples in Figure 6.23 acting as reference. The SL approach with $\gamma = 4$ is applicable on Facebook. This encoder needs 4 pixels to store one byte. Since Picasa and Flickr allow the access to original uploaded images, the ML approach with $\gamma = 256$ is used. 256 different values per component are stored in each pixel resulting in 3 bytes per pixel. The experimental setup leading to this choice is documented in the technical report [64].

6.2.2.4 The costs of free Storage

The idea to use free available, content-bound storage as No-SQL stores sounds promising. The following benchmarks show that the price for free storage comes at a cost: The access to the data is significantly slower as shown in Figure 6.25.



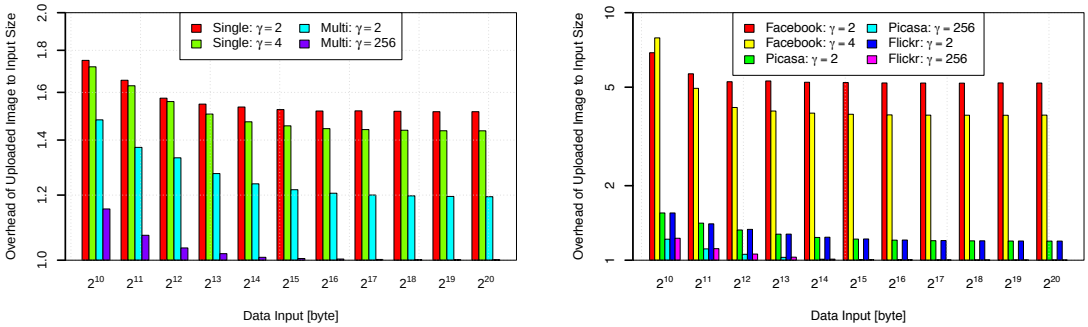
(a) Encoding/Decoding Time **(b)** Upload/Download Time

Figure 6.25: Performance of Image Generation and Interaction with Photo Sharing Websites

The encoding of data into images consumes additional time compared to the access to professional No-SQL stores. Figure 6.25a represents the time necessary for encoding and decoding data ranging from 1 KiB to 1 MiB. The encoding, relying on write accesses on the disk, consumes always more time than the decoding. Decoding data only needs to read data. Images are faster created and accessed with the ML approach and $\gamma = 256$ than with a lower γ or by using the SL approach. Higher data rates result in smaller images to be accessed. Besides, the creation of SL images takes less time than the creation of ML images with $\gamma = 2$. Obviously, the effort needed to encode colored images consumes more time compared to the generation of SL images. All encoders scale linear with respect to the input data as expected.

To evaluate the performance in the context of professional cloud storage, the encoders are bound to jClouds and compared to AWS S3. The comparison of the upload and download is represented by Figure 6.25b. This benchmark includes the encoding and decoding of the data before uploading and after downloading. Amazon S3 performs best with fast access times. Accessing buckets with a size of at most 2^{17} takes less then 300 ms for downloading. For an increasing amount of data, the upload and download time to Amazon increases linearly. The performance needed to store data at Picasa and Flickr takes more or less constant time but scales significantly worse than using dedicated bucket stores. For 1 MiB buckets, the overhead of storing the data in images lies between 2.197 ($\frac{t_{down}\text{Picasa}}{t_{down}\text{AWS}}$) and 5.594 ($\frac{t_{up}\text{Flickr}}{t_{up}\text{AWS}}$). The local encoding and decoding effort represents thereby not the bottleneck. The photo sharing websites have to handle the incoming images instantly. The acknowledgment of requests thereby slows down the transfer rates. The time consumed by Facebook therefore increases with the file size. Since no original images are stored, incoming images must be compressed and handled on the fly. The performance therefore restricts a direct usage without local caching.

The performance is also influenced by the sizes of the images themselves. An overhead is generated between the file size of the image and the inlaying data relying on the encoding. Figure 6.26 presents the comparison of the overhead.



(a) Relative File Size Overhead, Local

(b) Relative File Size Overhead, Download

Figure 6.26: Overhead of Images compared to Data Sizes

Figure 6.26a represents the overhead by generating images only. The ML approach with $\gamma = 256$ performs best as expected. The overhead is hardly recognizable. The lower the data rate of an encoder is, the more the overhead increases. The SL approach with $\gamma = 2$ thereby has the largest overhead.

Figure 6.26b shows the overhead after downloading the generated images from the photo sharing websites. When accessing original data stored on Flickr and Picasa, the uploaded and downloaded images show only negligible differences. As soon as compression is applied like in Facebook, the downloaded image is even larger than the uploaded image. The increased overhead results in the transfer of more data. The files downloaded from Facebook are significantly larger. The downloading and decoding performance is thereby influenced by the overhead as well.

Summarized, photo sharing websites are usable in real life scenarios as storage backends. The bad performance is acceptable for non-blocking accesses. Nevertheless, encoded images are easily identifiable. Hardening encoding approaches against compression as well as auto-recognition from the site of the photo sharing websites stayed out of focus of this work. Countermeasures protecting automatic generated images thereby represent future work areas worth to be evaluated.

6.2.3 Conclusions on the Storage of Buckets on Photo Sharing Websites

Users are confronted with a fragmented market, inflexible pricing, and compatibility issues when relying on different cloud storage providers. Projects such as jClouds [13] are addressing compatibility issues by providing an uniform interface. Since jClouds is extensible by providing own cloud bindings, photo sharing websites are used as No-SQL stores. The usage of these content-specific clouds breaks open the fragmented market. Offering high available storage for free, images are generated automatically out of data representing buckets. The specific characteristics of different photo sharing websites are respected in the implementation resulting in different encoders. The automatic access via APIs to these image portals is now provided by jClouds enabling the usage of Flickr, Picasa and Facebook as storage backends.

6.3 Conclusions

Treetank represents, as shown in this chapter, a pipeline for storing different data containers securely on various cloud storage.

- The entry point to Treetank and therefore to the entire pipeline are stackable data containers. Blocks, files, XML and REST services are bound to Treetank showing the feasibility and drawbacks of the proposed architecture. Even though each data container is implementable in Treetank as proven by applying different third-party benchmarks, some data containers needs direct, continuos instant access to the underlying buckets like e.g. iSCSI accessing single blocks.

- The exit point of Treetank is represented by jClouds. Offering flexible access to several clouds by providing a common API speaking different REST dialects, jClouds enables Treetank not only to avoid any vendor lock-in. The extensibility of jClouds is shown by the prototyped usage of photo sharing websites as No-SQL stores. Images, representing single buckets, base on an adaptable data encoding. Even though the access on photo sharing websites performs worse than on traditional cloud stores, photo sharing websites are well suited as archiving storage thanks to their often complimentary offers.

In summary, this system enforces integrity and accountability based on the bucket structure and the applicable versioning. The independence from the data container as well as to the No-SQL store seconds the appliance of Treetank as an API for storing data securely in the cloud. Even though the proposed measures enforce secure data storage, confidentiality stayed out of focus so far. All buckets stored by Treetank are encrypted, motivating a key handling without restricting collaborative workflows. An approach for managing keys in a flexible manner, relying on versioned data as well as the high availability of cloud-based services, is therefore described in the next Chapter 7.

Two can keep a secret if one is dead.

Unknown

7 Flexible Key Management for a Versioned Cloud

Contents

All-time availability, ubiquity and vast resources are major arguments for using cloud storage for collaboration and sharing. Secret sharing among a fixed defined set of individuals is a known problem illustrated by the quote at the beginning of this chapter. Protecting confidentiality by encrypting data combined with sharing mechanisms is even more challenging, since hiding and sharing information are conflicting. The following problems need be solved.

1. Accessing parties must be flexible definable by any user including user- and group-based shares.
2. Managing keys for storage needs techniques to handle backward secrecy. Data encrypted with former key material needs to be accessible.
3. Supporting collaboration on versioned data needs time-based access orthogonal to content-based access.
4. Creation of new shares and removal/modification of existing shares results in new key material. The cloud is barely trustworthy for key exchange even though key propagation benefits from its high availability.

5. Computational power in the cloud can not be trusted. The key handling has therefore rely on the SaaS-level only, obviating any computational updates on the key material in the cloud.

Relying on common approaches from the area of stream encryption [31, 69, 112, 113], a key management is described and evaluated offering time-restrictive access for versatile groups. The cloud is thereby used as propagation platform without harming confidentiality. Guaranteeing scalability for a variable number of groups and versions, the following approach not only offers key management for cloud-stored data but makes direct use of the advantages of the cloud namely all-time availability and ubiquitous accessibility.

This chapter is based on joint work with Patrick Lang, Stefan Hohenadel and Marcel Waldvogel resulting in the paper *Versatile Key Management for Secure Cloud Storage* [61]. Graph-theoretical foundations described in an extended version of the paper but excluded from this chapter, are contributed by Stefan Hohenadel. The results mainly base on the thesis of Patrick Lang and are reused. The rest of the paper represents my own contribution.

7.1 Background

Most applications simply encrypt the data to protect the information against unauthorized access. Satisfying common understandings of confidentiality, encrypting works well for a limited amount of accessing users due to the necessity to share a common secret. The availability of modern mobile devices, enabling access of cloud-based data from anywhere, intensifies the sharing for disjoint users. The resulting flexibility results in collaborative workflows where different users work on common data. Shared secrets neither offer secure ways to support such workflows nor make use of the availability and scalability of the cloud. Changes in the set of authorized clients result in complex re-encryption operations and the distribution of new shared secrets. Versioning of the data provided by multiple cloud storage providers further complicates the key handling. The access to specific versions has to rely on additional shared secrets.

7.1.1 Contribution: Versatile Distributed Key Graph

The described approach tackles the challenge of managing access rights on shared versioned data with the help of the following techniques.

- Disjoint clients share data using hierarchical access rights. The hierarchy relies on a *Directed Acyclic Graph*(DAG) where *Encryption Keys*(EKs) represent group keys. EKs summarize disjoint clients denoted as *Client Keys*(CKs).

- Updates on the keys occur encrypted and scalable using well-established approaches from the area of stream encryption.

- Access rights are applied to any data stored in past versions, the current version or future versions.

The approach consists of three components. A global key graph is stored on a trusted third party environment. Encrypted updates on the key graph are stored in the cloud. Each client stores sub graphs of the key graph denoting specific access rights.

The key graph relies on existing graph-based key management approaches namely *VersaKey* [112] extended as a DAG similar to [117]. This approach binds key material to nodes representing the DAG. The source nodes are represented as CKs in the key graph. They constitute the client rights. The terminal nodes represent the most common access rights. Despite current approaches, the EKs are not only used to offer scalability. They also provide group-based access.

A second adaption on *VersaKey*, besides the usage of a DAG with semantic EKs instead of a tree, includes the persistence of the updates applicable on the key graph. These updates are denoted as key trails. They are not only broadcasted to the clients once. Instead, they are stored in the cloud for on-demand updates of the client keys as well. Similar to *VersaKey*, the nodes are versioned. Each version of each node contains unique key material to decrypt an element of the versioned data.

Any modification of the key graph results in an update of all reachable EKs. The update originates from the adjusted access right and includes the generation of new key material for these nodes. The resulting key trails are consisting of the fresh key material encrypted with the related valid nodes. The number of key trails scale with the number of updated nodes. Each key trail relies on an edge incident to the updated nodes. Instead of updating all shared keys, only the summarizing groups are updated.

By introducing virtual nodes for functional combination of groups, the scaling regarding the evolving key management is constant. The number of key trails scale with the number of updated nodes instead of the number of incident edges. The *VersaKey* approach is thereby extended to version the nodes providing temporal-aware access rights.

By applying stream-based key management to versioned data, well-established graph-based key managements are extended. The generation of encrypted key updates allows the storage of key trails in the cloud parallel to encrypted data. The storage and handling of the keys and the key trails in the cloud needs no computation power. The same bucket-based storage paradigms used for data storage are also applied to the key material. As a result, NoSQL stores are sufficient for offering convenient sync-and-share functionality.

7.1.2 Existing Approaches

Related approaches in this area cover the scalable handling of keys with the help of key graphs. Sato [100] proposes a trust model for secure cloud usage. This model describes no concrete key management. Damiani and Pagano [46] propose an hierarchy of keys used for encrypting and storing data in the cloud. Existing users are excluded by propagating

new keys and re-encrypting the data. Xu [118] uses public/private keys and makes collaborative key handling obsolete.

Storage upon untrusted components always needs sophisticated approaches to grant disjoint access on common data. No information about the underlying group management should thereby escape. *Cryptree* [67] represents an approach to store data in an hierarchy with permission rights mapped on subtrees. The underlying recursive data structure scales with the number of keys since the keys are inherited top-down in the tree. The focus of *Cryptree* is similar to *VersaKey*. Hierarchical group permissions are combined with an hierarchical data structure.

Versakey [112] denotes the base for the own approach. Hassen [70] proposes an extension by introducing intra-level changes on the tree. This approach enables key graphs to change the affiliation of a node to a group.

The EKs in these approaches ensure scalability of join-/leave-operations of clients. The resulting structure is not a tree anymore but a DAG as proposed in [109, 117]. Even though the scaling of the DAG degenerates in consecutive changes on the keys, the combination of nodes using a more efficient DAG representation as described in [121] stays out of focus. The hierarchy of the key graph is used as semantic representation for organizational issues.

7.2 Key Management for Cloud Storage

Figure 7.1 shows a DAG constructed similar to classic stream-bound approaches [109, 117, 121].

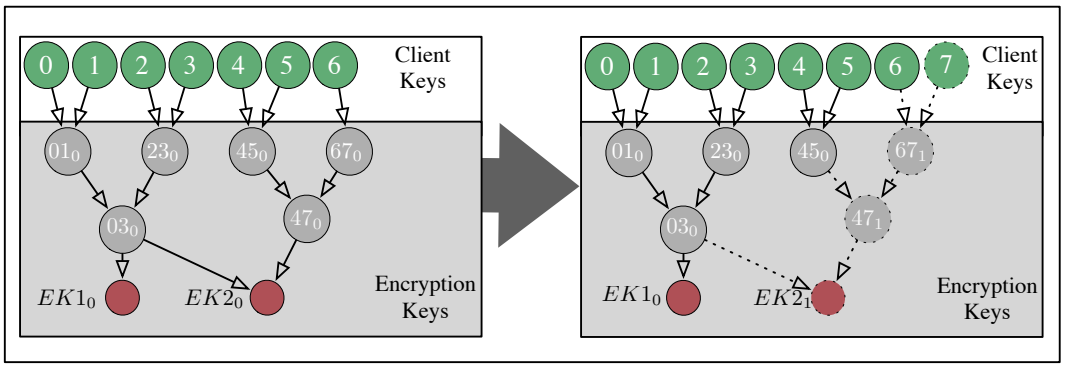


Figure 7.1: Classic Key Graph used for Stream Encryption

Any data is encrypted with the help of EKs or CKs. To ensure scalable updates, CKs are combined with the help of the EKs. Each client contains a sub-graph consisting of its own CK and the descendants. A global key graph manages join-/leave-operations of nodes as well as insert-/remove-operations of edges. If a client, represented by a CK, joins or leaves the set of authorized clients, only parts of the keys stored in each client must be updated. These parts include the descendants of modified nodes.

Figure 7.1 shows the insertion of the client 7. As a consequence, the nodes 67, 47 and $EK2$ have to update their key material. The new client has afterwards the ability to access the data encrypted by the descendants of its CK. Since each node contains a version counter, represented by the number in the subscript of the actual node identifier, the version of the updated nodes increases.

Relying on this graph representation, *VersaKey* encrypts the new key material of the updated descendants with the keys stored in the adjacent nodes staying valid after the modification. These encrypted updates, represented by the edges in the key graph, are called key trails. For each dotted line in Figure 7.1, one key trail is computed as update e.g. $E_6(67_1)$ where the updated node 67 is encrypted with the already existing CK 6.

7.2.1 Key Graphs and Data Storage

VersaKey is originally applied to stream-based architectures making access to former encrypted data not necessary. Backward secrecy is as a result not supported. Regarding the usage of evolving key graphs for encrypting data in storage, four adaptions must be made to apply *VersaKey*.

- The stored data to be encrypted must be hierarchical organized. Due to the inheritance of access rights in the key graph, any data structure relying on a hierarchy as well, such as a file systems or XML, provides the ability to encrypt different levels of data with related access rights.

- Stream-based encryption abdicates the availability of former keys and former data. The keys as well as the encrypted data are only valid at a given point of time. Changes in the key management result in a versioned key graph as well. Versioning of the data allows a mapping between different status of the data and the key graph.

- The key graph is versioned equivalent to the versioning of the data. Since the key material changes by adapting access rights, all former keys from the key graph must stay available ensuring access to all former versions of the data. After each update on the key graph, the modified nodes are therefore persisted as a new version.

- The updates on the key graph occur over key trails. Relying on the versioning of the key graph, the key trails are not only used for on the fly adapting of the local key graphs only. They also are used as delta between two versions on the key graph. The set of key trails must therefore respect the version of the nodes to modify. The encrypted nature of the key trails is used to store updated key material in the cloud as explained in Section 7.2.2.

The defined adaptions result in a versioning of the key graph independent from the versioning of the data. The data is encrypted with the key material of the most recent

version of a suitable node. The decrypting of the data is provided by the version of the corresponding node at the point of time of modification. Old versions of the data are thereby only decipherable with fitting versions of related nodes. The current version of the key graph nodes encrypts nevertheless ongoing modifications. Because of the binding of node versions to data versions, re-encrypting becomes unnecessary. Figure 7.2 shows an evolving key graph, the related key trails and a hierarchical data structure.

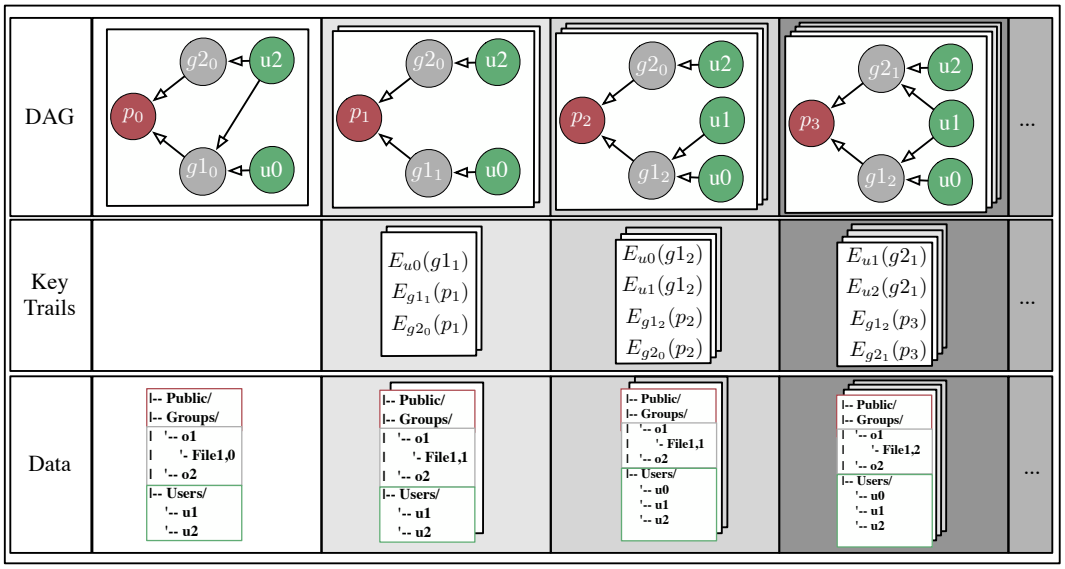


Figure 7.2: Evolving Data, Key Trails and DAG

The versions of the nodes relate to the key trails applied. The updated nodes rely on the points of insertion/removal of edges/nodes in the key graph: All nodes reachable from the node inserted/removed must update their key material. Figure 7.2 shows the leave of the CK $u2$ updating the same nodes like the insertion of the new CK $u1$. The version of the nodes $g1$ and p are incremented twice. The join of the existing CK $u1$ to the group $g2$ however results in the update of the nodes $g2$ and p. $g1$ stays thereby unmodified. The updated nodes receive new key material and an increased version. The old versions of the nodes stay accessible containing their original key material and the corresponding pointers to other related nodes in their specific version.

Independent from the updates of the key graph, the data is modified as well. The keys on higher levels in the key graph encrypt higher-level elements in the hierarchical data. Mapping both hierarchies to each other generates a benefit leaned on *Cryptree* [67]. The encryption of data uses the current version of the nodes in each modification. Figure 7.2 shows the exclusion of CK $u2$ from the group $g1$. If the exclusion occurs before the modification of the file "File1", the EK used for encrypting the new version of "File1" is relying on the new version of the node $g1$. If the modification on the data occurs before the update of the key graph, $u2$ has still access to the related version of $g1$ and therefore "File1". The challenge to access former versions with new access rights

is described in detail in Section 7.3.

All modifications on the key graph result in the generation of the key trails based on the set of reachable edges. The key trails represent retrievable deltas replaying any changes upon the key graph. All key trails are therefore persisted. Each key trail consists of a plain readable part and an encrypted part. The plain readable part hosts the relevant information for identifying adjacent nodes including their version. The *BaseID* and the *Base Version* identify the node and its version with which the key trail was encrypted. The *KeyID* and the *Key Version* represent the updated node. As a result, the *BaseID* represents u while the *KeyID* represents v of the edge $\langle u, v \rangle$. For example, in key trail $E_{g1_2}(p_3)$, $g1$ is the *BaseID* in the version 2 while p represents the *KeyID* in the version 3. Despite the plain part needed for the identification of the key trail, each key trail contains encrypted data storing the secret material for the updated node.

Related to the example represented by Figure 7.2, CK $u1$ gains access upon insertion to the most recent EK p by decrypting the key trail $E_{u1}(g1_2)$. With the resulting access on the most recent node version of the node $g1$, the key trails $E_{g1_2}(p_2)$ gives access to p_2. By adding $u1$ to group $g2$ as well, the key trail $E_{u1}(g2_1)$ grants access. The updated p_3 is afterwards either accessed via $E_{g2_1}(p_3)$ or $E_{g1_2}(p_3)$. With the help of the key trails, any version of the key graph can be reconstructed. Depending of the key trails generated, versioned-based access can be granted. A possible key trail $E_{u1}(g2_0)$ would result in the access to data encrypted with a former version of $g2$. Each client has thereby access to a substructure based on the CKs and the key trails provided.

7.2.1.1 Virtual Nodes

The key trails scale with the number of incident nodes. The in-degree of the updated nodes therefore plays an important role related to the scaling of the number of key trails. A join/leave operation incident to a node with a high fanout results in the generation of key trails for all other children nodes. The number of key trails should thereby be restricted. To restrict the number of key trails, a threshold t for an allowed maximum of incoming edges on any node is defined.

t is applied as follows: By inserting a node, the number of incoming edges $\#e_v$ is checked and compared with t. If the number of edges after the insertion exceeds t, namely $\#e_v + 1 > t$, an extra node, without any semantic purpose, w is inserted to compensate the in-degree. w is denoted as virtual node.

Virtual nodes are contained in the set of nodes in the graph and formally not distinguished from EKs but do neither encrypt any data nor represent semantic groups. Instead, virtual nodes are used to act as proxy between their parents and their children. The key trails rely on the set of edges incident to all updated nodes. Relying on the capping of the in-degree of all nodes, the number of key trails scales with the number of updated nodes and not with the number of incident edges any more.

Figure 7.3 represents the usage of virtual nodes where a new node 7 is introduced in the DAG with $t = 4$. The current nodes associated to $g2$ are after the modification referring to the virtual node $v1$. The generated number of key trails ongoing with the insertion of node 8 decreases from 8 (without virtual nodes) to 6. With the help of

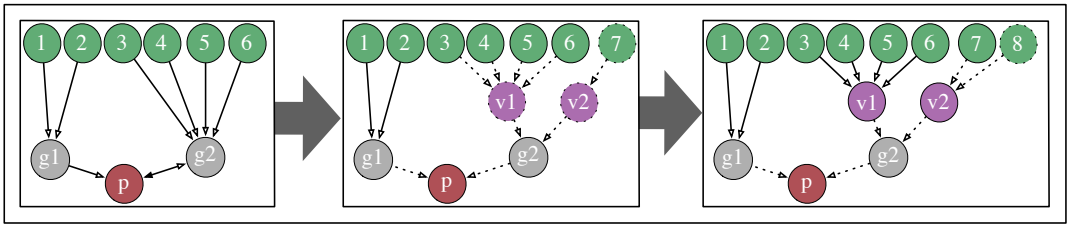


Figure 7.3: Inserting Virtual Nodes in DAG

the virtual nodes, the number of generated key trails scales with the number of the updated nodes and not with the number of incident edges. Even though virtual nodes have the same layout than other nodes in the DAG, their key material is, due to their fanout-reducing purpose only, used for encrypting related key trails.

7.2.2 Synergies between the Cloud and the Key Management

The high availability of the cloud offers easy propagation of key material even though the cloud should be seen as untrustworthy. The key graph, the data and the key trails are therefore distributed over different components.

- The key management is provided in two types of key graphs.
  - A centralized instance stored on a trusted component represents the overall key graph. This key graph contains all nodes in all versions and triggers all changes on the authorized client-set.
  - Additionally, each client holds its CK including all descendants in all accessible versions.
- The key trails are stored on the cloud. Due to the high availability of the cloud, the key trails stay accessible for any accessing client even if the centralized key manager is not available. Since the key material in the key trails is encrypted, the cloud storage provider has no ability to access any encrypted data in the cloud. The cloud stores only the updates and offers easy propagation of the key trails to the clients. Each client is able to gain new versions of nodes if suitable, decipherable key trails exists for the client.

Figure 7.4 shows the appliance of the example of Figure 7.2 in the cloud. A centralized key manager maintains the complete key graph representing all access rights for all data. Any changes regarding the client set such as joins and leaves occur in the key manager. Since the key manager adapts the changes in the client set on demand, the availability of the key manager is not necessarily guaranteed. Only when adaptions on the key graph occur, the key manager must be accessible. After each update of the central key graph, the key manager pushes the resulting key trails to the cloud.

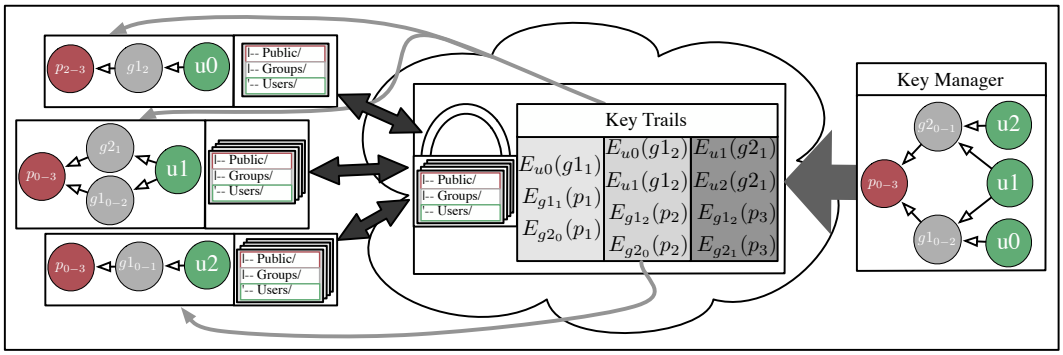


Figure 7.4: Key Management in den Cloud

Due to the availability and the scalability, the cloud has the ability to store all key trails including all versions. Since the key trails are encrypted, the cloud storage provider has no ability to decrypt them neither to access any data with the help of the key trails. Each client accessing a fixed version of the data retrieves the suitable version number and identifier for the corresponding node. If the client has access to this node but not in the suitable version, the related key trail is transferred to the requesting client. The key trails are not versioned on the client since they only represent an encrypted delta on the key graph for replaying any changes on the subgraphs in the clients.

The data is stored encrypted in the cloud denoted by the locker in Figure 7.4. Since the encryption takes place before transfer, the confidentiality of the data is provided while the data is in transfer and in storage. After each data modification, the version number of the accessing node is checked to be up to date. If the version is outdated, the related key trails are transferred to the client and are decipherable if the client was not excluded in the overall key graph. Since each modification on the data results in a unique version, the version is mappable to exact one version of one node in the key graph.

7.3 Version Access

Even though the modifications on the key graph are independent from the modifications on the data, one node of the key graph in one version must map to one data element in one version. Depending on the granted access rights, a client might have the non-exclusive access on former versions of the data, on the current version of the data or on future versions of the data. Access to former versions of the data is provided by sharing the related node including the relevant key trails. Since one EK in one version has the ability to encrypt multiple modifications, consecutive modifications might be guarded by only one key. Access to upcoming modifications is thereby equivalent to *VersaKey*: A suitable CK is inserted and linked to the EKs representing the related rights. All modifications encrypted with the descendants of the CK are afterwards accessible for

this client. Although the access to preceding and succeeding versions is given, an adaption must be made to provide access only to the current version of the data. Since one version of an EK has the ability to secure multiple versions of the data, an additional restriction of the EKs must be provided. If a client should gain access only to the current status without the ability to read former versions, simple sharing the related node violates this restriction.

Figure 7.5 describes this problem: Consecutive modifications on the "File2" are encrypted with one version of $g2$. Using one EK to protect multiple versions, the sharing of this key results in the access of all versions. A CK joining $g2$ in its first version automatically has the ability to access all versions of "File2". The classic *VersaKey* mapping does not support an access to only version 2 of the data. The access to current versions without exposing past versions encrypted with the same node therefore represents a problem to be solved.

Figure 7.5: Mapping Versioning of Keys to Versioning of the Data

Due to the independence of the key graph versioning and the versioning of the data, simple sharing an older version of the fitting EK without additional restriction might breach the confidentiality of former versions. Furthermore, the access to the data can not be provided since the last modification on the requested data can be encrypted by any former version of the key graph. Two solutions are therefore eligible to solve this problem: The first solution consists of redundant data and additional keys resulting of a shadow structure of the data and the keys as described in Section 7.3.1. The second solution uses the distributed architecture as described in Section 7.3.2.

7.3.1 Shadow Structure

The first extension restricting access to only the recent version inserts shadow structures partly mirroring the key graph and the data. The shadow structure of the key graph needs to be versioned similar to the key graph itself. The shadow structure of the data mirrors the data only in its most actual version. All modifications are thereby encrypted by the suitable node in the most recent version and stored in the versioned data storage. Additionally, the modification is also encrypted by the same node of a shadow key graph called shadow key and applied to an non-versioned shadow structure called shadow data. The shadow key is provided as additional key material stored in each node of the DAG to access only the shadow data. The shadow data consists of only the most recent version of the data. Each new client joining the DAG thereby maps to the newest version of

the key graph and the first version of the related shadow key. With the help of the applicable key trails on the shadow key, the new client has the ability to decrypt all data in the shadow data. Since the shadow data consists of only the current version of the data, no access to former versions is provided. If a client is excluded, the shadow keys are adapted similar to the normal EKs prohibiting any access to newly modified content in the shadow data.

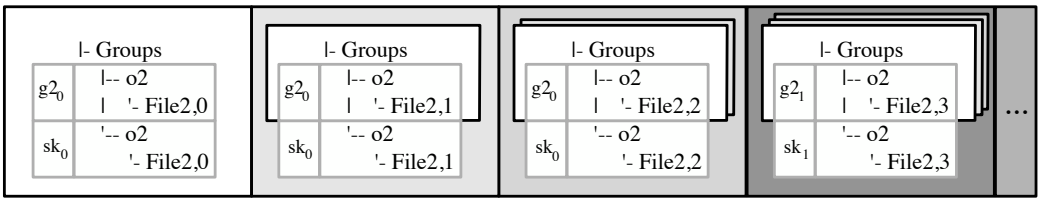


Figure 7.6: Shadow Structure providing Access to the current Version

Figure 7.6 represents an example of the shadow structures. While the data is encrypted and versioned with the help of the EKs, the most recent modifications are also applied to a copy encrypted by the shadow key. Therefore, the access to "File2" is provided not only via $g2$ but also via the related shadow key. Even if $g2$ in its version 0 protects several versions of the data, the related shadow key guards only the most recent version stored in the shadow data. If a client should only access the latest version, the shadow key is published in version 0. Suitable key trails are provided to reconstruct its version 1. Besides, $g2$ is only provided in its most recent version. The client, accessing the shadow key with all of its versions, has the ability to access the shadow data. The provided node $g2$ with its latest version offers no access to former versions of the normal data. Therefore, the client has only access to the most recent version of "File2": The access to the versioned storage is only provided by non-accessible versions of $g2$. The shadow data is encrypted with the accessible versions of the related shadow key.

By sharing the shadow key to accessing clients in a secure way (e.g. by encrypting the shadow key with the related CK as key trail), the client gains access to the most recent version without any possibility to read preceding versions of the data.

7.3.2 Token-based Extension

Another mechanism to restrict access to former versions on the client is the deployment of an authorization layer in the distributed environment. The key manager contains a list of resources mapped to the nodes. This mapping between the nodes and the data is enriched by a list of valid versions mapping the different clients. For each client, the resources in the data encrypted with a node are bound to versions. The additional authorization structure is deployed together with the global key graph in the key manager representing the binding between the versioning and the key graph.

Figure 7.7 shows an example: "Client 0" has access to version 3 of $g1$. "Client 1" has the ability to access all versions stored under the same EK. Since the mapping between

DAG and data takes place by a dedicated structure, each client contains all descendants of the own CK in all versions. Related to the example of Figure 7.7, "Client 0" contains all versions of $g0$ since the authorization for different versions is not in need of the key graph.

Each descendant of each CK represents at most one rule mapping the versions of the data to the versions of the key graph including the client. Regarding the example of Figure 7.7, "Client 0" contains three descendants of the own CK resulting in at most three rules. The access structure acts as a token-based approach including the cloud, the disjoint clients and the centralized key manager. The workflow is denoted by Figure 7.7.

1. The client requests a version. The requested access is verified against the authorization structure in the key manager.

2. Relying on the versions allowed for this client and the requested resource, a token is negotiated between the key manager and the cloud instance. The token is encrypted and only readable for the cloud and the key manager. Each token represents a single rule for a dedicated client.

3. After negotiation, the resulting token is sent to the client. The client is not able to decrypt the content neither to modify the content without violating it.

4. With every request, bound to the fixed resource in one of the desired versions, the token must be delivered to the cloud instance from the client. The cloud has the ability to decrypt the token and to verify the access on the requested version of the data.

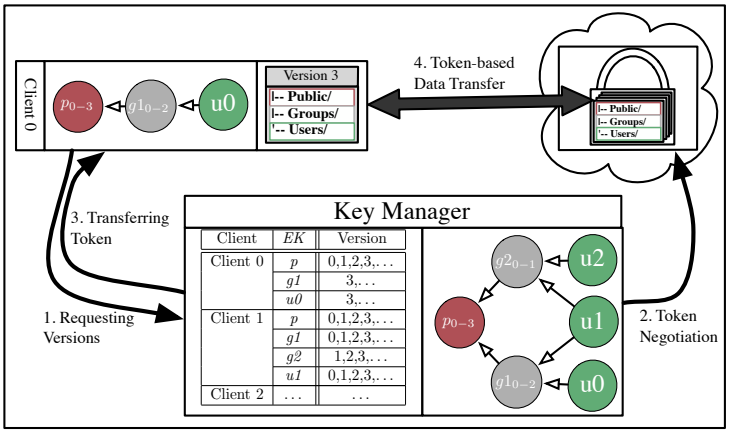


Figure 7.7: Token-based Extension to provide Access to the current Version

As already pointed out in Section 7.2.2, the storage must be aware of the versioning of the data although the different versions are encrypted. Due to the encryption of the

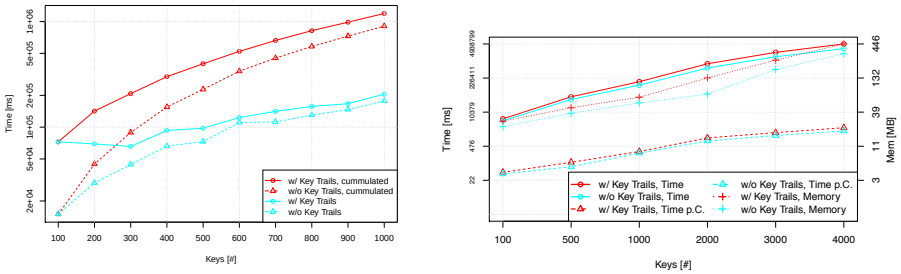
modifications with the help of the different EKs, the awareness of the data by the cloud storage provider is constricted only to the versioning and not to the data itself. As a result, the cloud storage provider can identify different versions in the EKs and the data but has neither the ability to gain access to the inlying data nor to the keys.

The client has the ability to encrypt all data stored in the cloud with the help of the decrypted key trails resulting in a subgraph of the global key graph. The access is thereby managed by the key graph additionally to the encryption. Each request from the client must contain a valid token, negotiated between key manager and cloud instance. The token enables the client to only access an allowed subset of the hierarchical data. Nevertheless, it would be possible to use the accessibility of all versions of the descendants of the CK if the client would have unauthorized access to all versions of the data.

7.4 Evaluation and Scaling

The results base on the evaluation of memory and time consumed for generating nodes and key trails. Furthermore, the impact of the virtual nodes is evaluated. The DAG, represented by nodes and key trails, is stored in a Berkeley DB [94].

Figure 7.8a shows the time consumed for inserting incrementally 1000 nodes including optionally adjacent key trails. In this benchmark, it is not distinguished between CKs or EKs. The time is measured once for the single nodes and once cumulated over all versions. Each insertion results thereby in one version. The lookup and adaption of the database indices by inserting nodes explains the increasing time consumed. Cumulated over all versions, the insertion scales linear with the size of the DAG. Furthermore, the creation of the key trails consumes a constant overhead.



(a) Time to generate Nodes and Key Trails

(b) Time and Memory for Nodes and Key Trails

Figure 7.8: Insertion Time of Nodes and Key Trails in the DAG

The measurement of the memory relies on a fixed-group simulating scenario where 4000 EKs are inserted in 12 levels resulting in at most 134 nodes per level. These values were randomly chosen, resulting in a DAG with the node distribution represented by Table 7.1.

Table 7.1: DAG Setup for Benchmark shown in Figure 7.8b

Nodes Inserted	DAG size	Nodes Updated
100	172	32
500	589	52
1000	1104	61
2000	2128	110
3000	3134	150
4000	4134	185

Figure 7.8b shows the results. The time needed to generate the nodes with and without key trails is measured and scales similar to Figure 7.8a. Focusing on the time consumed per client, the constant overhead of the generated key trails becomes visible. The memory consumed for inserting nodes and key trails scales linear as well. The artifact by inserting 2000 nodes is explainable by the non-linear insertion of nodes. Overall, the scaling of the utilized memory scales with the inserted nodes adjacent to the time consumed.

The evaluation of the impact of the virtual nodes takes place on a similar randomly generated DAG. The set up of the DAG is shown in Figure 7.9.

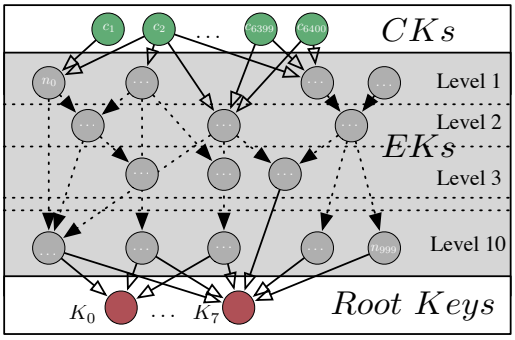
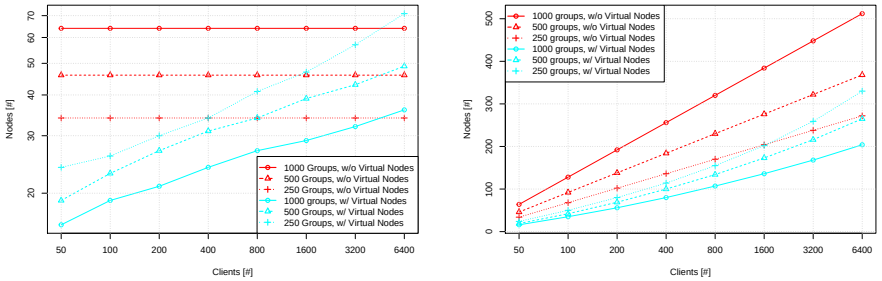


Figure 7.9: Random Setup for evaluating Virtual Nodes

This DAG is generated randomly with 250, 500 and 1000 EKs (representing groups) consisting of 10 levels. While the EKs are distributed equally on maximal 10 levels, 8 terminal nodes, called *Root Keys* in Figure 7.9, are included in this set of EKs. The out-degree of each node, except the terminal nodes, is at most 3, meaning that each node has at most 3 children. The in-degree of the EKs in opposite are not restricted. In this setup, incrementally, 6400 CKs are inserted linking to at most three 3 EKs. After each CK insertion, a new version for all descendants of the inserted CK is created. After a fixed number of CK insertions (50, 100, 200, 400, 800, 1600, 3200 and 6400) the generated key trails and the updated nodes are traced.

The creation is evaluated once by relying on virtual nodes. Another benchmark

constructs the DAG without virtual nodes. Since the number of edges relies on the number of nodes inserted, the number of nodes affected increases with the numbers of inserted CKs.



(a) Nodes updated **(b)** Nodes updated, Cumulated

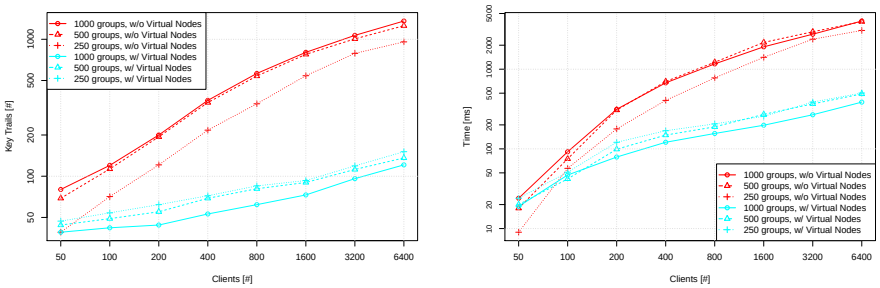
Figure 7.10: Number of Nodes affected by inserting CKs

Figure 7.10a represents the number of nodes touched by the insertion of the CKs. The y-axis denotes the number of EKs updated. The x-axis represents the CKs joining the DAG mapped on single versions. By inserting single CKs, only a constant number of related EKs is updated if no virtual nodes are inserted. Since the edges between the EKs already exist before the insertion of the CKs, the set of nodes updated by the insertion is constant. The less EKs exist in the DAG, the fewer nodes need to be updated.

The same benchmark is performed including virtual nodes represented by Figure 7.10a. Since the virtual nodes are inserted in the DAG while inserting the CKs, the density of the graph increases. The inserted CKs always connect to the same amount of EKs ignoring the number of available EKs. As a result, more virtual nodes are introduced when having less EKs. Since the virtual nodes are inserted in the DAG while creating the key graph, the DAG with the virtual nodes has a different layout than the DAG created without virtual nodes. This different architecture explains the lower number of nodes updated at the beginning. Even though the number of updated nodes is lower at the beginning, the scaling shows the price to be paid for the usage of the virtual nodes: Independent from the size of the key graph, the scaling is not linear any more due to the continuous insertion of virtual nodes. As a result, the scaling results in lesser nodes updated in larger graphs. The usage of virtual nodes loosens the DAG since each CK has less descendants. The number of updated nodes by using virtual nodes is therefore decreasing with the size of the key graph. This assumption is seconded by the number of cumulated updated nodes represented by Figure 7.10b. The scaling of the cumulated nodes updated is linear in the DAG without virtual nodes. This scaling substantiates the assumption that only a constant number of nodes is updated in each CK insertion. By inserting virtual nodes, the scaling of updated nodes is not linear any more. Due to the increasing number of virtual nodes in each update, the number of nodes increases even by insertion of CKs only. The scaling improves, the more nodes the DAG contains.

A loosely coupled structure enforces less updates ongoing with less insertions of virtual nodes.

The main argument for introducing virtual nodes is the generation of key trails. Figure 7.11a shows the number of key trails cumulated over all versions. The y-axis represents the generated key trails. The x-axis denotes the CKs joining the DAG. Logically, the more CKs are introduced in the DAG, the more key trails are generated. Since the key trails are computed relying on incident edges on the modified nodes, the scaling is linear regardless if virtual nodes are introduced or not. Since fewer nodes are updated by introducing virtual nodes, the number of key trails generated in each joining operation of a CK is smaller. Fewer nodes are affected if the key graph is already existing by inserting single CKs and virtual nodes. Both aspects motivate the usage of virtual nodes relying on suitable selected thresholds even though the number of nodes in the DAG might increase.



(a) Number of Key Trails per User **(b)** Time generating key trails

Figure 7.11: Evaluating Insertion Time and Number of Key Trails by inserting CKs

The time consumed to generate the key trails, represented by Figure 7.11b, scales with the amount of key trails needed. Using virtual nodes results in a better performance related to the key trails. Less updates are needed relying on the fixed defined in-degree of the children.

The evaluation of virtual nodes shows that the approach behaves like expected: Any modification on the DAG results in only a constant number of updated nodes, namely the descendants of the modified node. Furthermore, the number of key trails plays an important role in the approach. It scales with the number of nodes and can be further improved by using virtual nodes.

7.5 Conclusions

The described key management is tailored to versatile sharing functionalities on versioned data stored in the cloud by representing the following characteristics.

1. The awareness of versioned data about being encrypted not only results in mechanisms to offer backward secrecy without the need to re-encrypt data. It further-

more results in an orthogonal rights management granting versatile access for flexible groups as well as to user-defined versions.

2. An arrangement of keys enables users to flexibly grant, modify, and revoke access for users and own defined groups representing nodes in a directed acyclic graph. The graph-based structure thereby guarantees scalability and adaptability.

3. Distinguishing between managing the key graph and propagating updated key material allows the usage of the cloud as a distribution mechanism. The central key manager must only be accessible when actually adapting access rights. Updated key material is encrypted for secure distribution over the cloud. The storing cloud storage provider thereby has no knowledge about the key material but offers high available access for updating shares. The key trails are stored in buckets adjacent to the data using No-SQL stores directly. No computations on the keys in the cloud need thereby to be performed at any time.

The graph-based key management brings stream-based key graph approaches to the area of cloud storage. The distributed architecture enables evolving shares without the need of re-encrypting any data. Applying stream-based key managements, focusing only on the protection of new data (representing Forward Secrecy) to data at rest is challenging. Access to already persisted data must be guaranteed providing backward secrecy as well. Versioning data as well as the key material offers backward secrecy and includes higher level security goals like non-repudiation [107]. The key graph can therefore be equipped with signatures unique for the creating/updating node. Such functionality becomes even more important when taking regulations such as legal restrictions into account.

When a company processes data in the UK, stores it on a server in Ireland but sends it via France - as it may have a subsidiary there - it is not yet clear which country's law would prevail in a legal dispute.

EurActiv

8 Legal Aspects of Secure Cloud Storage

Contents

Achieving security is a challenge not solvable by applying technical solutions only. As described in the definition of security goals in Chapter 3.1, providing throughout security exceeds all technical possibilities. Cloud services have such an impact, that covering other areas such as jurisprudence becomes crucial. Recent developments show the need for techniques as well as for establishing global ethical and legal regulations to protect user data. Not only the worldwide distribution of information ignoring country borders are slowing down the creation of suitable procedures. There is an impedance between technical measures and legal regulations restricting collaboration on the same issues. This impedance even becomes obvious when only focusing on Germany itself. Legal regulations and technical measures use entirely different terminologies resulting in different interpretations questioning if the same problems have to be solved.

This chapter provides a small glimpse to law-related problems mapped on technical measures. An understandable summary mapping commonly defined security goals builds a basis for computer scientists as well as legal experts. The resulting vocabulary illuminates different areas like civil law, criminal law and technical measures resulting in a consistent mapping bridging disjoint areas of research.

8.1 Background

Evaluating legal issues in the context of cloud storage is challenging. Cloud data is not, in opposite to legal regulations, bound to countries any more. EurActiv provided a report about the occurring problems summarized in the quote at the beginning of this

Chapter [8]. EurActiv thereby focus in the European Union only, demanding binding regulations for its countries. Since most cloud storage providers store in the United States, things get even more complicated: Current whistleblowing activities show not only the missing deficiency for data protection in the United States. They also prove the unauthorized access on data stored by Facebook, Google or any other US-based company. This enforced access already lead to the shutdown of companies caring about confidential data handling. Lavabit, offering secure email in the United States, had the choice to either grant government agencies access to their infrastructure or to shutdown their business.

8.1.1 Contribution

Nontransparent external or internal access on the data show the need for reliable legal regulations. An evaluation of international legal aspects becomes even more difficult. Since covering legal aspects only represents a secondary area of this thesis, the interpretation is tailored to German applicable laws only. The contribution therefore is represented by two mappings.

- The technical applicable measures are mapped on the common security goals described in Chapter 3.1 summarizing the techniques.
- German laws are roughly applied to the defined security goals as well. This mapping is just an interpretation representing the challenge of a mapping of regulations to technical definitions.

The evaluation of the following legal aspects is mostly derived from the joint work with Jörg Eisele, Marcel Waldvogel and Marc Strittmatter published under the title "A Legal and Technical Perspective on Secure Cloud Storage" [59]. The interpretation of the legal aspects in Section 8.3 relies on the inputs of Jörg Eisele and Marc Strittmatter.

8.1.2 Defining the Focus

Storage of information obeys to legal implications in all law-related areas. For example, the regulations for taxes in Germany (AO) require taxpayers to store their tax records in Germany (Sec. 146 par. 2). With the allowance of the German tax authorities, storage in the European Union is possible (Sec. 146 par. 2a). According to Sec. 148, tax payers are allowed to store their records out the EU if the storage in Germany creates serious challenges for the tax payers according to long-term accessing and retrieval of the data. This single example shows the different applicable regulations and their impact from a legal perspective. Other examples can be found for example in the German Commercial Code (HGB), related to the storage of documents and business letters, as well as in Labor law, related to the storage of personal data of employees.

An in-depth analysis is not possible in the context of this thesis resulting in the following restrictions.

- Only regulations directly matching security goals are listed and interpreted. As denoted by the examples above, regulations cover all types of statutes. Since a complete list of all storage-related statutes exceeds this chapter, only acts are listed having a direct impact to the defined security goals from Section 3.1.

- Only German regulations are mapped to the security goals. Even though many of them apply to regulations in the EU, no interpretation or mapping of external statutes like US-regulations are made.

8.2 Combination of Technical Measures

As already stated in Section 3.1.1, security measures need to be combined: Relying on each other, partially satisfied security goals make a system vulnerable. So far, different measures satisfy the defined security goals recapitulated in Figure 8.1.

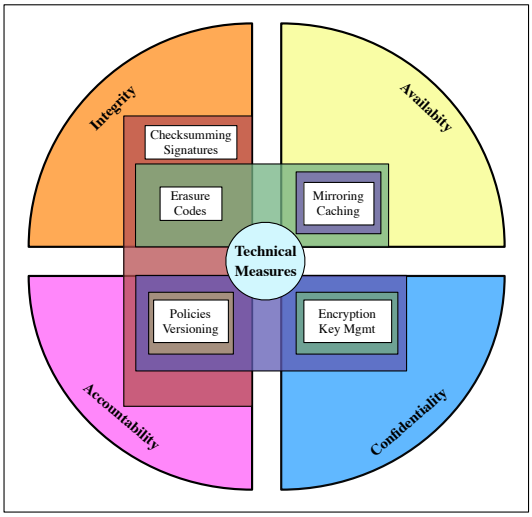


Figure 8.1: Technical Measures mapping Security Measures to Security Goals

Some of these techniques comply with more than one security requirement. Equipping the data with checksums as well as signatures guarantees integrity. If combined with versioning, accountability is provided. Erasure codes compensating small errors in the remotely stored buckets guard not only integrity but also increase availability of the data. Mirroring and caching of the data furthermore increases the probability of a successful access. Encrypting the data and managing the keys with respect to cloud storage scenarios guards confidentiality.

This mapping just gives a rough overview since further combinations of applied techniques are possible: Handling signatures for example may benefit from a suitable key management approach. The ability to revoke signatures thereby influences the integrity as well.

Proposed techniques must, as denoted in Chapter 3, be applied on the client-side. The approaches of this thesis aim to provide such a combination to satisfy all security goals. Nevertheless, the delocalization of uploaded information generates interdisciplinary interests as well. One example are legal aspects discussed in the following Section 8.3.

8.3 Legal implications

Legal implications always influence the handling of user data in the cloud. The interpretation of legal statutes thereby extends technical measures providing at least partly assurance. Since jurisprudence judges on the base of circumstances, a matching of legal aspects to the denoted security goals is hard to achieve. From this perspective, the following mapping should more be seen as a rough approximation instead of a definitely detailed mapping. The legal disciplines touched in the following approximation are data protection law, torts law, contracts and criminal law.

Applying acts to cloud storage is not as easy as it seems: The geographical distribution of services makes the application of mandatory law often hard to determine. The global nature of interconnected infrastructures ignores country borders. Data stored in a cloud while uploaded from e.g. Germany might be illegal in other countries. Consequently, the focus of legal statutes to cloud storage security lies on the application of German statutory laws relying on Sec. 9 I StGB. The place of action (namely the initial push of the data into the cloud) is defined in Germany. Sec. 7 I StGB furthermore defines the application of repressive measures if the action is unlawful in other countries as well.

Technical undefined terms in statues as “notable” and “necessary” make an interpretation of law even more complicated. Therefore, the implications of security to cloud storage are evaluated based upon different scenarios:

8.3.0.1 Unauthorized access

From a technical point of view, there is an immense difference how to establish confidentiality. Either data is encrypted or the access is just blocked. From a legal point of view, Sec. 202a StGB and Sec. 202b StGB prevent any unauthorized access independent from restricted measures. Only accessing restricted content in an unauthorized manner is sufficient to harm those statutes, regardless if the accessor attacks from outside or is represented by an internal person. It is important to know that even the preparation of unauthorized data access is indictable by Sec. 202c StGB.

8.3.0.2 Harming data

Unauthorized modifications or deletions of data are tackled by criminal law covered by Sec. 303a StGB. Roughly mapped to the denoted security goals, an unexpected status of the data thereby harms integrity and availability. If a copy of the unauthorized removed/modified data exists, this act might not be impinged. The preparation to

make data inaccessible in an unauthorized way is covered by Sec. 202c StGB similar to the preparation of unauthorized access. Adjacent to possibly unauthorized access, it is unimportant if the attack or the preparation occurs from outside or inside the cloud as long as the data is harmed

8.3.0.3 Data privacy

In the cloud, data privacy is most probably the most discussed issue. Pushing data in the cloud disables the user from gaining any knowledge about access to the data. The storage of information in untrusted infrastructures thereby not only harms confidentiality. If the data is interpreted as personal-related data, it touches all security requirements from a legal point of view. German law about data privacy is rather strict when personal information is stored. From an EU perspective, any stored personal information must be handled in a way, that the user keeps control over the data. This control has to be enforced directly or indirectly by installing a contractual data controller-data processor relationship. Data processing furthermore must be restricted to countries with acceptable levels of data security. In practice, these regulations are rarely enforced since the major cloud storage providers are based in the United States: The worldwide accessibility of cloud services results in the suspicion of unattended data access. Claiming in-depth knowledge about personal data stored e.g. by Facebook or Google is possible, but the results are open to be questioned. Especially when it comes to user-triggered removal operations, it is questionable if the data is really deleted. Nevertheless, harming the related German statute Sec. 43 II BDSG can be interpreted as unauthorized modification (mapping the confidentiality and integrity). Furthermore, based on these statutes protecting the personal data, the accountability might be harmed: The user has officially the ability to order a reconstruction of all actions taking place on the data. As already mentioned, data privacy is handled differently in different countries complicating related user-requests. European harmonization has installed a minimum level of protection. Current 2011 ECJ (European Court of Justice) decisions have triggered legal discussions of the need for a maximum protection level by EU law. Such regulations overrule more protective country laws (such as German BDSG). Ongoing discussions about feasible data handling in Europe furthermore include the establishing of technical security measures including their impact on relevant privacy statues. As an example, it is currently unclear from a legal point of view if data privacy statutes must be applied on encrypted data.

8.3.0.4 Author's rights

The ease of collaboration brings concerns about author's right into the focus of security. Unauthorized access thereby not only covers the field of confidentiality, it furthermore harms the accountability. Unauthorized copies of data are harming author's rights especially when the attacker's intent is to make unlawful profit. Related statues harmed in these scenarios are Sec. 106 and 108 UrG.

8.3.0.5 Contracts

Contractual definitions of "confidentiality" and "security" are typically subject to the parties´ appraisal in contracts. The applicable law is thereby freely eligible by merchant parties to contracts, with some restrictions where one party is an end-user. Depending on this applicable law, the definition of what the parties accept as "secure" or define as "confidential" is often missing. It can vary based on the perception of the parties. The cloud storage provider even ties the minimum level of security to one of its contractual partners. Regulations in contracts like "You will take all reasonable measures to avoid disclosure, dissemination or unauthorized use of XY Confidential Information, including, at a minimum, those measures you take to protect your own confidential information of a similar nature." try to establish this minimum level of protection. Similar to privacy law, European interpretations are stricter than US-based ones. In essence, there is definitely a need to discuss standardized legal concepts, which are intended to be used for multi-jurisdictional relationships.

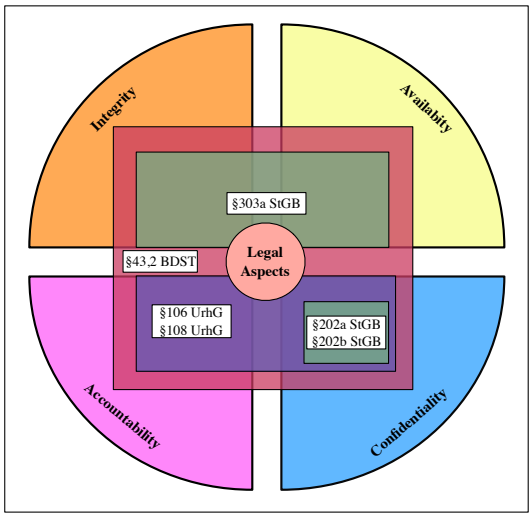


Figure 8.2: Legal Measures mapping Regulations to Security Goals

Figure 8.2 summarizes the denoted regulations mapping the different security requirements. Similar to the technical measures, many regulations match multiple security goals: Privacy law for example guards all security requirements since the data must be handled in a way like a physical possession is present. As a consequence, the user has, referring to privacy law statutes, the right to access the data at any time, to rely on unharmed content, to gain knowledge about actions occurred on the data as well as to restrict access to remotely stored information. Criminal law focus mainly on the availability and the integrity of the data as well as on the access. Access to unharmed, own data is protected including the certainty that unauthorized access is prohibited. Besides guarded by privacy and criminal law, own created data is also protected by applicable copyrights. Not only has a creator the choice to define the accessing parties,

but also to gain knowledge where the own created content is re-used touching the area of accountability. Further regulations are possible depending on concrete cases they could be applied on. Contracts are excluded in Figure 8.2 since they represent such a special case applicable only between the participating parties.

8.4 Conclusions

The need to establish a universal vocabulary is satisfied by mapping German regulations as well as technical measures to common security goals. This mapping is thereby very challenging since legal regulations are often interpreted by facts. Using the common security requirements defined in Chapter 3.1, the proposed mapping not only translates legal problems in a language understandable for computer experts and vice versa. Establishing security measures results in an even new legal perspective. It is questionable if and how many statutes interact with established security measures. The resulting mapping not only presents a breakdown of techniques and legal situations. By offering a common terminology, it also provides a base for interpreting established security measures from a legal point of view.

It's stupidity. It's worse than stupidity: it's a marketing hype campaign.

Richard M. Stallman - Software Activist / 1953 -

9 Conclusions

Cloud-based services are widely considered as panacea even though the technical ideas behind are rather old and well-established. Nevertheless, cloud storage are hyped everywhere, although, technically, the term "public cloud" is replaceable with "untrusted, highly available storage". Of course, the latter is not as marketable as the former. Higher bandwidth paired with the need to store and exchange information has triggered a hype, as Richard M. Stallman states.

When it comes to data, the pervasive and easy utilization of cloud storage made its way to end users. Allowing easy and permanent access for free sounds too alluring for many customers. Most users are unaware that they pay even for complimentary offers: The price is their data.

The users only became aware of the need to protect their cloud-based data in the past few months. The uncontrolled and intransparent access of nation states oder government agencies to the internal infrastructure of major cloud storage providers scrutinizes the storage of personal data in the cloud. Security measures protecting personal information just find their ways into end-user products such as Boxcryptor [4] or Spideroak [19]. Even though these clients currently mostly guard confidentiality, research approaches already cover the protection of other security requirements as well.

In this context, this thesis not only describes approaches to protect accountability, availability, integrity, and confidentiality of different data. Major parts of the architecture are implemented, just waiting to be turned into products. Revisiting the claims within Section 2.2, the contributions of this thesis can be summarized as follows.

1. **Legal Aspects of Secure Cloud Storage**
   → Legal regulations must not remain incompatible with technical security measures.
   Descriptions of technical measures and legal regulations in the security domain use completely different terminologies. The legal perspective of secure cloud storage has to map the technical possibilities for guarding the data to reduce friction loss. Relying on commonly defined security goals, Chapter 8 combines such a consistent mapping applied to German laws and regulations. The result is an easy-to-understand description, hoping to reduce tension and misunderstandings between lawmakers and technical experts. For the first time, aspects of jurisprudence are directly connected to computer science measures bridging disjoint research areas.

2. **Flexible Key Management for a versioned Cloud**
   → Versioning not only enhances transparency and reliability, but simplifies protection of confidentiality and accountability.
   Collaboration represents one of the main arguments for using cloud storage. Sharing needs to consider access on versioned, encrypted data, protecting accountability and confidentiality. Collaboration, versioning and encryption can be combined by sharing key material as described in Chapter 7. Flexible group management enables a key mapping on data as well as on revisions guaranteeing accountability without harming confidentiality. Continuously propagating new key material takes place over the cloud in a confidential way making use of its high availability. Presenting an orthogonal method for time-restrictive access, the evaluation supports the scalability and versatility of this key management.

3. **Secure and Independent Data in No-SQL Stores**
   → The easy and efficient gateway from legacy applications to safe and secure cloud storage.
   Establishing security measures independent of the data by relying on defined policies needs mappings of data to key/value stores. Covering different abstraction layers ranging from blocks to services, Chapter 6.1 not only shows reference implementations of four mappings but provides also in-depth evaluations of different data containers. Mapping data to No-SQL structures not only increases confidentiality, accountability and integrity free of charge but also enables the use of professional cloud storage for various kinds of data.

4. **Photo Sharing Websites as Complimentary Cloud Stores**
   → Why pay for a service, if you can get it for free?
   Professional cloud storage bills transfer and storage while end-user cloud services stay mainly free of charge. Since cloud services are stacked upon each other, the hardware infrastructures are equivalent and independent from the running service. To use such vast, free but data-dependent cloud services, Chapter 6.2 describes the mapping of key/value tuples to images stored in Facebook, Picasa and Flickr. Relying on APIs, the implementation provides free use of photo sharing websites as (almost) professional storage gaining most benefits of cloud storage.

5. **Integrity in Key/Value-Stores**
   → Flexible bucket packing results in higher efficiency and ACID properties.
   REST offers versatile access to remote No-SQL stores, although stateful modifications on resources are not possible. Integrity in the cloud relies thereby mostly on the establishment of probing, error correction and mirroring. Chapter 5 interconnects buckets offering hierarchical integrity checks upon retrieval and storage. Fine-grained versioning furthermore enables automatic scrubbing of different revisions ongoing with standard accesses. The resulting architecture not only offers integrity and accountability but also enables COW on RESTful resources resulting in ACID conformant operations.

6. **Adaptive Versioning**
   → Clever versioning relieves the system from wasting time with other tasks.
   Backups need to cover multiple former versions of data motivating the usage of versioning techniques. Using remote storage techniques results in unscalable scrubbing operations, peak loads and visiting an infinite number of change sets when accessing a single version. Chapter 4 describes the writing of unmodified data piggybacked onto ongoing modifications. One single value, the sliding window, offers an adaptive and powerful parameter providing a flexible choice between heavier loaded change sets and larger number of change sets representing single versions.

Although the ideas, evaluations and approaches offer the ability to guard cloud-stored data, gaining security is not only a technical problem. Users must become aware that usability and versatile access including sharing functionalities gained by using the cloud always have to be paid for: Either with money or with information itself. Nevertheless, it is the users' right to claim back the authority on her own data, either by establishing security measures already available or by using services which keep in mind that the physical control is lost in the cloud including all consequences. Most of these properties are simplified by versioning. Versioning throughout the system might provide even more benefits, including better updates for full-text indexing, and many more.

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

References

[1] Amazon EC2. http://aws.amazon.com/ec2/, 2013.

[2] Amazon Elastic Beanstalk. http://aws.amazon.com/elasticbeanstalk/, 2013.

[3] Amazon S3 Storage. http://aws.amazon.com/s3/, 2013.

[4] BoxCryptor. https://www.boxcryptor.com, 2013.

[5] Dropbox. http://www.dropbox.org, 2013.

[6] Duplicity. http://duplicity.nongnu.org/, 2013.

[7] EncFS. http://www.arg0.net/encfs, 2013.

[8] Euractiv - cloud computing: Leveraging the digital economy. http://www.euractiv.com/innovation-enterprise/cloud-computing-legal-maze-europ-linksdossier-511262, 2013.

[9] Git. http://git-scm.com/, 2013.

[10] Google App Engine. https://appengine.google.com, 2013.

[11] Google Cloud Storage. https://cloud.google.com/products/cloud-storage, 2013.

[12] Google Drive. https://drive.google.com, 2013.

[13] jClouds, Java API for accessing cloud services. http://jclouds.org, 2013.

[14] Microsoft Skydrive. https://skydrive.live.com, 2013.

[15] Microsoft Windows Azure. http://www.windowsazure.com, 2013.

[16] Owncloud. http://owncloud.org/, 2013.

[17] Saxon - xslt and xquery processor. http://www.saxonica.com/, 2013.

[18] Sparkleshare. http://sparkleshare.org/, 2013.

[19] Spideroak. https://spideroak.com/, 2013.

[20] TeamDrive. http://www.teamdrive.com, 2013.

[21] Truecrypt. http://www.truecrypt.org/, 2013.

[22] Wuala. http://www.wuala.com, 2013.

[23] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. Racs: A case for cloud storage diversity. In *Proceedings of the Symposium on Cloud Computing*, 2010.

[24] Mohammed A. AlZain, Eric Pardede, ben Soh, and James A. Thom. Cloud computing security: From single to multi-clouds. In *45th Hawaii International Conference on System Services*, 2012.

[25] Mohammed A. AlZain, Ben Soh, and Eric Pardede. Mcdb: Using multi-clouds to ensure security in cloud computing. In *Proceedings of the Ninth Internation Conference on Dependable, Automatic and Secure Computing*, 2011.

[26] Raluca Ada Popa andJ acob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang. Enabling security in cloud storage slas with cloudproof. In *Proceedings of 2011 USENIX Annual Technical Conference, Portland, OR*, 2011.

[27] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[28] Doug Beaver, Sanjeev Kumar, Harry C. Li, and Jason Sobel an Peter Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *USENIX OSDI*, 2010.

[29] Anders Berglund, Scott Boag, Don Chamberlin, Mary F/ Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon. Xml path language (xpath) 2.0. *W3C recommendation*, 2007.

[30] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. In *Proceedings of the Sixth European Conference on Computer systems (EuroSys)*, 2011.

[31] David M. Blenson, David A. McGrew, and Alan T. Sherman. Key management for large dynamic groups: One way function trees and amortized initialization. *Advanced Security Research Journal*, 1998.

[32] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu andJonathan Robie, Jérôme Siméon, and Mugur Stefanescu. Xquery 1.0: An xml query language. *W3C working draft*, 2003.

[33] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *FAST 2003: 2nd Usenix Conference on File and Storage Technologies*, 2003.

[34] Moritz Borgmann, Tobias Hahn, Michael Herfert, Thomas Kunz, Marcel Richter, Ursula Viebeg, and Sven Vowé. On the security of cloud storage services. Technical report, Fraunhofer Institute for Secure Information Technology SIT, 2012.

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

[36] Kevin D. Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: theory and implementation. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, 2009.

[37] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1, 2000.

[38] Tim Bray, Jean Paoli, Michael C. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Journal*, 1997.

[39] Patrice Brend'amour. *Performance und Robustheit bei blockbasierter Datenkommunikation am Beispiel jSCSI.* Bachelor's thesis, University of Konstanz, 2009.

[40] Christian Cachin, Robert Haas, and Marko Vukolić. Dependable storage in the intercloud. Technical report, IBM Research - Zurich, 2010.

[41] Christian Cachin, Idit Keidar, and Alexander Shraer. Trusting the cloud. In *SIGACT News*, 2009.

[42] Germano Caronni and Marcel Waldvogel. Establishing trust in distributed storage providers. In *Proceedings of Third IEEE International Conference on Peer-to-Peer Computing (P2P 2003)*, 2003.

[43] Yao Chen and Radu Sion. To cloud or not to cloud? musings on costs and viability. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011.

[44] James Clark. Xsl transformations (xslt). *World Wide Web Journal*, 1999.

[45] Russell Coker. Bonnie++. `http://www.coker.com.au/bonnie++/`, 2013.

[46] Ernesto Damiani and Francesco Pagano. Handling confidential data on the untrusted cloud: An agent-based approach. In *Cloud Computing '10*, 2010.

[47] Hrishikesh Dewan and R C. Hansdah. A survey of cloud storage facilities. In *Services (SERVICES), 2011 IEEE World Congress on*, 2011.

[48] Gurpreet Dhillon and James Backhouse. Technical opinion: Information system security management in the new millennium. *Communications of the ACM*, 2000.

[49] Marten Van Dijk and Ari Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. In *Proceedings of the 5th USENIX conference on Hot topics in security*, 2010.

[50] Andreas Ergenzinger. *A platform-independent iSCSI Target in Java.* Bachelor's thesis, University of Konstanz, 2012.

[51] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[52] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Rfc 2616: Hypertext transfer protocol–http/1.1. `http://www.ietf.org/rfc/rfc2616.txt`, 1999.

[53] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, Irvine, 2000.

[54] Craig Gentry. *A fully homomorphic encryption scheme.* PhD thesis, Stanford University, 2009.

[55] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. Sirius: Securing remote untrusted storage. In *Proc. NDSS*, 2003.

[56] Sebastian Graf. A secure cloud gateway based upon xml and web services. In *Proceedings of the 9th IEEE European Conference on Web Services, PhD Symposium*, 2011.

[57] Sebastian Graf, Sebastian Kay Belle, and Marcel Waldvogel. Rolling boles, optimal XML structure integrity for updating operations. In *Proceedings of the 20th international conference on World wide web*, 2011.

[58] Sebastian Graf, Patrice Brend'amour, and Marcel Waldvogel. jSCSI 2.0, Multithreaded Low-Level Distributed Block Access. Technical report, University of Konstanz, 2010.

[59] Sebastian Graf, Jörg Eisele, Marcel Waldvogel, and Marc Strittmatter. A legal and technical perspective on secure cloud storage. In *Proceedings of the 5. DFN-Forum Kommunikationstechnologien : Verteilte Systeme im Wissenschaftsbereich*, 2012.

[60] Sebastian Graf, Marc Kramis, and Marcel Waldvogel. Treetank: designing a versioned XML storage. In *Proceedings of the XMLPrague, Poster Track*, 2011.

[61] Sebastian Graf, Patrick Lang, Stefan A. Hohenadel, and Marcel Waldvogel. Versatile key management for secure cloud storage. In *Proceedings of the First International Workshop on Dependability Issues in Cloud Computing*, 2012.

[62] Sebastian Graf, Lukas Lewandowski, and Christian Grün. JAX-RX, unified REST access to XML resources. Technical report, University of Konstanz, 2010.

[63] Sebastian Graf, Lukas Lewandowski, and Marcel Waldvogel. Integrity assurance for RESTful XML. In *Proceedings of the 2010 international conference on Advances in conceptual modeling: applications and challenges*, ER'10, 2010.

[64] Sebastian Graf, Wolfgang Miller, and Marcel Waldvogel. Utilizing photo sharing websites for cloud storage. Technical report, University of Konstanz, 2013.

[65] Sebastian Graf, Vyacheslav Zholudev, Lukas Lewandowski, and Marcel Waldvogel. Hecate, managing authorization with restful xml. In *Proceedings of the 2nd Workshop on RESTful Services*, WS-REST '11, 2011.

[66] Bernd Grobauer, Tobias Walloscheck, and Elmar Stócker. Understanding cloud computing vulnerabilities. In *IEEE Security and Privacy*, 2011.

[67] Dominik Grolimund, Luzius Meisser, Stefan Schmid, and Roger Wattenhofer. Cryptree: A Folder Tree Structure for Cryptographic File Systems. In *25th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2006.

[68] Andreas Haeberlen. A case for the accountable cloud. *SIGOPS Operating Systems Review*, 2010.

[69] Hugh Harney and Carl Muckenhirn. Rfc 2094: Group key management protocol (gkmp) architecture. `http://www.ietf.org/rfc/rfc2094.txt`, 1997.

[70] Hani Ragab Hassen, Abdelmadjid Bouabdallah, and Hatem Bettahar. A new and efficient key management scheme for content access control within tree hierarchies. In *Advanced Information Networking and Applications Workshops*, 2007.

[71] Wayne Jansen and Timothy Grance. Guidelines on security and privacy. *National Institute of Standards and Technology*, 2011.

[72] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[73] Seny Kamara and Kristin Lauter. Cryptographic cloud storage. In *Proceedings of Financial Cryptography: Workshop on Real-Life Cryptographic Protocols and Standardization*, 2010.

[74] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Cs2: A searchable cryptographic cloud storage system. *Microsoft Research, TechReport MSR-TR-2011-58*, 2011.

[75] Auguste Kerckhoffs. *La cryptographie militaire*. University Microfilms, 1978.

[76] Vishal Kher and Yongdae Kim. Securing distributed storage: challenges, techniques, and systems. In *Proceedings of the 2005 ACM workshop on Storage security and survivability*, 2005.

[77] Ryan K.L. Ko, Peter Jagadpramana, Miranda Mowbray, Siani Pearson, Markus Kirchberg, Qianhui Liang, and Bu Sung Lee. Trustcloud: A framework for accountability and trust in cloud computing. In *IEEE World Congress on Services*, 2011.

[78] Marc Kramis. Method for hosting a plurality of versions of memory pages in a storage system and accessing the same. `http://www.patentlens.net/patentlens/patent/WO_2009_141161_A1/en/`, 11 2009.

[79] Marc Kramis, Alexander Onea, and Sebastian Graf. Perfidix : a generic java benchmarking tool. In *Jazoon '07 - The International Conference on Java Technology*, 2007.

[80] Marc Kramis, Volker Wildi, Bastian Lemke, Sebastian Graf, Halldór Janetzko, and Marcel Waldvogel. jscsi - a java iscsi initiator. In *Jazoon '07 - The International Conference on Java Technology*, 2007.

[81] Pradip Lamsal. Understanding trust and security. Technical report, University of Helsinki, 2001.

[82] Bastian Lemke. *Freiheitsgrade beim Einsatz Verteilter Disks*. Bachelor's thesis, University of Konstanz, 2008.

[83] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. What's inside the cloud? an architectural map of the cloud landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, 2009.

[84] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS)*, 2011.

[85] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Distributed Computing*, 1998.

[86] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: the penn treebank. In *Computational Linguistics*, 1993.

[87] Peter Mell and Timothy Grance. The nist definition of cloud computing. *National Institute of Standards and Technology*, 2009.

[88] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, 1987.

[89] Wolfgang Miller. *Exploiting Facebook, Flickr, and Picasa : Utilizing Photo Sharing Websites as Cloud Storage Backends.* Master's thesis, University of Konstanz, 2013.

[90] Shuai Mu, Kang Chen, Pin Gao, Feng Ye, Yongwei Wu, and Weimin Zheng. μlibcloud: Providing high available and uniform accessing to multiple cloud storages. In *Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on*, 2012.

[91] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, 2011.

[92] Surya Nepal, Shiping Chen, Jinhui Yao, and Danan Thilakanathan. Diaas: Data integrity as a service in the cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, 2011.

[93] Martin S. Olivier. Database privacy. *SIGKDD Explorations*, 2003.

[94] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, 1999.

[95] Bernd Panzer-Steindel. Data integrity. Technical report, CERN/IT, 2007.

[96] Siani Pearson and Andrew Charlesworth. Accountability as a way forward for privacy protection in the cloud. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, 2009.

[97] Andreas Rain. *Mapping different datatypes ensuring secure cloud storage.* Bachelor's thesis, University of Konstanz, 2013.

[98] Francisco Rocha and Miguel Correia. Lucy in the sky without diamonds: Stealing confidential data in the cloud. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, 2011.

[99] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 1984.

[100] Hiroyuki Sato, Atsushi Kanai, and Shigeaki Tanimoto. A cloud trust model in a security aware cloud. In *Applications and the Internet '10*, 2010.

[101] Julian Satran, Kalman Meth, Constantine Sapuntzakis, Mallikarjun Chadalapaka, and Efri Zeidner. Rfc 3720: Internet small computer systems interface (iscsi). `http://www.ietf.org/rfc/rfc3720.txt`, 2004.

[102] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *International Conference on Very Large Data Bases*, 2002.

[103] Bruce Schneier. *Secrets and lies: digital security in a networked world.* John Wiley, 2000.

[104] Adi Shamir. How to share a secret. *Communications of the ACM*, 1979.

[105] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. Venus: Verification for untrusted cloud storage. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, 2010.

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

[107] Gary Stoneburner. Underlaying technical models for information technology security. *National Institute of Standards and Technology*, 2001.

[108] Subashini Subashini and V. Kavitha. A survey on security issues in service delivery models of cloud computing. In *Journal of Network and Computer Applications*, 2010.

[109] Yan Sun and K.J. Ray Liu. Scalable hierarchical access control in secure group communications. In *Proceedings of the 2004 IEEE Infocom*, 2004.

[110] Hassan Takabi, James BD Joshi, and Gail-Joon Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security & Privacy*, 2010.

[111] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. Bluesky: A cloud-backed file system for the enterprise. In *Proc. of FAST*, 2012.

[112] Marcel Waldvogel, Germano Caronni, Dan Sun, Nathalie Weiler, and Bernhard Plattner. The VersaKey framework: Versatile group key management. *IEEE Journal on Selected Areas in Communications*, 1999.

[113] Debby Wallner, El Harder, and Ryan Agee. Rfc 2627: Key management for multicast: Issues and architectures. `http://www.ietf.org/rfc/rfc2627.txt`, 1999.

[114] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Ensuring data storage security in cloud computing. In *Proceedings of the 2009 IWQos Workshop on Quality of Service*, 2009.

[115] Lifei Wei, Haojin Zhu, Zhenfu Cao, Weiwei Jia, and Athanasios V Vasilakos. Seccloud: Bridging secure storage and computation in cloud. In *Distributed Computing Systems Workshops (ICDCSW), 2010 IEEE 30th International Conference on*, 2010.

[116] Volker Wildi. *Java iSCSI Initiator*. Master's thesis, University of Konstanz, 2007.

[117] Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. Secure group communication using key graphs. *IEEE/ACM Transaction on Networking*, 2000.

[118] Jin-Song Xu, Ru-Cheng Huang, Wan-Ming Huang, and Geng Yang. Secure document service for cloud computing. In *CloudCom '09*, 2009.

[119] Jinhui Yao, Shiping Chen, Surya Nepal, David Levy, and John Zic. Truststore: Making amazon s3 trustworthy with services composition. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010.

[120] Jinhui Yao, Shiping Chen, Chen Wang, David Levy, and John Zic. Accountability as a service for the cloud. 2010.

[121] Qiong Zhang, Yuke Wang, and Jason P. Jue. A key management scheme for hierarchical access control in group communication. 2008.