https://doi.org/10.35662/unine-thesis-2865

Université de Neuchâtel
Faculté des Sciences
Institut d'Informatique

Composing private and censorship-resistant solutions for distributed storage

par

Dorian Burihabwa

Thèse

présentée à la Faculté des Sciences
pour l'obtention du grade de Docteur ès Sciences

Composition du jury:

Prof. Pascal Felber, Directeur de thèse
Université de Neuchâtel, Suisse

Dr. Valerio Schiavoni, Co-Directeur de thèse
Université de Neuchâtel, Suisse

Prof. Daniel E. Lucani
Aarhus University, Danemark

Prof. Romain Rouvoy
Université de Lille, France

Soutenue le 3 Septembre 2020

UNIVERSITÉ DE NEUCHÂTEL

Faculté des Sciences
Secrétariat-décanat de Faculté
Rue Emile-Argand 11
2000 Neuchâtel – Suisse
Tél : + 41 (0)32 718 21 00
E-mail : secretariat.sciences@unine.ch

IMPRIMATUR POUR THESE DE DOCTORAT

La Faculté des sciences de l'Université de Neuchâtel autorise l'impression de la présente thèse soutenue par

Monsieur Dorian BURIHABWA

Titre:

"Composing private and censorship-resistant solutions for distributed storage"

sur le rapport des membres du jury composé comme suit:

- Prof. Pascal Felber, directeur de thèse, Université de Neuchâtel, Suisse
- Prof. Romain Rouvoy, Université de Lille, France
- Prof. Daniel Lucani, Aarhus University, Danemark
- Dr Valerio Schiavoni, Université de Neuchâtel, Suisse

Neuchâtel, le 8 septembre 2020

Le Doyen, Prof. A. Bangerter

Acknowledgements

This work would not have been possible with the help and collaboration of the staff at IIUN. I would like to thank Pascal Felber for the opportunity to join his group and the vision he shared along the way. Valerio Schiavoni for his mentoring and guidance throughout the PhD. Hugues Mercier for his availability and sharing his work. Members of the jury for their work and the insightful discussion during the defense.
But most importantly I would like to thank my parents, Anne-Élysée and Joseph, for their continuous support along this tortuous journey and my sisters, Leslie and Céline, for cheering me up during the years.

Abstract

Cloud storage has durably entered the stage as go-to solution for business and personal storage. Virtually extending storage capabilities to infinity, cloud storage enables companies and individuals to focus on content creation without fear of running out of space or losing data. But as users entrust more and more data to the cloud, they also have to accept a loss of control over the data they offload to the cloud. At a time when online services seem to be making a significant part of their profits by exploiting customer data, concerns over privacy and integrity of said data naturally arise. Are their online documents read by the storage provider or its employees? Is the content of these documents shared with third party partners of the storage provider? What happens if the provider goes bankrupt? Whatever answer can be offered by the storage provider, the loss of control should be cause for concern.
But storage providers also have to worry about trust and reliability. As they build distributed solutions to accommodate their customers' needs, these concerns of control extend to the infrastructure they operate on. Conciliating security, confidentiality, resilience and performance over large sets of distributed storage nodes is a tricky balancing act. And even when a suitable balance can be found, it is often done at the expense of increased storage overhead.

In this dissertation, we try to mitigate these issues by focusing on three aspects. First, we study solutions to empower users with flexible tooling ensuring security, integrity and redundancy in distributed storage settings. By leveraging public cloud storage offerings to build a configurable file system and storage middleware, we show that securing cloud-storage from the client-side is an effective way maintaining control. Second, we build a distributed archive whose resilience goes beyond standard redundancy schemes. To achieve this, we implement RECAST, relying on a data entanglement scheme, that encodes and distributes data over a set of storage nodes to ensure durability at a manageable cost. Finally, we look into offsetting the increase in storage overhead by means of data reduction. This is made possible by the use of Generalised Deduplication, a scheme that improves over classical data deduplication by detecting similarities beyond exact matches.

Keywords: distributed storage, file systems, erasure coding, encryption, data entanglement, censorship-resistance, data deduplication, user-space

Résumé

Le stockage dans le cloud est une pratique communément adoptée pour la sauvegarde de données privées et professionnelles. Virtuellement illimitées, les capacités de stockage dans le cloud permettent à tout un chacun de se concentrer sur son activité sans crainte de manquer d'espace ou de perdre des données. Mais si les utilisateurs confient de plus en plus de données à ces fournisseurs de service de stockage en ligne, ils le font au prix d'une certaine perte de contrôle. Et à une époque où de nombreux services en ligne font une partie de leur chiffre d'affaires sur l'exploitation des données et méta-données utilisateurs, des questions de confidentialité et de sécurité se posent. Les documents mis en lignes par les utilisateurs sont-ils lus par le fournisseur de service ? Le contenu de ces documents est-il partagé par le fournisseur de service avec des partenaires tiers ? Qu'advient-il des données lorsque le fournisseur de service fait faillite ? Si les fournisseurs de services s'efforcent d'apporter des réponses satisfaisantes à leurs clients, la perte de contrôle sur les données continuent d'alimenter de réelles inquiétudes.
À ces inquiétudes vient s'ajouter la question de la fiabilité du service offert par ces fournisseurs de stockage. En effet, la plupart des offres sont construites sur des centres de données dispersés à travers le monde. Si la répartition des données permet une meilleure qualité de service, elle amène également son lot de difficultés. Les fournisseurs de services doivent désormais anticiper et prévenir les problèmes survenant à la fois à l'intérieur mais également sur le réseau entre centres de données. Trouver l'équilibre entre sécurité, confidentialité, résilience et performance tout en coordonnant un grand nombre de nœuds de stockage répartis n'est pas une chose aisée. Et même lorsqu'une formule équilibrée est trouvée, elle se paie souvent par une augmentation des coûts de stockage.

Dans cette thèse, nous tentons d'apporter des solutions à ces problèmes en nous concentrant sur trois aspects. Premièrement, nous étudions des solutions flexibles garantissant la sécurité, l'intégrité et la redondance des données pour du stockage dans le cloud. En tirant parti des offres de stockage grand public, nous montrons qu'il est possible de conserver le contrôle du stockage dans le cloud depuis le client.
Dans un second temps, nous construisons une archive de données répartie dont la résilience va au-delà des techniques de redondance standards. Pour cela, nous implémentons Recast, un prototype usant du data entanglement, qui encode et répartit les blocs de données sur de nombreux nœuds de stockage afin d'en assurer la durabilité.
Enfin, nous examinons comment réduire l'augmentation des coûts de stockage entrainés par les méthodes proposées ci-dessus par de la dépduplication. Plus précisément, nous faisons usage de la Generalised Deduplication, une méthode dont les résultats vont au-delà de la déduplication classique grâce à une détection de similarité plus fine que la correspondance exacte.

Mots-clés: stockage réparti, systèmes de fichiers, codes correcteurs d'erreur, chiffrement, data entanglement, résistance à la censure, déduplication

Contents

1 Introduction **1**

1.1 Context and motivation 1

1.2 Contributions 3

1.3 Outline 4

2 Background and Related Work **5**

2.1 Redundant cloud storage 5

2.1.1 Replication in distributed environments 5

2.1.2 Erasure codes 6

2.1.3 Erasure codes in distributed environments 8

2.2 Secure cloud storage 9

2.2.1 Secure single-cloud storage 9

2.2.2 Secure multi-cloud storage 10

2.3 Censorship resistance 11

2.4 File system design and modularity 14

2.4.1 Stackable file systems 14

2.4.2 Software-defined storage 15

2.4.3 File systems in user-space 15

2.4.4 Privacy-aware file systems 17

2.5 Data reduction 18

2.5.1 Compression 18

2.5.2 Delta-encoding 19

2.5.3 Deduplication 19

3 On the Cost of Reliable and Secure Storage on Public Clouds 23

3.1 Introduction . . . 23

3.2 Security and availability requirements . . . 25

3.3 Deployment scenarios . . . 25

3.3.1 Single-cloud deployment . . . 26

3.3.2 Multi-cloud deployment . . . 26

3.4 Architecture . . . 27

3.5 Implementation details . . . 29

3.6 Configurations . . . 29

3.6.1 Erasure coding . . . 29

3.6.2 Security configurations . . . 31

3.7 Evaluation . . . 31

3.7.1 Evaluation settings . . . 32

3.7.2 Micro-benchmark — Erasure coding throughput . . . 32

3.7.3 Micro-benchmark — Erasure coding storage overhead . . . 34

3.7.4 Micro-benchmark — Secure drivers throughput . . . 34

3.7.5 Macro-benchmark — Single-cloud latency . . . 35

3.7.6 Macro-benchmark — Multi-cloud latency . . . 36

3.7.7 Macro-benchmark — Resource usage . . . 37

3.7.8 Macro-benchmark — Storage overhead . . . 38

3.8 Lessons learned . . . 39

3.9 Summary . . . 40

4 Random Entanglement for Censorship-Resistant Archival STorage 43

4.1 Introduction . . . 43

4.2 Design goals . . . 44

4.3 Data entanglement . . . 45

4.4 STeP-archival . . . 46

4.5 Hybrid entanglement . . . 49

4.6 Architecture . . . 53

4.7 Implementation details . . . 55
4.8 Evaluation . . . 55
4.8.1 Encoding/decoding throughput . . . 55
4.8.2 Metadata storage overhead . . . 57
4.8.3 Metadata reconstruction and documents availability . . . 57
4.8.4 Repair bandwidth costs . . . 58
4.9 Discussion . . . 58
4.9.1 Metadata service as a single point of failure . . . 59
4.9.2 GDPR compliance . . . 60
4.10 Summary . . . 61

5 SafeFS: A Modular Architecture for Secure User-Space File Systems 63
5.1 Introduction . . . 63
5.2 Design goals . . . 65
5.3 Architecture . . . 65
5.3.1 A day in the life of a write . . . 66
5.3.2 Layer integration . . . 68
5.3.3 Driver mechanism . . . 68
5.4 Implementation details . . . 69
5.4.1 Granularity-oriented layer . . . 70
5.4.2 Privacy-preserving layer . . . 70
5.4.3 Multiple back-end layer . . . 71
5.4.4 Layer stacks . . . 72
5.5 Configurations . . . 72
5.6 Evaluation . . . 73
5.6.1 Third-party file systems . . . 73
5.6.2 Evaluation settings . . . 74
5.6.3 Database benchmarking throughput . . . 74
5.6.4 Workload simulation throughput . . . 76
5.6.5 Runtime breakdown across layers . . . 77

5.7 Discussion . . . 78
5.7.1 Access control and key management . . . 78
5.7.2 Data processing in trusted execution environments . . . 79
5.7.3 End-to-end encryption . . . 80
5.8 Summary . . . 81

6 MinervaFS: User-Space File System with Generalised Deduplication 83
6.1 Introduction . . . 83
6.2 Generalised Deduplication . . . 86
6.2.1 A brief description of Generalised Deduplication . . . 87
6.2.2 Generalised Deduplication of the first file . . . 87
6.3 Architecture of MinervaFS . . . 88
6.3.1 Storing and loading data . . . 89
6.4 Implementation . . . 90
6.4.1 Processing . . . 90
6.4.2 Storage . . . 91
6.5 Evaluation . . . 93
6.5.1 Evaluation settings . . . 94
6.5.2 Storage usage . . . 94
6.5.3 Metadata overhead . . . 95
6.5.4 System throughput . . . 96
6.6 Discussion . . . 96
6.6.1 Granular encoding and decoding . . . 97
6.6.2 Garbage collection of unused bases . . . 97
6.6.3 Moving Generalised Deduplication up and down the stack . . . 98
6.6.4 Selecting type relevant transformation functions . . . 99
6.7 Summary . . . 100

7 Conclusion 101
7.1 Summary . . . 101

7.2 Perspective - Piecing it all together 103
7.3 Directions for future work 104

A Publications **105**

Bibliography **107**

List of Figures

2.1 XOR based codes 6
2.2 Reed-Solomon based codes 7
2.3 Communication between a user-space application and a FUSE file system. . 16

3.1 Architecture of Playcloud, our experimental testbed. 28
3.2 Micro-benchmark: Erasure codes - encoding and decoding throughput 32
3.3 Micro-benchmark: Erasure codes - reconstruction throughput 33
3.4 Micro-benchmark: Erasure codes - storage overhead. Storing 500 files of 4 MB. 34
3.5 Micro-benchmark: Secure drivers - encoding and decoding throughput. . . . 35
3.6 Macro-benchmark: Single cloud - average latency per block 36
3.7 Macro-benchmark: Multi-cloud - latency CDF over 3 cloud providers. 37
3.8 Macro-benchmark: Single-cloud - Storage overhead. 39

4.1 Dagster on top of `xor_rsa_sha_512` over 3 cloud providers. 45
4.2 Dagster on top of `cauchy_rsa_sha_512` over 3 cloud providers. 45
4.3 Entanglement logical flow with STeP. 47
4.4 Variation in the number of pointers to documents a uniform STeP-archive . 49
4.5 Variation in the number of pointers to documents a normal STeP-archive . . 49
4.6 Creeping attack on normal STeP-archives with a varying number of pointers 50
4.7 Leaping attack on hybrid STeP-archives with a varying number of pointers . 51
4.8 Most effective attacks on uniform, normal and hybrid STeP-archives 51
4.9 Backtracking the creeping attack on a n100-$(1, 5, 2, 3)$-archive 52
4.10 The Recast architecture and execution flow. 53
4.11 Encoding and decoding throughput 56
4.12 Writing and reading throughput 56
4.13 Metadata storage overhead 57
4.14 Metadata reconstruction 58
4.15 Repair bandwidth 59
4.16 Integration of MetaBlock with Recast 60

5.1 Architecture of SafeFS. 66
5.2 Flow of a write request in SafeFS 67
5.3 Layers and drivers available in SafeFS prototype 69
5.4 Throughput of db_bench workloads on top of SafeFS configurations 75
5.5 Throughput of filebench workloads on top of SafeFS configurations 76
5.6 Breakdown of time spent between the different layers of SafeFS configurations 78

6.1 Effect of chunk size on deduplication potential 84
6.2 Serial correlation coefficient in a HDFS log file 85
6.3 Compression ratios of Gzip, ZFS and MinervaFS on different types of datasets. 86
6.4 Generalised deduplication for the first file. 88

6.5 Architecture of MinervaFS 89
6.6 Effect of basis size on MinervaFS performance 90
6.7 Storage overhead comparison of MinervaFS and other systems 95
6.8 System throughput comparison of MinervaFS and other systems 97

List of Tables

1.1 Contributions and their use in this thesis 3

2.1 Survey of cloud-based storage systems 9
2.2 Survey of censorship-resistant storage systems 12
2.3 Survey of privacy-aware file systems 17

3.1 Erasure coding libraries under evaluation 30
3.2 Deployment targets, drivers and security guarantees 31
3.3 Storage overhead estimation 31
3.4 CPU and memory usage at encoding time 38
3.5 CPU and memory usage at decoding time 38

4.1 Elements of entanglement notation 48

5.1 Evaluation configurations for SafeFS 72

6.1 Storage capability of MinervaFS and ZFS based on chunk and registry size 93
6.2 Evaluation datasets for MinervaFS 94

Chapter 1

Introduction

1.1 Context and motivation

As the march towards digitization of the world continues, cloud storage must rise to the challenge of an increasing volume of data and more demanding requirements. In part due to the greater number of devices and sensors connected to the internet, projections set the amount of data stored in the cloud to 175 ZB by 2025 [223]. While these numbers give an abstract and distant idea of the challenge, they translate to concrete concerns of durability, confidentiality, flexibility and cost for all actors involved. For customers, safe archival and seamless synchronization of data across devices is no longer desired but expected. The livelihood of companies, especially in times of enforced remote work, depends on the ability of their employees to efficiently and securely access data. Finally, storage providers must be able to reliably and efficiently store all this data while offering their services at competitive prices. Satisfying customer requirements while maintaining operational efficiency is a complicated balancing act.

But despite the challenge, storage providers try to offer measures to ensure the confidentiality and integrity of their customer's data. Measures targeting confidentiality include encrypting customer data and segmenting their storage infrastructure to prevent rogue employees from snooping on customer data. Measures can also be taken to avoid data loss such as replicating files across data centers. But these measures keep the initiative on the provider's side and essentially require blind trust from the customers. This is likely to inspire skepticism from customers and thus negatively impact the provider's reputation [245]. And regardless of the good intentions of the providers, their systems can still fail. Negligence can lead to data loss as was the case of Swisscom losing personal cloud data for free-tier users [254]. And even when the provider is technically competent, the human side of operations can fall prey to social engineering leading to irreparable loss [114]. With their work and personal archives possibly disappearing overnight, customers may feel like relying on a single storage provider is too much of a risky choice. These concerns are not new but as the needs of customers evolve over time, the solutions put in place have to evolve accordingly. However, plugging a secure solution between existing applications and their storage backends can prove trickier than expexted. In a world that expects instant and seamless access to data, adding complexity to the processing pipeline may significantly affect user experience. Besides, beyond the performance aspect, there is no universal protocol to interface between existing applications and their storage stack. Empowering users with flexible client-side tooling to securely and reliably offload their data to the cloud is challenging but necessary for a secure cloud storage ecosystem.

Leaving aside customer data, some of the information stored online requires a stronger emphasis on availability and integrity than confidentiality. Every year, new items fall in the public domain, new documents are published by governmental institutions and snapshots of public facing websites are archived for preservation and research purposes. This data is archived and looked after by public and private initiatives like the Internet Archive [264]. And to operate on this data, legacy software has to be saved as well. In a world ruled by software, the need for its long-term preservation is such that it can lead to extreme solutions such as Github storing code in vaults in the arctic circle [174]. These initiatives ensure that precious documents remain available for future generations. However, the platforms hosting this content still have to comply with demands to delete what is deemed to be illegal or copyright infringing material. From that point on, the days of a targeted document on the internet are counted. While popular content can rely on the Streisand effect[1] and alternative networks to keep on living a little longer, unpopular content can just disappear off the face of the internet. And when refusing to delete the content also exposes the platform or provider to cyber attacks [124], removing the content seems like the best option to avoid exposure but also reduce their storage costs.

Indeed, the cost to host all this data is considerable. Files in storage must be replicated to avoid data loss, multiplying the storage overhead by a constant factor. The Internet Archive, for instance, preserves snapshots of websites, music, video and text. They currently store over 45 PB of content, of which they keep at least two copies, at an estimated cost of 2 $/GB [10]. In addition to replicated cold storage of data, providers also have to account for caching and load-balancing infrastructure. While most storage companies keep their cloud spend a secret, they often boast about the amount of data they store. An example of this is Backblaze that claims the storage of over 1 ExaBytes of data. With such numbers, finding ways to reduce the storage overhead and bandwidth use is essential to keep costs under control. Providers must then turn to data reduction techniques such as compression and deduplication to reduce the storage overhead.

In light of the context painted above, this work seeks to contribute by working along 3 angles.

1. Enabling secure and reliable cloud storage from the client-side
2. Building resilient archival storage in the cloud
3. Taming storage overhead with practical and efficient data reduction

First, we take a look at the client side integration of flexible tooling to securely and reliably offload data to the cloud. To do so, we look at ways to build flexible solutions interfacing with REST based and file systems APIs. Second, we look at how the distributed aspect of cloud storage can be leveraged to build archives with a level of resilience that goes beyond standard redundancy schemes. Finally, we look at how storage overhead can be tamed beyond with better compression gains than classic data reduction techniques without sacrificing on performance.

[1] `https://en.wikipedia.org/wiki/Streisand_effect`

Table 1.1: Contributions and their use in this thesis

Chapter	Summary	Publications
Chapter 3	Benchmarking of cryptographic and coding schemes on top of public cloud storage.	[32, 34, 31]
Chapter 4	First and fully functional implementation of STeP	[17, 141]
Chapter 5	Implementation and evaluation of a framework for stackable user-space file systems	[207, 33]
Chapter 6	Integration and evalution of a data deduplication scheme in user-space file system	[183]

1.2 Contributions

This manuscript is the product of contributions made across 6 conference papers (1 currently under review), 1 short paper and 1 PhD forum presentation. The contributions can be described as follows:

- A benchmark of erasure coding libraries in a distributed setting [32]. This work was done under the supervision of Pascal Felber and Valerio Schiavoni with the collaboration of Hugues Mercier.
- Playcloud [31], an open-source platform to benchmark erasure codes in a distributed setting.
- A benchmark of cryptographic and erasure coding schemes on top of public cloud storage [34]. This work was done in collaboration with Rogério Pontes, Francisco Maia, João Paulo, Rui Oliveira and Hugues Mercier under the supervision of Pascal Felber and Valerio Schiavoni.
- The introduction of Recast [17], a fully functional implementation of STeP [14], a coding scheme that blends erasure coding for redundancy and data entanglement for censorship detection. This work was led by Roberta Barbi under the supervision of Pascal Felber, Valerio Schiavoni and Hugues Mercier.
- MetaBlock [141] an extension to Recast that backs up metadata in a private Ethereum blockchain. This work was led by Arnaud L'Hutereau under the supervision of Pascal Felber, Valerio Schiavoni and Hugues Mercier.
- SafeFS [207], a modular framework for stackable file systems in user-space. This work was led by Rogério Pontes with the collaboration of Francisco Maia, João Paulo, Rui Oliveira, Pascal Felber, Valerio Schiavoni and Hugues Mercier.
- Sgx-Fs [33] a user-space in-memory file system offering insights into the trade-offs of delegating the encryption work to a specific Trusted Execution Environment: Intel SGX [120]. This work was supervised by Pascal Felber and Valerio Schiavoni with the collaboration Hugues Mercier.
- MinervaFS [183] A user-space file system demonstrating the efficiency and practicality of Generalised Deduplication in file systems. This work was led by Lars Nielsen with the collaboration of Daniel Lucani, Pascal Felber and Valerio Schiavoni.

1.3 Outline

The chapters of this work are organized as follows. Chapter 2 introduces the necessary background by giving a primer on redundancy schemes, secure cloud storage, modular file system design and data reduction.

In Chapter 3, we build a platform on top of public cloud storage providers to benchmark a series of coding schemes integrated on the client-side. We start with a presentation of different erasure coding schemes and libraries. We then evaluate their implementations to measure throughput and storage overhead trade-offs. To guarantee the security of the data, we then blend in cryptographic schemes. We combine erasure coding with encryption, hashing and cryptographic signatures to form a better idea of their cost in practice. To do so, we measure the impact of the different combinations on resource consumption, throughput and storage overhead. Overall, the results show, as expected, that combinations should be picked with care for the workload they support. In particular, we observe an important increase in CPU usage for most combinations and a significant increase in latency for combinations including redundancy.

Chapter 4 shifts the focus from personal cloud storage to archival storage in the cloud. In this context, we look at how documents stored in the cloud can be protected from accidental loss of servers and censorship attempts by the use of data entanglement. In particular, we study a specific flavor of data entanglement STeP [14]. We discuss improvements over the existing scheme and present Recast, the first full implementation of STeP built on top of distributed storage. We then put our prototype through evaluation to measure its throughput, storage overhead, repair bandwidth use and metadata storage overhead. To close the chapter, we discuss the shortcomings of our prototype and its positioning regarding current international legislation.

In Chapter 5, we take another look at flexibility and the combination of coding schemes through the lens of file systems in user-space. More specifically, we design and implement SafeFS, a framework to build modular and configurable file systems in user-space. By providing a layered approach to file system design, SafeFS enables the construction of customizable storage configurations where layers can be reordered and powered by a variety of drivers. To put this prototype to the test, we build a set of security-oriented configurations and compared their performance against monolithic and secure user-space file systems. Our results show that SafeFS configurations fare comparably to the other monolithic file systems in user-space with the benefit of requiring no recompilation to change their behavior. The chapter is closed with a discussion on the limitations of our prototype and the challenges of building a completely secure file system on top of the FUSE framework.

After placing much of the processing work on the client-side (and in user-space) by adding redundancy and security, Chapter 6 tackles the issue of storage overhead. This chapter presents MinervaFS, an implementation of Generalised Deduplication (GD) in a user-space file system. We start with a description of GD and how it differs from classic deduplication. We then present the architecture and design of our file system before putting it under evaluation. We close this chapter with a discussion on the integration of GD in a file system and the potential for improvements in our prototype.

Finally, Chapter 7 concludes this work with a look back at the topics covered and lessons learned along the way.

Chapter 2

Background and Related Work

The design of reliable and efficient storage systems is essential to make the most out of local or cloud storage. But allowing for flexibility in design is also necessary to accommodate varying use cases and workloads. Namely, users should be confident that the integrity and confidentiality of the data they entrust to the cloud is maintained without significant impact to their workloads. When such a compromise cannot be reached, users should be able to select and build feature sets that match their expectations with reasonable efforts.

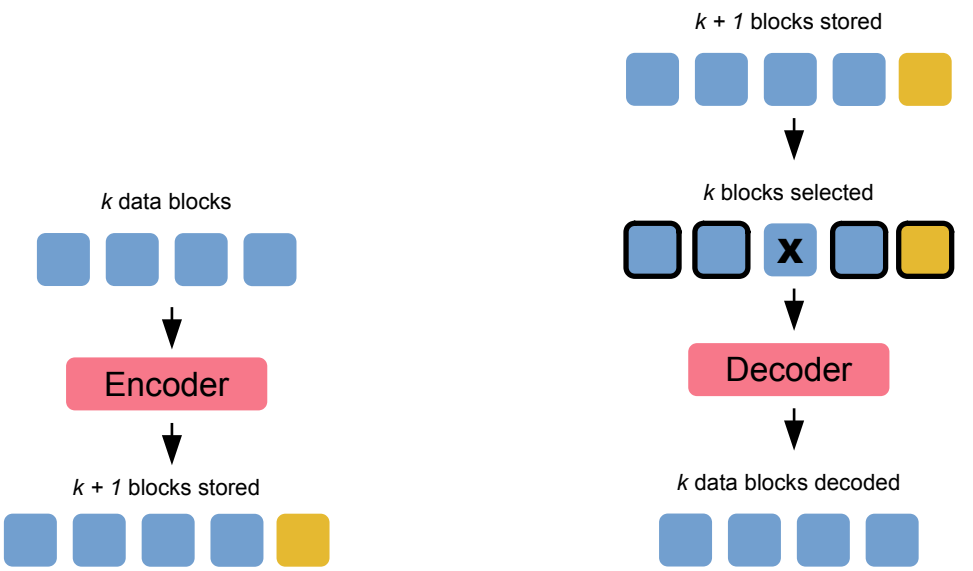
In this chapter, we take a look at state-of-the art methods that enable both users and providers to make the most of cloud storage. To this end, Section 2.1 looks at how redundant storage techniques can be used to avoid data loss at reduced storage and repair costs in distributed storage environments. Section 2.2 presents a list of storage systems built on top of consumer-grade cloud storage with a focus client-side control of data confidentiality. Section 2.3 discusses distributed archival storage systems resistant to censorship. Section 2.4 presents the tooling available for the modular design and re-usability of file systems. Finally, Section 2.5 presents data reduction technique that may be used by the cloud storage provider for final storage of the data.

2.1 Redundant cloud storage

Storage systems rely on the quality of the underlying medium to ensure data durability. In case of failure of the storage medium, the system must be able to recover the data. An obvious way to prevent the loss of data is to replicate the data on independent storage media. In case of failure of one these media, a copy can be recovered from an independent source. The cost of a recovery is simple to compute in terms of I/O and bandwidth since is proportional in the size of the data to recover. However, the simple repair process also implies the presence of a number of copies and thus some storage overhead : n copies of the data requires n times the storage space. To counter replication's costly storage overhead, the research field of the error correcting codes has come up with interesting solutions. The rest of this section presents research in replication at scale, the basics of erasure codes and state-of-the-art research to make them work in a distributed setting.

2.1.1 Replication in distributed environments

Full-replication is the simplest solution to add redundancy and prevent data loss. The cost of storing copies is straightforward: data of size r replicated s times requires $s \times r$ the storage space. From these numbers, it is easy to understand that full-replication can be costly. The main challenge of replication in distributed settings is the dispersion and maintenance



(a) Redundancy using XOR.

(b) Failed disk recovered using XOR.

Figure 2.1: XOR based codes

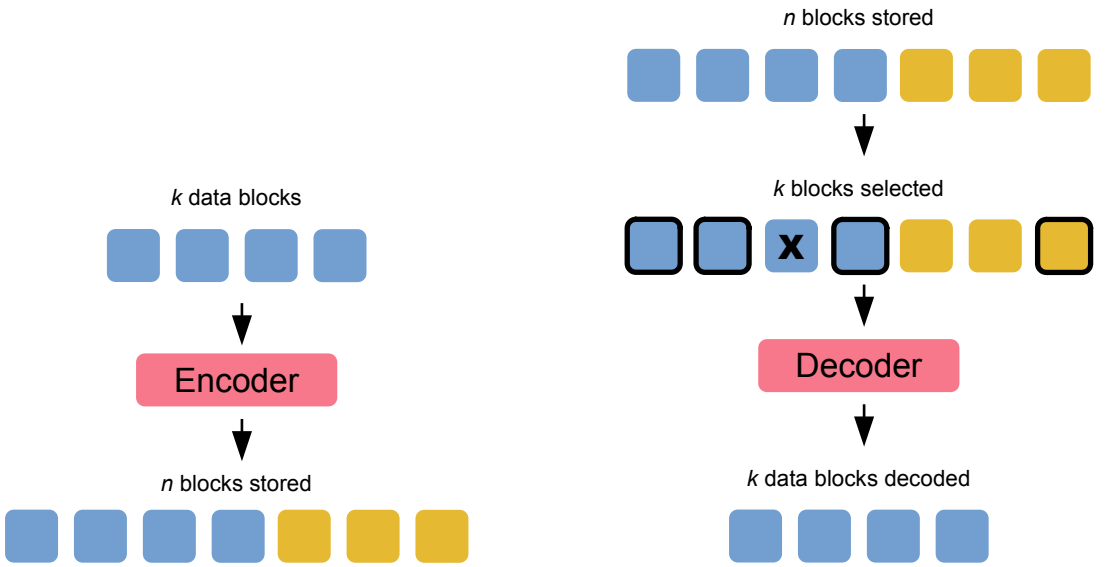
of replicas. Replicas must also be kept independent in order to avoid data loss in case of correlated failures. A common way to prevent this issue is to perform random replication. It is a common method used in distributed hash tables, databases and distributed file systems. By picking a set of r random storage nodes to store a chunk of data, the system ensures that the amount of data loss when a storage node fails is predictable and uniform. Copyset replication [43] chooses to pre-select sets of r, sometimes overlapping, storage nodes according to some balancing properties in order to maximize data availability at a greater per node repair cost. Considering cheaper storage media such as tapes may be a solution but what is gained in storage cost is paid in latency. In order to maintain availability, most setups are built on top of n identical replicas. Compromising between faster and slower replicas [42] shifts the traditional 3 equal replicas to a 2 primary + 1 secondary. This maintains durability at a lower cost.

2.1.2 Erasure codes

The objective of error-correcting codes for data storage is to carefully add redundancy to data in order to protect it against corruption when stored on media like DVDs, magnetic tapes or solid-state drives. In these systems, the errors are usually modeled as erasures, meaning that their locations are known. Consider the example shown in Figure 2.1a, where a coding disk is used to store the XOR ("exclusive or") of k data blocks. If the system realizes that one of the disks has failed, as shown in Figure 2.1b, it can XOR the healthy disks and recover the failure. This is the maximum decoding capability of this code, and there will be data loss if more than one disk fails.

In general, k data blocks are coded to generate $n - k$ coding blocks, as illustrated in Figure 2.2a. After disk failures, the system will try to decode the original codewords from the healthy disks, like in Figure 2.2b. The number of recoverable disk failures depends on the code itself.

The most famous class of erasure codes are Reed-Solomon codes (RS), first introduced in 1960 [222]. An (n, k) Reed-Solomon code is a linear block code with dimension k and



(a) Generic erasure code encoder.

(b) Generic erasure code decoder.

Figure 2.2: Reed-Solomon based codes

length n defined over the finite field of n elements. Reed-Solomon codes have many interesting properties. First, they achieve the singleton bound with equality, and thus are maximum distance separable (MDS) [150]. In other words, they can correct up to $n-k$ symbol erasures, *i.e.*, any k of the n code symbols are necessary and sufficient for decoding. Second, using a large field, they can correct bursts of errors, thus their widespread adoption in storage media where such bursts are common. Encoding and decoding Reed-Solomon codes is challenging, and optimizing both operations has kept many coding theorists and engineers busy for more than 50 years. There is a large amount of literature on these topics, covering theory (*e.g.*, [105]) and implementation (*e.g.*, [232]), but in a nutshell the best encoding and decoding implementations are quadratic in the size of data.

Reed-Solomon codes, while storage-efficient, were not originally designed for distributed storage and are somewhat ill-suited for this purpose. Besides their complexity, their main drawback is that they require at least k geographically distributed healthy disks to recover from a single failure, followed by decoding of all the codewords with a block on the failed disk. This incurs significant bandwidth and latency costs. This handicap has led to the development of codes that can recreate destroyed redundancy without decoding the original codewords. The trade-offs between storage overhead and failure repairability is an active area of research [65, 188], and there are many interesting theoretical and practical questions to answer. Among other work of interest, NCCloud [40] reduces the cost of repair in multi-cloud storage if one cloud storage provider fails permanently. We also mention the coding work done for Microsoft Azure [36, 117], and XOR-based erasure codes [131] in the context of efficient cloud-based file-systems exploiting *rotated* Reed-Solomon codes. RAID-like erasure-coding techniques have been studied in the context of cloud-based storage solutions [131]. Plank *et al.* [203] studied chosen erasure-coding libraries, which has in great part motivated and inspired the first part of Chapter 3.

2.1.3 Erasure codes in distributed environments

The development of online cloud storage services has sparked a new flame in research efforts to make erasure codes work in the context of distributed environments. In particular, the efforts have focused on minimizing the use of resources during the repair process. Three aspects of the repair process are listed: the number of storage nodes involved in the repair process (locality) [100], the number of bits transiting over the network (bandwidth) [65] and the number of bits read on disk (I/O) [132]. Most of the research on these issues has focused on delivering lower and upper bounds depending on code and system configurations.

To tackle these issues, numerous codes categorized under the family of *Locally Repairable Codes* (LRC) [198, 135] have been proposed since the late 2000s. Locally repairable codes strive to minimize the amount of blocks fetched, and thus the number of storage nodes involved, during degraded reads or block repairs. This locality gain is achieved by introducing *local* parities resulting from the combination r blocks from the original codeword (where $r < k$). Lost blocks can then be recovered by reading r blocks rather than k. These LRCs, notation (n, k, r), are usually based on (n, k) Reed-Solomon codes where groups of at most r blocks in the codeword are XORed together to produce local parities. A distinction is sometimes made between *data-LRCs* that produce local parities only for the data blocks and *full-LRCs* that produce local parities for all blocks in the codeword [135]. It is important to understand that even though LRCs offer better locality, they are not Maximum Distance Separable (MDS). Indeed, while these codes can in theory recover from up to $r + (n - k)$ failures, they cannot however handle arbitrary failure patterns. The failure patterns that can be recovered from are sometimes called *information-theoretically decodable* [117].

Notable LRC examples include Pyramid codes [116], Xorbas [233] and Optimal-LRC [256]. Pyramid codes, defined (k, l, m) [116], are data-LRCs that improve locality by splitting the data in k blocks, generating m global parities and generating local parities for groups of l data blocks. Pyramid codes were tested in Windows Azure Storage [117] in a $(12, 2, 2)$ configuration which translates in the (n, k, r) notation introduced before into $(14, 12, 6)$. Starting from a (14,10) Reed-Solomon code, Xorbas [233] generates 3 extra parities: one per 5 data blocks and a third combining these two new parities and the Reed-Solomon parities. For this 14% increase in storage overhead, Xorbas reduces a 2 $\times$ reduction in disk I/O and repair network traffic compared to a (14,10) Reed-Solomon code. Optimal-LRCs [256] create a comparable construction to the other (n, k, r) schemes but manage to lower the minimum distance of the code with some extra constraints where n must be divisible by $r + 1$.

Finally, other constructions focus on reducing read and repair latency. Agar [106] proposes suitable caching policies for globally distributed erasure coded data. And HACFS [291] to significantly reduce latency of degraded reads, disk I/O and network traffic by combining a fast code for fast repairs and a compact one for long-term storage overhead.

To summarize, numerous redundancy schemes can be used to prevent data loss in distributed environments. Replication is trivial and makes system administration simple but reduces the optimal usage of storage space. Erasure codes offer a good alternative with lower storage overhead and good fault tolerance but require fine-tuning when it comes to repairs. For this reason, erasure coding schemes offer different trade-offs between storage overhead, bandwidth usage, disk I/O and latency. Ultimately, the choice of a scheme falls on the system operator's priorities and their ability to combine their domain knowledge with the infrastructure at hand.

Table 2.1: Security features offered by secure cloud solutions for single and multi-cloud (top and bottom respectively) solutions: confidentiality, self-integrity, origin authentication, censorship-resistance, symmetric encryption, asymmetric encryption, hash functions.

System	Confidentiality	Self integrity	Origin authentication	Censorship resistance	Symmetric encryption	Asymmetric encryption	Hash function
CloudProof [208]	✓	✓	✓	✗	AES	RSA	SHA-1
Kamara *et al.* [128]	✓	✓	✓	✗	AES	✗	✗
Depot [161]	✗	✓	✓	✗	✗	RSA	SHA-256
SUNDR [147]	✗	✓	✓	✗	✗	ESIGN	SHA-1
BlueSky [278]	✓	✓	✗	✗	AES	✗	SHA-256
Hail [28]	✗	✓	✗	✗	✗	✗	Univ. Hash
MetaSync [108]	✓	✗	✓	✗	AES	✗	MD5
DepSky [21]	✓	✓	✓	✗	AES	RSA	SHA-1
UniDrive [257]	✓	✓	✓	✗	DES	RSA	SHA-1
SafeSky [305]	✓	✓	✗	✗	CCM	✗	✗

2.2 Secure cloud storage

.

Offloading the storage of data to cloud providers implies a considerable loss of control. While the previous section focused on data integrity, the confidentiality aspect must also be addressed. Numerous solutions have been proposed during the past years to target the challenges of secure cloud storage over in commercially-available online solutions. This section splits proposals in two categories: single-cloud and multi-cloud. We start by discussing approaches that rely on a single storage provider in Section 2.2.1. Then, in Section 2.2.2, we discuss federated storage systems that split data across multiple providers. Table 2.1 presents a summary survey of these two types of systems.

2.2.1 Secure single-cloud storage

Unable to provide availability guarantees beyond the provider's own, single-cloud systems still make use of hashing and signature algorithms on top of the standard encryption to verify data integrity.

SUNDR [147] proposes an architecture that leverages asymmetric encryption and cryptographic hash function to ensure the integrity and consistency of stored data (blocks). In particular it uses SHA-1 digests to index each data block and a protocol based on ESIGN [189] to detect unauthorized attempts of file modifications.

Depot [161] offers stronger liveness guarantees under node failure than SUNDR, however it lacks native support for data confidentiality or privacy. Data is stored in plain-text along with SHA-256 message digests to enable data integrity checking. Also, data is

cryptographically signed using RSA with 1024-bit keys to ensure data authenticity. Both signatures and digests are verified upon each read request.

CloudProof further adds an encryption step: data blocks are protected with AES [208]. Once encrypted, blocks are signed with RSA to prevent unauthorized users to tamper with the data. Also, data integrity is assured by means of SHA-1 digests. The authors present further proposes a strategic approach to use CloudProof's guarantees in the service level agreements.

Kamara *et al.* [128] discuss a high-level architecture for a storage service that can be implemented with different cryptographic primitives, thus offering different security features. Moreover, data privacy is ensured by using symmetric/asymmetric ciphers.

BlueSky [278] tackles the privacy and integrity problem of enterprises with a proxy server that handles the communication between the client and cloud provider. This proxy is installed in the enterprise network so clients do not require any modification. The proxy encrypts the clients data and checks the integrity of the files retrieved from the cloud provider.

2.2.2 Secure multi-cloud storage

The previous solutions store all information in a single storage service. By doing so, users are locked to a specific storage service that also represents a single point of failure, both for data availability and security breaches. We can mitigate these drawbacks by resorting to multiple cloud providers.

MetaSync replicates users' files to tolerate data loss and the unavailability of storage providers [108]. An additional plug-in can also be used to conceal data using AES encryption.

Hail [28] adopts a secure multi-cloud approach. While it does not provide native support for data confidentiality, it handles data integrity and recoverability from node failures by using a single trusted verifier. This verifier can be a client or a proxy that performs a periodic check of the files integrity on the providers and reconstructs corrupted blocks.

In DepSky [21] and Unidrive [257], data is balanced and replicated across multiple providers with MDS erasure coding [150]. A data object is split into k blocks and coded to generate n coded blocks which are then spread across the cloud providers. The user can reconstruct the original data object from any subset of k out of n the blocks. This decreases the storage overhead compared to replication approaches like the one used in MetaSync, while achieving the same reliability level. Unidrive uses non-systematic Reed-Solomon codes so that data is not directly stored online. DepSky goes further by encrypting the data before encoding it. The secret key is then divided using secret sharing [240], and each server receives one code block and one share of the key. This insures that any malicious entity gaining access to less than k blocks obtains no information whatsoever about the original data object. Unidrive encrypts the metadata with DES and replicates it on all clouds, whereas DepSky signs metadata files using AES.

SafeSky [305] provides a middleware layer at the operating system level that intercepts file system calls and redirects storage requests to cloud providers. The data object is first encrypted. The encrypted object, the secret key, and the cipher type are then divided using secret sharing.

Hybris [67] proposes a Byzantine Fault Tolerant (BFT) [142] solution based on a hybrid model. It splits the trust domain in two parts: a private cloud (or machines on company premises) to store metadata and public clouds to store the data. It builds on the idea that machines on premises are not prone to Byzantine faults where public clouds are. Hybris ensures strong consistency on the metadata by using a replicated Zookeeper [263] cluster and

uses redundancy in order to tolerate faulty or malicious behavior from cloud providers. This setup makes it possible for Hybris to offer BFT properties at lower redundancy costs: from $f + 1$ to $2f + 1$ instead of the usual $3f + 1$ to tolerate up to f failures.

CYRYUS [41] provides a multi-cloud storage solution centered on client-defined security requirements and a versioning-based conflict detection mechanism. It splits files into chunks based on their content and processes the chunks into shares through a keyed Reed-Solomon in a similar approach to secret sharing. The shares are then distributed across the storage providers. A single metadata file for each file is processed split and distributed similarly. The number of shares k is determined by a user-defined threshold ϵ and the number of storage providers n. CYRUS also maintains information about the latency, upload and download bandwidth on the links to cloud providers in order to pick the best candidates for performance. For better fault tolerance, CYRUS also tries to determine through route tracing whether different cloud storage providers are truly independent. Since CYRUS does not rely on any specific coordination service, it allows client to write to the same files by having file updates written as new versions of the file (linked to the previous one). Whenever a branching appears in the history of a file, the client must resolve the issue manually.

CHARON [171] is a multi-cloud storage solution built for big data that exposes a near POSIX interface built for Byzantine fault tolerance. Relying on the constructs proposed by DepSky [21], CHARON stands out from other solutions by relying solely on publicly available cloud services, both for data, metadata and coordination. More specifically, it implements lease algorithms usually built on top of existing coordination services, such as Zookeeper [263], on top of message queuing products offered by the cloud services removing the need for maintenance of additional virtual machines. While multiple clients can read the same file concurrently, they must obtain a quorum of leases in order to write or update a file.

Some systems offer censorship-resistant storage by creating dependencies, or entanglement, across stored data [172]. Such entanglement makes it difficult for unauthorized parties to censor or tamper with data, but usually require modifying the implementation of storage backends. In this paper we implement, deploy and evaluate entanglement techniques atop unmodified third-party public cloud storage providers.

To summarize, the solutions listed in this section leverage the storage capabilities of one or multiple cloud providers to ensure the integrity and availability of stored data. Each solution uses a set of known and well-established techniques to guarantee the confidentiality and detect tampering attempts. However, the system impact of choosing these specific techniques over other options is largely ignored. This means that users lack information on the cost of combining these techniques. Chapter 3 tries to provide answers to these questions.

2.3 Censorship resistance

The problem of anti-censorship for digital data has been first explored in [8] and subsequently extensively studied in *e.g.* [252, 45, 138, 165, 279, 66]. We can broadly distinguish between three main approaches: (i) *replication* to protect against a censor compromising a small number of servers [8], (ii) *anonymity* of the user to hide their identity and/or of the content to hide its location [165, 66], (iii) *entanglement* to prevent an attacker from deleting a single target document without causing collateral damage [252, 279].

To prevent censorship a system must detect that data has been compromised and be able to repair it if needed. While tamper detection is a basic feature offered by all

Table 2.2: Survey of censorship-resistant storage systems. All provide tamper-evidence features. •=missing information.

System	Type	Anonymous upload	Anonymous download	Server deniability	Fault tolerant	Caching	Open source	REST API (`put`/`get`)	Support for `update`	Support for `delete`	Recursive repair
Dagster [252]	(iii)	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗
DeepStore [295]	(i)	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗
FreeHaven [66]	(ii)	✓	✓	✓	✓	✗	✗	✓	✗	✗	✗
Freenet [45, 44]	(i)	✓	✓	✓	✓	✓	[91]	✓	✗	✓	✗
OceanStore [224]	(i)	✗	✗	✓	✓	✓	[187]	✓	✓	✗	✗
Potshards [250]	(i)	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗
Publius [165]	(ii)	✓	✗	✓	✓	✗	[211]	✓	✓	✓	✗
SafeStore [138]	(i)	✗	✗	✗	✓	✓	✗	✓	✓	✓	✗
Tangler [279]	(iii)	•	•	✗	✓	✗	✗	✓	✓	✗	✗
RECAST	(iii)	✗	✗	✓	✓	✗	[47]	✓	✗	✗	✓

censorship-resistant systems, data reconstruction has received much less attention: it is either not supported by the system [252] or implemented trivially by fetching a fresh replica of the corrupted data [45, 165, 279], assuming one exists. In the remainder of this section, we survey the systems that directly offer storage-based anti-censorship properties, synthetically presented in Table 2.2.

Dagster [252] is a censorship-resistant publishing system. It uses an anonymous channel between data publishers/consumers and the servers. For each b-bits block of the original document, it stores the block encrypted and XORed with c old blocks from the pool of archived blocks. Such newly created block together with the c old blocks form a codeword. The original block can be locally retrieved (from the codeword) only if all blocks of the codeword are available. Similar to RECAST, the size b of blocks and the number of entangled blocks c can be configured but RECAST offers a much wider choice for coding parameters, thus enabling greater fault tolerance. Encryption guarantees server deniability only as the encryption keys are stored in clear in the Dagster resource locator.

Similarly, Tangler [279] allows users to publish a set of documents anonymously. It uses a naming convention and a public/private key pair that ensure that only the owner can update their own data. Each document block is entangled with exactly 2 old blocks β_1, β_2 using (3,4)-Shamir secret sharing [240]. The two parity blocks in output p_1, p_2 are then stored in the system. The owner distributes p_1 and p_2 (as well as the old blocks β_1 and β_2 used for entanglement), thus replicating other documents. Any three of these blocks can be used to reconstruct the original document. In contrast, RECAST provides a more flexible scheme where the number of blocks required to retrieve a document depends on the configuration of the entanglement code in use.

Freenet [45, 44] is an anonymizing and censorship-resistant distributed storage system. To guarantee file anonymity, nodes know only their neighbors in the chain of queries, ignoring their specific role (producer/consumer of the file). Freenet lacks permanent storage guarantees:

the least recently used files are deleted from a node's datastore when the arrival of a new file causes the datastore to exceed the designated size, which is configurable by the Freenet node operator. RECAST on the other hand is designed to store documents indefinitely, making it prohibitively difficult to delete a single document. Freenet places multiple copies of the same document across the nodes overlay, according to their popularity. A censor must target all those copies to permanently delete a document. RECAST makes it very hard for a censor to delete a single document without affecting a large portion of the archive.

FreeHaven [66] is an anonymous publishing system enabling users to associate an expiration time to documents, similarly to Comet [96]. It embeds a reputation system that relies on the capacity of nodes to store documents until their expiration. In contrast, RECAST assumes to always have sufficient storage capacity, and documents never vanish. Moreover, RECAST is not intended to provide anonymity, an orthogonal concern to be handled by upper layers.

Publius [165] encrypts and spreads documents over a set of (static) servers. The data encryption key is secret-shared among k of the n servers using (k, n)-Shamir secret sharing. Each server hosts the encrypted Publius content and a share of the key. The content can be tamper-checked because it is cryptographically tied to the Publius address: any modification of one of these two components will cause the tamper check to fail. RECAST implements tamper-check by exploiting decoding in document reads: if any block is corrupted, then the decoding fails and reconstruction is triggered.

Mnemosyne [109] is a steganographic peer-to-peer storage system. Nodes are continuously filled with random data. Upon archival of a real document, its content is encrypted and made indistinguishable from the random substrate, preventing an attacker to determine the existence of a file and ensuring privacy and deniability of the content itself. As in Publius, data is spread across nodes [216] mainly for resiliency and each of the nodes is unaware of the other nodes holding parts of the file. RECAST tolerates a stronger threat model where storage nodes have access to metadata information.

OceanStore [224] is a persistent data store on top of an untrusted infrastructure. It uses promiscuous caching (data is cached everywhere and at any time) to enhance locality and availability. This solution relies on classic erasure coding techniques, specifically a Cauchy Reed-Solomon code of length 32 and dimension 16 [160], to ensure durability.

DeepStore [295] is a large-scale archival system for immutable data. It achieves a good efficiency/redundancy compromise by applying compression with inter-file dependencies and replicating the most valuable pieces of compressed information. DeepStore relies on the Presidio framework [296] to implement hybrid compression across heterogeneous data, as well as deduplication to eliminate redundant data.

Lockss [164] is an archival system based on a voting protocol. The "opinion pools" provide system peers with confidence in content authenticity and integrity. Moreover, content is replicated among peers and replicas are regularly audited to promptly repair any damage. RECAST implements temporary replication to protect recently inserted documents and uses an auditing subsystem to remove temporary replicas from the system as soon as blocks are sufficiently entangled.

SafeStore [138] is a distributed storage system offering long-term data durability. It exploits an aggressive isolation scheme across multiple storage service providers. It uses an informed hierarchical erasure coding scheme that maximizes data durability and provides redundancy in the fault-isolated domains. Depending on the code's parameters, a certain number of faults can be recovered in each fault-isolated domain and an auditing subsystem further monitors data loss and triggers reconstruction. RECAST and Safestore sit on opposite

sides from an architectural point of view: while RECAST uses entanglement, SafeStore exploits aggressive isolation, which leads to high durability but does not offer strong resilience against an active censor.

Percival [89] and Potshards [250] are two systems relying on secret-splitting techniques. The latter offers long-term security by using two levels of secret splitting and placement. The first level provides secrecy by XORing content with random data. It produces n fragments using (k, n)-Shamir secret sharing and places them into shards for availability. Shards retrieval is based on indexes accessible by all users, in a similar manner to RECAST's metadata indexes. Potshards uses *approximate pointers* to allow for quick reconstruction of user data without the need of external information, similar to RECAST's emergency recovery procedure.

Finally, we include the system at the heart of Chapter 4: RECAST. The design of RECAST is based on STEP-archival [173]. STEP-archival is the theoretical foundation of RECAST and provides tools such as greedy attack heuristics and recursive reconstruction that we exploit in this work. We elaborate on such a base to build a *practical* STEP-archive. In practice, we overcome the poor short term protection of the uniform random entanglement studied in [173] by developing new entanglement heuristics.

2.4 File system design and modularity

Storage solutions, cloud-based or not, provide a rich set of features covering all sorts of redundancy, privacy or cost saving needs. However, it is often the case that only a select subset of these features are relevant or beneficial to our workloads. In addition, monolithic designs and vendor incompatibilities, make combining features from different storage solutions a difficult endeavor. Worse, enforcing end-to-end policies (storage bandwidth consumption, I/O request latency, etc) across various feature layers is a complicated task, since all layers must know the details of each request and enforce the desired policies. And yet being able to selectively assemble and enable the right features is key to running efficient and successful jobs.

In Chapter 5, we propose SAFEFS, a modular architecture for file systems in user-space. There is a large body of literature and numerous deployed systems to which SAFEFS relates to. This section presents a survey of related systems and frameworks, both in kernel- and user-space, with a special focus as well as a review of file systems in user-space with privacy-aware features.

2.4.1 Stackable file systems

Considerable efforts have gone into the design of frameworks and tooling for a layered approach to building file systems. In this subsection, we discuss the different frameworks available and how they relate to SAFEFS.

Stackable file systems [111] decouple data processing into multiple layers that follow a common interface. As each layer is independent and isolates a specific data processing stage, these file systems can be extended with additional layers. Kernel-space stackable file systems apply this concept by extending the classic *vnode* interface [169]. This design is used by CryptFS and NCryptfs [298, 289] to enhance file systems with access control, malware detection, and privacy-aware layers. NCryptfs extends the security features presented in Cryptfs, while both are built on top of WrapFS [299], which proposed *stackable templates* to ease the development of stackable file systems. Stackable templates have a limited API

and are bound to a single platform. These limitations were mitigated by FiST [300], a high-level language to define stackable file systems and a correspondent cross-platform compiler. Nevertheless, even with a high-level language abstraction, implementing layers with complex behavior (*e.g.*, predictive caching, replication) at the kernel level is an extremely challenging task. Furthermore, when any of the layers changes the size of the file being processed (*e.g.*, compression/decompression layers, cryptographic primitives with padding), these systems require a system-wide global index that tracks the offsets and the size of data blocks written at the final file system layer. This indexing requires a global metadata structure to keep track of the real block and file size, thus introducing dependencies between layers.

SafeFS relies on the FUSE API for layer stacking, which provides a richer set of operations and avoids the aforementioned problems. Each layer is independent of its adjacent layers while still supporting size-changing processing.

2.4.2 Software-defined storage

Our design draws inspiration from recent work in software-design storage (SDS) and network (SDN) systems [37]. These proposals make a clear distinction between control and data stage planes. The control plane is logically centralized, and it has a global view of the storage infrastructure in order to dynamically manage all data stage layers that correspond to heterogeneous storage components. This is the case for IOFlow [265], whose goal is to ensure a given quality of service (QoS) for requests from virtual machines (VMs) to storage servers. A common set of API calls is implemented by both control and data stage layers. Through these calls, the control plane is able to enforce static policies to a set of VMs. Notable policy examples are for prioritizing I/O requests, defining a minimum bandwidth or maximum latency usage, and routing I/O requests through specific data stage layers.

The control plane can also be used to enforce policies not directly related to QoS metrics [7]. Notably, data stage components can be stacked and configured in an workload-aware fashion, thus supporting distinct storage workloads even in a dynamic setting. Moreover, SDS proposals are able to dispatch data requests toward multiple storage layers or devices in a flexible and transparent way for applications using the storage stack [249].

This work focuses on the vertical data stage to provide a flexible solution for stackable file systems. This design enables (1) easy integration of existing file system implementations and (2) significant speedup in the development of novel file system implementations that require a specific set of storage optimizations. In the future, we plan to integrate and extend SafeFS with the control plane design and an extensive set of policies, similar to the OpenStack Swift storage component [101].

2.4.3 File systems in user-space

Traditional file system development on UNIX systems usually takes the form of kernel modules that are then loaded at startup or runtime. However, it is also possible to implement file systems in user-space using FUSE [94]. FUSE is both a library against which a developer builds a custom file system [149] and a kernel module [48] that forwards calls to the custom file system. A 2015 study [259] observed that at least 100 user-space file systems have been published between 2003 and 2015 while only about 70 kernel-based systems appeared in the previous 25 years. The same report showed that porting file systems functionalities to user-space reduced the time required to develop new solutions and improve their maintainability, reliability, extensibility, and security. An obvious advantage for user-space implementations

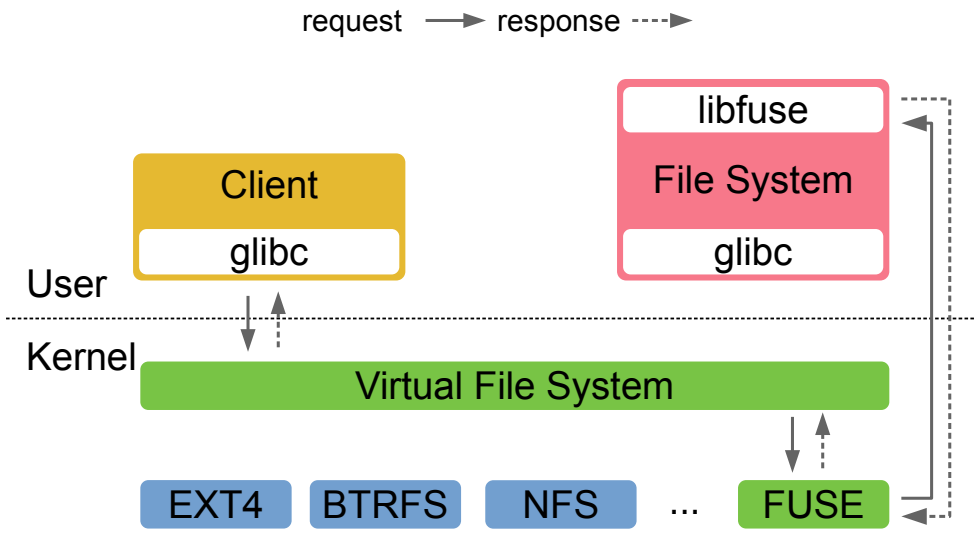


Figure 2.3: Communication between a user-space application and a FUSE file system.

is to have direct access to efficient user-space software libraries [20, 145]. Bridges can also easily be built to connect to commercial cloud storage services [228, 57, 98], self-hosted solutions [247, 97, 180] or even local file systems whose license conflicts with the kernel [70, 227] on linux [267, 218].

But despite the argument made for the relatively good performance of FUSE [259, 272], some developers remain skeptical due to the extra context switches between kernel- and user-space. To address these concerns, several improvements have been proposed [122, 24, 294].

In the context of high performance scientific computing with a distributed file system, [122] proposes a direct mapping of local files to the FUSE kernel module. Access to files on the same machine no longer requires exchanges between the kernel and the user-space daemon since the kernel module can directly serve the blocks requested by the client applications.

ExtFUSE [24] offers user-space file systems the possibility to register extensions to run in kernel mode. These extensions, written for the extended Berkeley Packet Filter (eBPF) virtual machine [104], run in kernel and can respond to client calls directly or forward them to the user-space daemon. A simple key-value store can also be used to cache some information at the kernel level.

Finally, DeFUSE [294] leverages a direct mapping of files between the user-space daemon and the underlying file system to minimize the number of context switches when answering metadata related calls. Some operations are then immediately performed in kernel-space without a need to communicate with the user-space daemon.

Other notable enhancement such as crash-recovery can be added to user-space file system implementation [253] with minimal changes.

SafeFS, in contrast to these other frameworks, does not try to transparently enhance the performance of existing systems by modifying the FUSE tooling. SafeFS provides a layered architecture that allows the combination and or the re-use of existing user-space file systems in a single yet configurable solution. To port an existing implementation to SafeFS, one simply needs to add initialization and cleanup functions to their existing codebase and re-compile their code against SafeFS.

Table 2.3: Survey of privacy-aware file systems. We categorize them by type (*Kernel*=kernel-space, *User*=user-space), if they are stackable and if the stacking architecture contemplates a single-layer (Single-L) or multi-layer (Multi-L) hierarchy, if they provide a plug & play design for easily switching between different privacy-preserving libraries (PnP-Privacy), if they can leverage multiple storage backends (Multi-B), and if the multiple backend algorithms (data replication, load-balancing, etc) are pluggable (PnP-B).

Type	Name	Stackable Layers		PnP-Privacy	Multi-Backend	
		Single	Multi		Multi	PnP
Kernel	PPDD [209]	✗	✗	✗	✗	✗
	StegFS [168], EFS [177], TCFS [38], BestCrypt [125], eCryptfs [107]	✗	✗	✓	✗	✗
	Cryptfs [298]	✓	✗	✗	✗	✗
	NCryptfs [289]	✓	✗	✓	✗	✗
	dm-crypt [55]	✓	✗	✓	✓	✗
User	CFS [25], LessFS [145], MetFS [175], S3QL [229], Bluesky [278]	✗	✗	✗	✗	✗
	SCFS [22]	✗	✗	✗	✓	✗
	EncFS [82], CryFS [54]	✗	✗	✓	✗	✗
	SafeFS	✓	✓	✓	✓	✓

2.4.4 Privacy-aware file systems

Data encryption is an increasingly desirable counter-measure to protect sensitive information. This becomes crucial when the data is being stored on third-party infrastructures where data is no longer in control of its rightful owners. However, the widespread adoption of these systems is highly dependent on the performance, usability, and transparency of each approach.

As shown in Table 2.3, several proposals address data encryption to tackle the aforementioned challenges. Cryptfs [298], NCryptfs [289], StegFS [168], PPDD [209], EFS [177], and TCFS [38] reuse the implementations (kernel-code or the NFS kernel-mode client) of existing file systems to integrate a data encryption layer. These systems support distinct symmetric key cryptography algorithms such as Blowfish [237], AES [112], and DES [248].

Similarly, dm-crypt [55] is implemented as a device-mapper layer and uses Linux Crypto API [151] to support different symmetric ciphers. BestCrypt [125] provides a kernel solution to export encrypted volumes. It supports several cipher algorithms such as AES, CAST [2], and Triple DES. The kernel-space eCryptfs [79] further extends BestCrypt with the support of additional ciphers (*e.g.*, Twofish).

Another set of solutions is focused on user-space implementations. CFS [25] provides a user-level NFS server modification that uses DES+OFB to protect sensitive information. EncFS [82], lessfs [145], MetFS [175] and CryFS [54] provide FUSE-based file systems that cipher data with techniques ranging AES, Blowfish, RC4, and AES-GCM.

Proposals such as S3QL [229], Bluesky [278], and SCFS [22] focus on remote FUSE-based file systems that store data in one or multiple third-party clouds. These systems ensure data confidentiality with AES encryption.

As shown in Table 2.3, with the exception of Cryptfs, NCryptfs and dm-crypt, all the previous solutions are monolithic systems, similar to traditional distributed file system proposals [5, 251] with additional data confidentiality features. Many proposals only support one type of encryption scheme, and even in the systems that allow the user to chose among a set of schemes, it remains unclear what are the trade-offs of using different approaches in terms of system performance and usability.

Only a small number of these proposals provide a design that supports the distribution of data across different storage back-ends, for replication, security, or load balancing purposes. Also, none of these proposals enable to easily switch across the distribution algorithms

(column PnP-B) and have a stackable architecture that support stacking layers with this distribution behavior (column Multi-L). For instance, it is not possible to provide a stackable design where a replication layer replicates data across several subsequent processing layers that encrypt data replicas with different encryption algorithms before storing them.

2.5 Data reduction

Considering the constantly growing amount of data stored in the cloud, finding cost-effective ways of storing it is a necessity for cloud providers. While the previous sections of this background chapter covered ways to guarantee the confidentiality, integrity, availability and durability of data from a client's perspective, they often have done so by growing the storage overhead. While storage providers must offer good durability guarantees to their clients, they can resort to any of the schemes described in Section 2.1 across an array of machines. But assuming that these machines are failure-independent domains, an opportunity to reduce the storage cost on individual machines arises. In the rest of this section, we list three common data reduction schemes and how they are used in practice can be used in practice.

2.5.1 Compression

Data compression, or the process of reducing the amount of bits used to represent a piece of information, is one of the oldest form of data reduction for the storage and transmission of digital information. Typically limited to single files or small sets of files, compression schemes are usually divided across two categories: lossy and lossless. Lossy algorithms irreversibly remove redundancy from data by targeting information whose absence is not too damaging to the original content. Lossy compression algorithms are especially effective at storing specific types of data (*i.e.*, images, sound, video). However, loss of information, even minimal, is not tolerable in content-agnostic storage systems. For this reason, most systems resort to lossless schemes to reduce storage overhead.

Looking at surveys and studies of compression algorithms proposed over the past 60 years [144, 303, 241, 219, 301, 123], we can further break down most historical schemes in two categories: entropy-based encoding and dictionary-based encoding. Entropy-based encoding targets symbol redundancy, running a frequency analysis on the message to compress and replacing the original fixed-length symbols with shorter representations. Notable examples of entropy-encoding include Huffman coding [118] and arithmetic coding [143]. Huffman coding runs a frequency analysis on the symbols in the message and builds a binary tree assigning each symbol to a node with the most frequent symbols mapping to the shortest paths. Arithmetic coding [143] represents the entirety of the message in a precision-arbitrary decimal number shortening the representation of frequently appearing symbols in the decimal part. But similarities often extend beyond single symbols at a time. To solve this, LZ77 [308] and LZ78 [309] schemes introduce dictionary-based encoding. In effect, these algorithms propose to compress a message by substituting repeated sequences in the message with pointers to full sequences stored in a dictionary. The LZ family has birthed a plethora of implementations, sometimes combining optimized dictionary-based encoding with entropy-based encoding, to favor fast processing speed (lZ4 [157], LZO [157], lZW [283], LZJB, snappy [99], ZStandard [85]) or good compression ratio (DEFLATE [64], LZMA[158]). And while knowledge of the domain can help to improve compression performance [92, 93, 220, 115, 58], most storage systems stick to the generic algorithms listed earlier as their behaviour are more predictable. In practice, these algorithms are often bundled as libraries (zlib [3])

in software ranging from file archivers (*e.g.*, 7Zip [1], bzip2 [35], tar [258], Gzip [95]) to file systems.

Numerous file systems offer transparent compression as part of their features. NTFS [70] provides a custom implementation of LZ77. ZFS [194] originally compressed its blocks using LZJB, a ZFS specific implementation of LZ77, but has since moved on to offer the same services using LZ4 and ZStandard. Other utility file systems such as readonly compressible file system squashfs [154] can be tuned to use LZMA, LZ4, LZO, LZMA2 and DEFLATE (through gzip). Intermediary systems such as Red Hat's Virtual Disk Optimizer [62] sitting as a logical layer between the file system and the block storage provide their own compression using their internal HIOPS compression technology [49].

Ultimately, the choice of compression algorithm in a system can be seen as a trade-off between compression ratio and compression/decompression speed and a balance must be struck to match user expectation. But as we'll see in the rest of this section, the data reduction possibilities are not limited to the ones mentioned so far.

2.5.2 Delta-encoding

Compression works best on files or limited sets of files. However, in most storage systems, files tend to change over time. For systems capable of versioning, figuring out the most efficient way to store changes over time is a necessity. This is where delta coding comes in.

Delta or differential encoding is a data reduction technique typically used for compression of a single file, but potentially using earlier versions of the file to efficiently compress new versions. Essentially, delta encoding captures the changes between two pieces of information in a form that allows transitioning from one to the other interchangeably (symmetric) or only from an older version to a more recent one (directed). Delta encoding can be used to to reduce bandwidth use for existing network protocols HTTP [179] or to distribute software updates [246].

Even though delta encoding can be found in various pieces of software (*e.g.*, file comparison and patching software, text editors and version control systems, ...) the formats are often implementation specific. However, at least two software-agnostic formats emerge: GDIFF [113] and VCDIFF [136] (based off [19]). Due to its better compression results, VCDIFF has emerged as the preferred format [137]. Expressing differences in terms of `ADD`, `COPY` and `RUN`, VCDIFF is based of LZ77[308] and thus compresses information by building a dictionary. The dictionary can be empty or have information from previous versions of the file. VCDIFF has high granularity: it can operate on repeated chunks ranging from a few bytes to hundreds or thousands of bytes without a specific alignment. Implementations of VCDIFF can be found as libraries or tools such as xdelta [126, 127], an open source solution designed for file compression.

But despite the good compression ratio and adaptability to file size, delta encoding is not suitable to handle a great number of files at reasonable throughput. To this end, we look at deduplication to enable data reduction at scale.

2.5.3 Deduplication

Compression and delta-encoding manage to get good compression ratios at the expense of intensive and deeply involved data processing at the symbol or byte-level. However, for most live systems, intensive data processing is likely to become a bottleneck. To solve this issue, data deduplication trades high compression ratios in small file sets for faster processing

speeds and slower convergence towards optimal storage usage by shifting the scale to the entire storage system and breaking down the process in a series of steps.

Data deduplication [163] is a data reduction technique that (i) splits incoming data into chunks, (ii) identifies duplicate chunks to decide whether to store the new ones or reference the old ones, (iii) registers this information in a registry and finally (iv) stores the deduplicated chunks. While composed of seemingly simple steps, deduplication proposals cover a wide variety of implementation choices [306, 46, 214, 199, 162, 130, 292, 243] made to work under a variety of workloads [242, 176]. Given its potential, it has been used in production in a number of applications, *e.g.*, archival storage systems [215], as part of SSDs [39], file systems such as ZFS [302] and Ceph [282], within distributed storage networks [80], or to store virtual machine images over public clouds [182]. Recent studies have also analysed its compression potential from an information theoretic perspective [184, 275]. Let us now see how systems can differ in the way they implement data deduplication.

From the ingestion of data by the system and its splitting (i), a choice of fixed or variably-sized chunks can already influence the compression gains. Fixed-size chunking works best for systems where data is processed in blocks of regular sizes and deduplication is performed in-band (*i.e.*, data is deduplicated in the writing path). An example is ZFS that can split data in fixed-size chunks ranging from 4 KB to 128 KB. However, fixed-size splitting might miss identical patterns in chunks when they reside at different offsets within the chunks. To solve this issue, variably-sized chunking uses a rolling hashing algorithm, such as Rabin fingerprinting [217], to detect identical patterns regardless of their position in the incoming chunks. To limit length variance, chunks are still bounded by a minimum and maximum value [176].

For systems generating fixed-size chunks, the deduplication pipeline can still vary in the way they identify chunks and detect duplicates (ii). Systems where metadata can be leveraged to detect duplicates, such as databases, can perform content-aware deduplication. Content-agnostic systems, on the other hand, must resort to low-collision hashing algorithms, most often defaulting to cryptographic hashing algorithms, to detect duplicates. For each chunk coming into the system, the hash value is used as the fingerprint and is matched against a registry of existing chunks. As an example, ZFS uses SHA-256 [192] to generate fingerprints of 32 B for chunks of 128 KB (default value) keeping the fingerprint size reasonably smaller than the original chunks. Opendedupe [29] features both a non-cryptographic function in MurMurHash3 [9] and cryptographic functions MD5 [225] and SHA-256. With deprecated schemes such as MD5 or SHA-1, and even the selection of a SHA-256 successor in SHA-3 [76], one could wonder why deduplication systems would resort to antiquated hashing schemes. What's more, non-cryptographic schemes like Murmurhash3, outperform cryptographic hash functions speed-wise but are not necessarily built to resist second preimage attacks where specifically crafted input maps to a specific fingerprint [26]. Indeed, cryptographic hashing algorithms are chosen because of their low-collision probability and predictable throughput despite their use being deprecated for security purposes.

Coming to step (iii), once duplicates have been detected, the decision to store new chunks or simply reference older ones implies the active maintenance of a registry. Depending on whether data deduplication is performed in-band, *i.e.*, as data is ingested, or out-of-band, *i.e.*, data has already been stored, we may need to balance fast duplicate look-up time and efficient storage of the registry. A naive approach would have implementers put the entire registry in memory. However, with the size of the registry bound to grow as new data comes in, keeping everything in memory is not practical for high-capacity storage systems. Memory-savvy solutions such as extreme binning [23] proposes a technique to reduce the

amount of RAM required for identifying the file chunks and performing deduplication only within the context of a given *bin* containing data chunks. Each file would have a single fingerprint for its *representative chunk* in memory and pointing to a specific bin, which would contain similar data. The representative chunk was selected based on the minimum fingerprint of all chunks in the file. Broder's theorem [30] shows that this particular fingerprint will correspond to files with high similarity. Although the technique is sub-optimal from a global deduplication perspective, *i.e.*, deduplication happens only within each bin, its performance degradation in backup systems was shown to be moderate compared to deduplication across all data.

Finally, when duplicates have been detected and registered as part of the metadata, comes the final storage step (iv). Deduplicated chunks can be stored as they are, but it is not uncommon for storage systems to compress them further. It is the case of ZFS but also Red Hat's Virtual Disk Optimizer [62] that can deduplicate and compress using their HIOPS technology [49]. This method has the benefit of greatly reducing the storage overhead and is thus ideal for archival storage. However, compression after deduplication also suffers, to a lesser extent, from the loss of direct access to data (as mentioned in Section 2.5.1).

To summarize, data deduplication is a pipelined technique of data reduction operating, either in-band or out-of-band, at the level of the complete storage system favoring processing speed over compression ratio. It works by breaking data down into fixed or variably-sized chunks before identifying new chunks against existing chunks and ensuring that a single copy of identical chunks is stored. Finally, data deduplication can be combined with other data reduction technique such as compression to further reduce storage overhead.

Chapter 3

On the Cost of Reliable and Secure Storage on Public Clouds

With the advent of smaller and more powerful mobile devices, the last decade has seen a paradigm shift in the way private individuals and companies store and share their data. Be it in the workplace, at home or on the move, data is expected to flow seamlessly from one device to the other. At their core, these new sharing patterns and their expectations, are largely supported by public cloud storage. Drawn by the wide range of services they provide, no upfront costs and 24/7 availability across all personal devices, customers are well-aware of the benefits that these solutions can bring.

However, complete trust in these services should not be granted without considering some of their flaws in terms of privacy and dependability. Data entrusted to these providers can be leaked by hackers, disclosed, censored or removed upon request from a governmental agency, or even mined by the provider without the explicit consent of the user. While there exist solutions to prevent or alleviate these problems, they typically require direct intervention from the clients, like encrypting their data before storing it, and reduce the benefits provided such as instant and transparent uploads. This chapter studies a wide range of security mechanisms that can be used atop standard cloud-based storage services. We present the details of our evaluation testbed and discuss the design choices that have driven its implementation. We evaluate several state-of-the-art techniques with varying security guarantees responding to user-assigned security and privacy criteria. Our results reveal the various trade-offs of the different techniques by means of representative workloads on top of industry-grade storage services.

3.1 Introduction

Public online cloud-based storage services such as Dropbox [72], Google Drive [98] and Microsoft OneDrive [190] are nowadays the de facto standard for users to store their photos, music and other types of documents online. The extremely low economic barrier of these services (which typically offer free basic accounts), their ubiquitous availability, as well as their ease of use with transparent client integration contribute to making them an attractive solution for both individuals and organizations [12].

Cloud-based storage services are also largely exploited by application developers. They typically expose cross-platform REST-based APIs that can be seamlessly plugged into existing systems. Developers therefore use these services to add online storage backends to their applications without having to face the costs and burdens of managing their own storage

infrastructure. Most online applications developed nowadays follow this architectural pattern (*e.g.*, online word processors, mobile applications, etc.).

Nevertheless, as soon as the data enters the cloud provider's service perimeter, the client essentially surrenders control over it [304], which is highly undesirable. In fact, the control over personal data is among the major concerns of individuals and organizations. A recent report [84] carried with European citizens shows that 67% of the population is concerned by the information they disclosed online (voluntarily or not), and only 15% think to be in control of their own data. As a consequence, concerns over the disclosure of private information by malicious insiders [75] and data breaches [231] have motivated a new class of secure and safe cloud-based storage applications and services. This trend is further amplified by the lack of security expertise from software developers [305].

To protect the privacy of the users and their data, researchers proposed several systems [21, 257, 305] that encrypt data at the client side before sending it to the cloud providers. These systems offer various security guarantees to the end-users (*e.g.*, integrity, authorization, privacy) and typically follow two different deployment strategies: single- or multi-cloud modes. The former stores data on a single storage provider, while the latter spreads it across multiple providers, possibly operating under distinct (non-colluding) administrative domains. Partitioning data across multiple storage providers ensures that, even if one of them is compromised, the attacker cannot access the complete original information. In fact, depending on the multi-cloud partitioning algorithm, it is possible to guarantee that no information from the original data is leaked as long as one of the storage providers remains secure [63].

Current systems suffer from a major drawback: they either provide very specific yet incomplete security mechanisms (*e.g.*, some only provide data integrity, others provide only data privacy [147]), or they integrate general-purpose security measures that cannot be tailored for a given application (*e.g.*, some systems bundle confidentiality, integrity, and access-control in a single package [21]). Neither approach allows further customization based on the user's security requirements, (*e.g.*, choosing among multiple security features with different guarantees concerning data confidentiality, anti-censorship and fault tolerance).

We strongly believe that it is essential to understand the impact of each security measure adopted by cloud storage systems in terms of resource consumption (*e.g.*, computing power, storage space, network throughput), economic impact, and overall performance (latency, ease of use, services offered). For example, increasing the size of an asymmetric encryption key to provide stronger security has non-negligible impact on the system's energy requirements, a crucial metric in today's mobile application market. Strong security measures can also render services unacceptably slow, and even disable them. The ability to take an informed decision on these design compromises is of paramount importance for the deployment of storage systems on public clouds [276].

In this context, our contributions are threefold. First, we define a set of basic security guarantees that can be combined and implemented by a client to securely store content in the cloud. Second, we design and implement a modular software architecture that can operate in single-cloud or multi-cloud mode, interfacing with well-known public storage clouds as well as on-premise private data stores. Third, we evaluate the different security features using a set of well-defined workloads. Our evaluation unveils the costs of each feature in terms of resource usage, storage space and latency. The rest of this chapter is organized as follows. We start with a look at the security guarantees that we expect from cloud storage solutions in Section 3.2. We then present different deployment scenarios leveraging one or multiple storage providers in Section 3.3. Our testbed's architecture is described in Section 3.4 and

the implementation details are further explained in Section 3.5. We detail the configurations put under test in Section 3.6 and present our evaluation in Section 3.7. We then look at the overall results with some perspective in Section 3.8 and present our conclusion in Section 3.9.

3.2 Security and availability requirements

The term *security* encompasses a rich set of concepts, and the definition of security itself usually varies according to context. We consider three fundamental security guarantees: *confidentiality*, *self-integrity* and *origin authentication*. But security alone is not enough. In order to build a complete solution that leverages multiple cloud storage providers, we must also consider how to maintain the *availability* of the system. That is that the system remains functional and continues serving client requests even in the case of one or multiple failures from the cloud providers. Availability can be maintained by combining self-integrity and redundancy.

Confidentiality. Confidentiality is a fundamental guarantee offered by storage systems. It ensures that stored data cannot be disclosed to third-party entities without the permission of its rightful owner. This guarantee is achieved by resorting to encryption schemes, as further discussed in Section 3.3. The security of cryptosystems used for data confidentiality can be formally quantified in different ways, such as information-theoretic and semantic security. Confidentiality is a primordial feature that we systematically include in the different security configurations explored in our evaluation in Section 3.7.

Self-Integrity. Self-integrity protects data against unauthorized or unintended undetectable data modifications caused by attackers or by data corruption. Self-integrity is typically achieved with secure hash functions: when fetching a requested stored block B, the system must either return the same block B or an error; it cannot return another data block $B' \neq B$. In practice, we can achieve this by having the key computed as a cryptographically secure hash function of B, and have the client recompute the hash from the data and check the matching key after every read operation. A malicious server must be able to break the hash function in order to break this self-integrity. Self-integrity requires that metadata cannot be tampered. But in practice, metadata is small enough to be kept at the client-side out of reach from the attacker.

Origin authentication. Data origin authentication is a particular instance of message authentication that allows a storage provider receiving data to assess and verify the rightful owner of the data. Like message authentication in general, data origin authentication can be implemented using digital certificates like signatures and message authenticated codes. The server can verify that certified data comes from a party in possession of the corresponding private key.

Redundancy. Redundancy is a corner stone of availability ensuring that the service remains available in case of data loss or corruption. For a storage system, redundancy is typically achieved by means of replication or erasure coding. In case of data loss, copies can be fetched to keep on serving client requests. By distributing the copies on failure independent storage nodes, the additional storage overhead can then be balanced across providers.

3.3 Deployment scenarios

We consider two distinct deployment scenarios. In the first, users' data is stored in a single cloud storage provider, while in the second data is stored across multiple storage providers.

These two distinct environments imply different trust models and security mechanisms. In this section, we revise their characteristics and discuss the security techniques that best fit each of them for obtaining different security guarantees. These deployment scenarios and mechanisms are then evaluated in Section 3.7.

3.3.1 Single-cloud deployment

The single-cloud scenario is representative of the way cloud storage services are typically used. Namely, users interact with a single cloud provider to store and fetch their data.

Trust model. In the single-cloud deployment scenario, an adversary has access to users' data as soon as it has access to the provider's premises (*e.g.*, it has hands-on access to the hardware hosting the persistent storage devices). The adversary can be internal to the cloud provider (*e.g.*, the cloud provider itself for commercial benefits) or an external unauthorized user (*e.g.*, hacker). The adversary can arbitrarily manipulate users' data. The cloud provider is therefore considered as untrusted. On the other hand, the client machine are assumed to be secure and network communications between client and cloud provider are done over secure channels (*e.g.*, TLS).

Security mechanisms. Security mechanisms must be applied at the client-side, as the cloud storage is untrusted. This entails encrypting data before uploading it to the cloud. There are several encryption mechanisms available and, from a high level perspective, these can be separated in two groups: those based on computationally hard problems and those based on the theory of information [63]. For the single cloud deployment we will only consider those based on computationally hard problems. Symmetric encryption guarantees data confidentiality. We resort to the AES block cipher with a 128-bit key size in CBC (Cipher Block Chaining) mode, which is a commonly used setup [90, 78]. AES is nowadays considered secure and industrial providers are currently phasing out its predecessor DES [230]. In addition, using AES in CBC mode has been shown to perform well for read heavy workloads [50]. Asymmetric techniques can also be used to achieve data confidentiality. They typically require extra computational power, when compared with the symmetric approaches, especially when dealing with large files [21]. For this reason, we just use them to sign the users' data and to verify its authenticity. Multiple signature schemes are available. The most relevant ones are based on RSA or DSA, but we only consider RSA as it is faster on signature verification and typically users perform more verifications than signatures [81, 287].

Finally, cryptographic hash functions can be used to generate data digests and to support integrity guarantees. MD5 [225] and SHA [77] digests are commonly used cryptographic hash functions. In our evaluation, we use SHA-512 message digests as they are commonly used in public software package repositories. SHA is also used with RSA to generate the digital signatures.

Fault tolerance. In a single-cloud scenario, as soon as the cloud-storage provider is unreachable, users lose access to their data. Moreover, if the service fails and data is lost permanently, the corresponding users' data cannot be recovered.

3.3.2 Multi-cloud deployment

In a multi-cloud deployment context, data is stored across multiple storage providers. This scenario brings benefits on several points: performance, storage capacity, data availability and security [305, 21, 67, 257, 171]. Under this scenario, we must guarantee that a single corrupted cloud provider does not lead to a full disclosure of the stored data.

Trust model. We assume that for n providers, the adversary has access to the data of at most $n-1$ storage providers and that it must be possible tolerate up to $n-1$ corrupted providers. Adversaries have the same computational powers as the single-cloud deployment scenario, and the client as well as client-to-cloud communications are secure.

Security mechanisms. Confidentiality in a multi-cloud deployment can be guaranteed by exploiting the same encryption mechanisms used in a single-cloud context. The multiple clouds can be leveraged to improve data availability and system performance.

Indeed, data can now be of split and distributed data across independent cloud providers. To leverage this, we consider two coding and distribution schemes: erasure coding (briefly described in Section 2.1) and XOR encryption [134]. For the first scheme, the usual workflow consists in encrypting the data using a symmetric cipher and then erasure coding the data to add redundancy and split it in parts that are distributed across the cloud providers. Erasure coding allows maintaining data availability even when several cloud providers become suddenly unreachable, while consuming significantly less storage space than data replication [281]. Moreover, erasure coding reduces by a fraction the costs of data migration from one cloud provider to another when compared to replication [305]. For the second scheme, we evaluate the performances of a one-time pad XOR [196] as alternative to erasure coding. In essence, XOR encryption takes a piece of data and encrypts it with a number of random blocks. The encrypted blobs and the random blocks are then ready for storage. This process ensures that no stored part leaks content from the original document. However, unlike with erasure coding, all parts must be available to successfully decode the original data. Therefore, to maintain data availability, each part must be replicated and thus adds to the storage overhead.

For the sake of completeness, we mention but do not implement in our system secret sharing [240]. Similar to erasure coding algorithms such as Reed-Solomon [222], Shamir's secret sharing algorithm encodes the data in n parts; it is then possible to reconstruct the original value from a subset of k parts (where $k < n$). Secret sharing schemes such as Shamir's seem ideal to blend redundancy and encryption. However, information secure secret sharing is impractical in terms of network bandwidth and storage overhead [139]. While more efficient alternatives exist [139, 67], we consider erasure coding and the one-time pad sufficiently distant extremes on the information security spectrum for our experimental evaluation.

Finally, similarly to the single-cloud deployment, SHA-512 and RSA-based signatures are used for providing self-integrity and origin authentication in a multi-cloud scenario.

Fault tolerance. When multiple cloud providers are available different failure scenarios can be considered depending on the security measures being used. In particular, when deployed without any replication, the adoption of the one-time pad does not offer any fault-tolerance guarantees, *i.e.*, if one of the providers becomes unavailable, it is impossible to recover the original data. Conversely, it is possible to support multiple storage faults using erasure coding techniques at the cost of increased storage overheads.

3.4 Architecture

The Playcloud architecture comprises the following components: a proxy that mediates interactions between clients and the Playcloud system, an encoder component, and a set of backend storage clouds (public clouds or private servers deployed on-premises). The architecture is depicted in Figure 3.1.

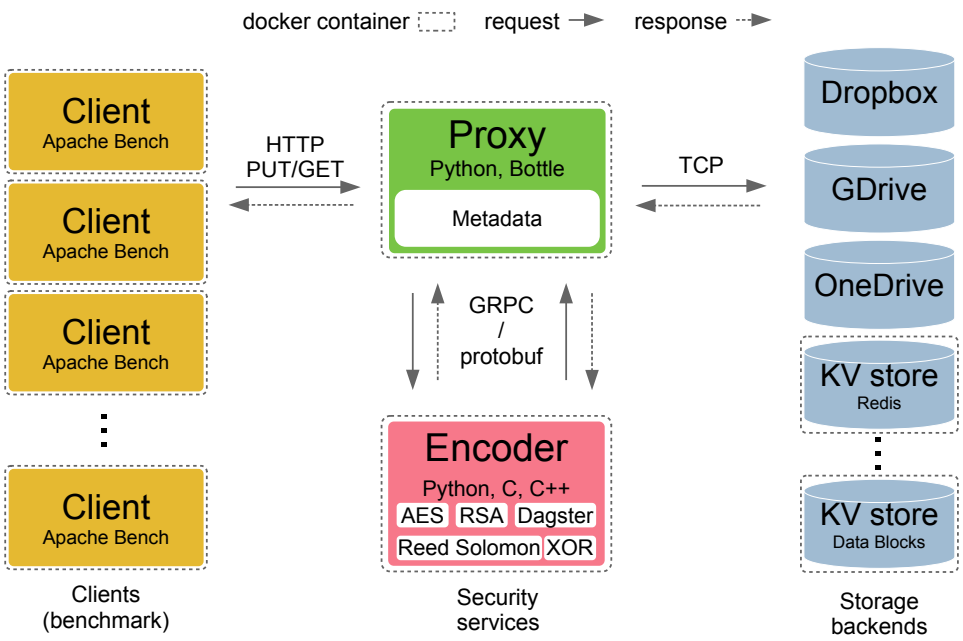


Figure 3.1: Architecture of Playcloud, our experimental testbed.

The proxy component acts as Playcloud's front-end and is responsible for keeping a mapping between client files and the actual storage backends where they are stored.[1] Clients, which run on independent nodes, contact the proxy component to write or read data through a simple REST interface that mimics the operating principles of well-established services like Amazon S3. The interactions between the proxy and the clients happen via synchronous HTTP messages over pre-established TCP channels.

The Playcloud system is configurable and different security mechanisms can be put in place. According to such configuration, the proxy component coordinates the other components in the system and different workflows may arise. For instance, some configurations require a single cloud backend while others need two or more.

Upon a write request, the proxy component asks the encoder to encode data blocks according to the configured security mechanisms. The resulting block or blocks are then dispatched, by the proxy, to the storage backends. To this end, the proxy maintains a data block index to keep track of where data is stored at the backends.

Upon a read request for a piece of data, the proxy checks the block index to figure out where the corresponding encoded blocks are stored. It fetches them from the backend storage and forwards them to the encoder that decodes the blocks before returning the data to the client.

The encoder is co-localized with the proxy to maximize throughput and avoid bottlenecks induced by high pressure on the network stack. To increase the flexibility of our Playcloud, our encoder provides a plugin mechanism to dynamically load and swap different coding and cryptographic libraries and associated bindings. This mechanism relies on a platform-independent transport format (using *protocol buffers*) and a stable interface between the proxy and the encoder. Security is ensured if both the proxy and encoder are deployed on a trusted domain. Typically, this domain can correspond to the client premises since the computational resources required are expected to be manageable even by handheld devices.

[1]Several proxy instances can co-exist if consistency is guaranteed across the various local mappings. However, we consider this extension was considered out of the scope at the time of this work.

3.5 Implementation details

Our implementation choices have been largely driven by performance and programming simplicity considerations, as well as by constraints from the storage backends interfaces.

The proxy component is implemented in Python (v2.7.10) and exploits the exporting facilities of the Bottle [27] framework (v0.12.9). The proxy handles PUT and GET requests via the WSGI [290] Web framework.

The encoder, also written in Python, integrates with various encoding libraries. Each library is wrapped exposing the same API to the encoder allowing the system to be expanded and to abstract Playcloud from the implementation details of each library. This allows Playcloud to support not only Python libraries but also native ones.

We leverage Cryptography [56], a python library that exposes a wide range of cryptographic primitives with an easy to use and well documented API. Namely, this library provides the AES and RSA ciphers by wrapping OpenSSL's cryptographic protocol implementations [88]. We use our own implementation for the one-time pad XOR encoding driver that resorts to the numpy [185] library to optimize vector computation.

For erasure coding, Playcloud leverages the libraries made available by liberasurecode [191] (implemented in C/C++) and its Python wrapper PyEClib [212] (v1.2).

For the client side, we built a suite of micro- and macro-benchmarks, leveraging Apache Bench [262] (v2.3), to measure the throughput and latency of client storage requests. The CPU and memory measurements presented in the evaluation are gathered with the `dstat` tool [74].

Finally, we have implemented drivers for four storage backends. First, we deployed a set of on-premises storage nodes using Redis [221] (v3.0.7), a lightweight yet efficient in-memory key-value store. Redis tools provide easy-to-use probing mechanisms (*e.g.*, the `redis-cli` command-line tool), which allowed us to measure the impact of the several security combinations used in our evaluation. Second, we have implemented drivers for the three most widely used cloud storage services: Dropbox [72], Google Drive [98], and Microsoft OneDrive [190]. The drivers are implemented leveraging the official Python SDKs from each provider. Similarly to the approach taken with the encoding component, storage backends are wrapped to expose a common interface with the required set of operations, *i.e.*, store, fetch and delete data, which allows to easily plug-in new storage backends in the future. Overall, our implementation consists of 2,723 lines of Python code, all components included.

3.6 Configurations

In this section, we detail the different configurations we choose to evaluate. We start with a list of candidate erasure coding libraries in Section 3.6.1 and then present the different combinations of security mechanisms and their associated security guarantees in Section 3.6.2.

3.6.1 Erasure coding

In 2009, Plank *et al.* [203] conducted a comprehensive performance evaluation of open-source erasure coding libraries available at the time. The focus of the study was on the raw performance of the libraries and the impact of different parameter configurations. In this first part, we take a fresh perspective at the state of practice in erasure coding for data storage.

We test four different coding libraries in our experimental evaluation. While many more coding libraries are available on the market, we focus our evaluation on widely available

Table 3.1: Summary of encoder names and libraries, support for hardware acceleration (HW), and the description of the algorithms (RS stands for Reed-Solomon).

Encoder	Library	HW	Description
`liberasure_rs_vand`	liberasurecode	✗	Vandermonde RS
`liberasure_flat_xor_3`	liberasurecode	✗	Flat-XOR ($d = 3$)
`liberasure_flat_xor_4`	liberasurecode	✗	Flat-XOR ($d = 4$)
`jerasure_rs_vand`	Jerasure	SIMD (SSSE3); CLMUL	Vandermonde RS
`jerasure_rs_cauchy`	Jerasure	SIMD (SSSE3); CLMUL	Cauchy RS
`isa_l_rs_vand`	Intel ISA-L	SIMD (SSE4); AVX(1/2)	Vandermonde RS
`longhair_cauchy_256`	LongHair	SIMD (SSSE3)	Cauchy RS

and document ones. We describe below the main features of each of them. Table 3.1 summarizes their principal characteristics. The results of this evaluation are presented in Section 3.7.

Liberasurecode. Liberasurecode[2] is an erasure code API library in C that supports pluggable erasure code backends. It supports backends such as jerasure and Intel ISA-L but also provides three erasure codes of its own: a Reed Solomon implementation and two flat XOR implementations. Flat XOR erasure codes [103] are small low-density parity-check (LDPC) codes [156]. With flat XOR codes, each parity element is the XOR of a distinct subset of data elements. Such codes are not maximum distance separable (MDS) and, hence, incur in some additional storage overhead over MDS codes. However, they offer the advantage of additional recovery possibilities, *i.e.*, an element can be recovered using many distinct sets of elements. We evaluate two flat XOR codes constructions, `flat_xor_3` and `flat_xor_4`, that respectively have a Hamming distance of $d = 3$ and $d = 4$.

Jerasure. The Jerasure library,[3] first released in 2007, is one of the oldest and most popular erasure coding library. Jerasure is written in C/C++ and implements several variants of Reed-Solomon and MDS erasure codes (Vandermonde, Cauchy [155], Blaum-Roth, RAID-6 Liberation [202],...). As it has been used in many different projects, Jerasure is also a stable and mature library. It notably provides a rich and well documented API, and has been optimized for speed on modern processors (*e.g.*, by leveraging SIMD instructions since version 2.0). More details about the internals of this library can be found at [204].

Intel ISA-L. Intel Intelligent Storage Acceleration Library (ISA-L)[4] is an implementation of erasure codes optimized for speed on Intel processors [206]. It is written primarily in hand-coded assembler and aggressively optimizes the matrix multiplication operation, the most expensive step of encoding. During the decoding operations, Intel ISA uses a cubic cost Gaussian elimination solver. For our evaluation we the latest version (v2.14).

LongHair. The LongHair library[5] is an implementation of fast Cauchy Reed-Solomon erasure codes in C [205]. It was designed to be portable and extremely fast, and it provides an API flexible enough for file transfer where the blocks arrive out of order.

[2] `https://bitbucket.org/tsg-/liberasurecode`
[3] `http://jerasure.org/jerasure/jerasure`
[4] `https://software.intel.com/en-us/storage/ISA-L`
[5] `https://github.com/catid/longhair`

Table 3.2: Deployment targets, drivers and security guarantees.

Deployment	Drivers	Guarantees		
		Conf.	Int.	Sign.
Single-Cloud	`aes`	✓	✗	✗
	`aes_sha_512`	✓	✓	✗
	`aes_rsa`	✓	✗	✓
	`aes_rsa_sha_512`	✓	✓	✓
Multi-Cloud	`cauchy`	✓	✗	✗
	`cauchy_sha_512`	✓	✓	✗
	`cauchy_rsa`	✓	✗	✓
	`cauchy_rsa_sha_512`	✓	✓	✓
	`xor`	✓	✗	✗
	`xor_sha_512`	✓	✓	✗
	`xor_rsa`	✓	✗	✓
	`xor_rsa_sha_512`	✓	✓	✓

3.6.2 Security configurations

We consider 4 different encoder combinations in 3 distinct setups (Table 3.2). All configurations are encrypted, but they vary in whether they include hashing for integrity (`sha_512`) and/or signature for origin authentication (`rsa`). Additionally, for multi-cloud setups, we leverage the distributed setting by including erasure coding (`cauchy`) or a one time-pad (`xor`). Coding-wise, `cauchy` is a combination of AES-128 in CBC mode and a (14, 10) Reed-Solomon using `jerasure_rs_cauchy` (see Section 3.6.1). `xor` on the other hand, generates 2 random blocks and XOR's them with the data. In terms of storage overhead, `cauchy` produces 14 blocks for a total of 1.4 times the size of the original data while `xor` produces 3 blocks for a total of 3 times the size of the original data. In both cases, the original data can be re-read with a subset of the blocks (10 for `cauchy` and 2 for `xor`).

Table 3.3: Expected storage overhead for a file of size b when using `cauchy` and `xor`.

Driver	Growth	b		
		4 MB	16 MB	64 MB
`cauchy`	$(0.10 * b) * 14$	6 MB	23 MB	94 MB
`xor`	$b * 3$	12 MB	50 MB	201 MB

3.7 Evaluation

This section presents our evaluation study of the different security guarantees. We start by describing the evaluation settings and related contextual information (Section 3.7.1). Then, we organize the remainder of the section in three sets of experiments.

First, we benchmark different erasure coding libraries in isolation looking at their throughput (Section 3.7.2) and storage overhead (Section 3.7.3). Second, we evaluate the different architecture components in isolation (Section 3.7.4). Finally, the macro-benchmarks stress the system as a whole along different axes and workloads: the latency in a single-cloud setting (Section 3.7.5), the latency in a multi-cloud setting (Section 3.7.6), the resource usage (Section 3.7.7), and the storage requirements (Section 3.7.8).

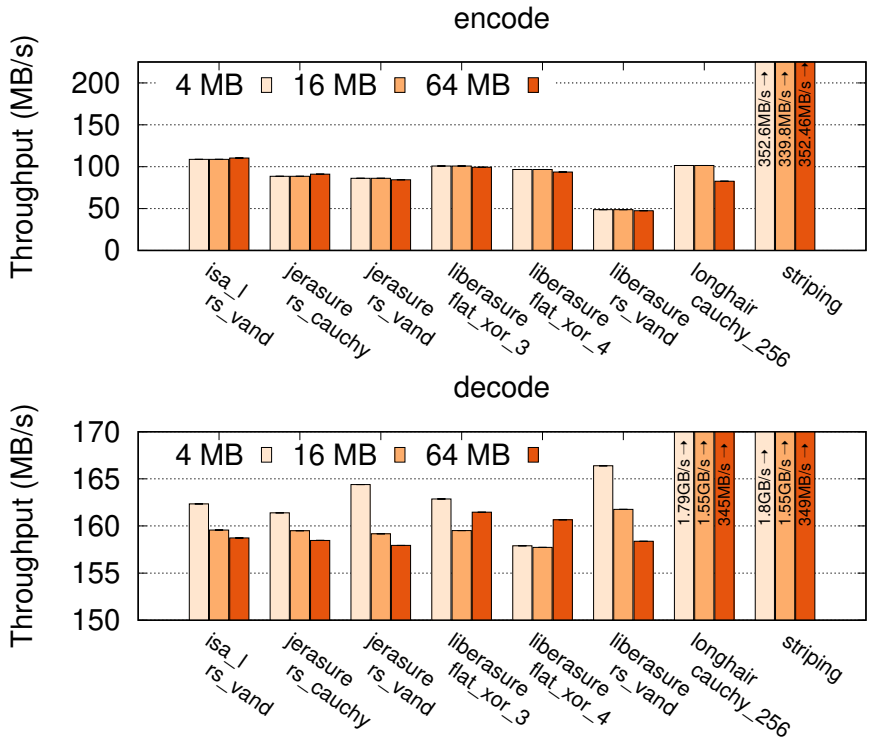


Figure 3.2: Micro-benchmark: `encode` (top) and `decode` (bottom) throughput for several coding libraries and block sizes.

3.7.1 Evaluation settings

We deploy our experiments over a cluster of machines interconnected by a 1 Gb/s switched network. Each physical host features 8-Core Xeon CPUs and 8 GB of RAM. We deploy virtual machines (VM) on top of the hosts. The KVM hypervisor, which controls the execution of the VM, is configured to expose the physical CPU to the guest VM and Docker containers by mean of the `host-passthrough` [266] option, to allow the encoders to exploit special CPU instructions. The VMs leverage the `virtio` module for better I/O performances.

We deploy Docker (v0.10) containers on each VM (1 container per VM) without any memory restriction to minimize interference due to co-location and maximize performance. In particular the proxy, the encoder and the Redis storage nodes reside in isolated containers, each of them running in VMs executed by separated hosts. Similarly, the client that injects requests into the testbed runs in an Docker container running in a separate host. We use regular accounts for the selected cloud providers (Dropbox, GDrive, and OneDrive).

3.7.2 Micro-benchmark — Erasure coding throughput

Our first set of experiments evaluate the throughput of the coding libraries for increasing block sizes of 4 MB, 16 MB and 64 MB. In this scenario, the libraries are tested in isolation via specialized clients that send a continuous stream of data blocks to encode or decode.

For each library we execute 10,000 times the `encode` function and show the average and standard deviation results. All the Reed-Solomon libraries are configured with $k = 10$ and $m = 4$, a typical configuration used in modern data centers (*e.g.*, at Facebook [233]). To approach a similar configuration, the Flat XOR libraries are set to $k = 10$ and $m = 5$. For

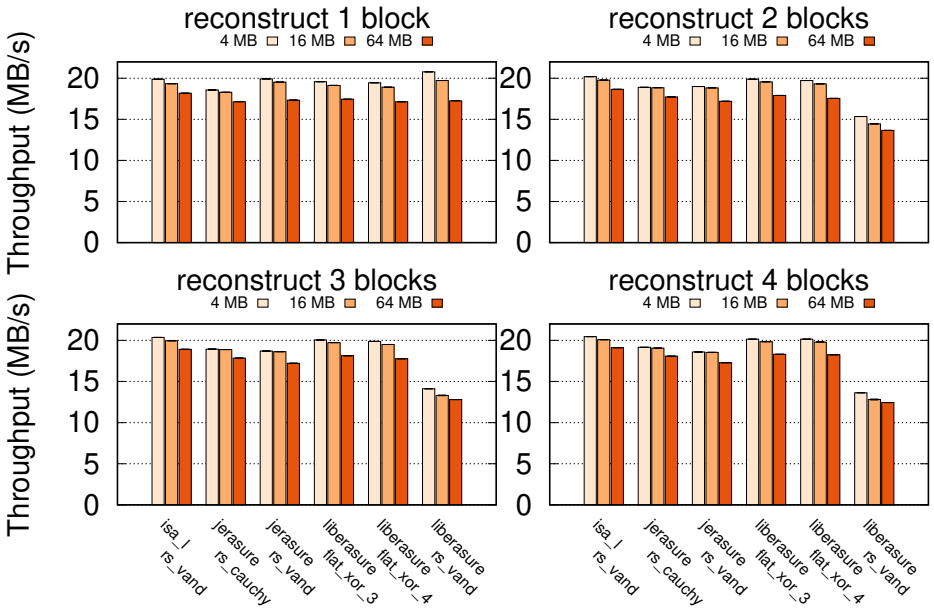


Figure 3.3: Micro-benchmark: Erasure codes - reconstruction throughput for an increasing number of missing blocks and block sizes.

reference purposes, we compare against a baseline `striping` encoder/decoder that simply splits the data in the requested number of blocks (typically one block per stripe) and immediately returns them to the client without any further processing.

Figure 3.2 presents our results for encoding (top) and decoding (bottom). We notice that `liberasure_rs_vand` is the slowest in the encoding phase, achieving at most 52.35 MB/s for a 4 MB block size. The Jerasure implementation of the same coding technique (`jerasure_rs_vand`) and Intel's `isa_l_rs_vand` perform *twice* as fast for the same block size, respectively up to 87 MB/s and 107 MB/s.

We can explain the performance gap between different implementations of the same coding techniques by two main reasons: 1) the longer foray of such libraries in the open-source community (the original design of Jerasure dates back to 2007) thus benefiting from several contributions and code scrutiny, and 2) native support for hardware acceleration for the Intel ISA and Jerasure libraries.

Finally, `longhair_cauchy_256` outperforms the other implementations for any block size. Indeed, not only is its implementation based on the Jerasure source code, but it embeds carefully hand-crafted low-level optimizations (*e.g.*, selection of the minimal Cauchy matrix, faster matrix bootstrap, etc.).

In the decoding scenario, the `decode` function is fed with all the available blocks. As expected, when all the blocks are available, the libraries can decode very efficiently, achieving throughputs that are never below 157 MB/s for any block size. For example, `liberasure_flat_xor_4` achieves a 158.13 MB/s throughput with 4 MB blocks, and `jerasure_rs_cauchy` reaches 164.87 MB/s. The highly optimized `longhair_cauchy_256` achieves results that are orders-of-magnitude better also in decoding (up to 1.79 GB/s for 4 MB blocks).

Finally, Figure 3.3 shows the cost of reconstructing missing blocks. We present the achieved throughput of the coding libraries in reconstructing from 1 to 4 missing blocks (from top-left to bottom-right). Figure 3.3 presents the average throughput for 100 executions.Notice how `liberasure_rs_vand` achieves the best result (20.77 MB/s) in reconstructing 1 missing block with 4 MB block sizes but steadily decreases with bigger block sizes and more to

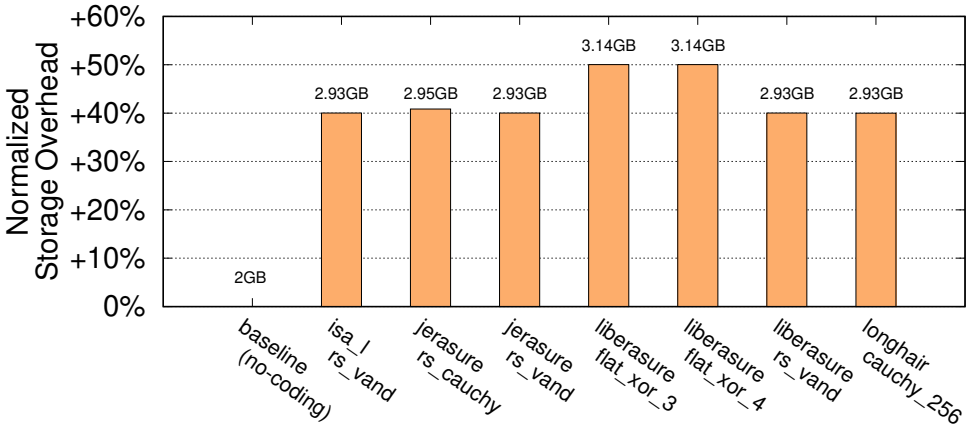


Figure 3.4: Micro-benchmark: Erasure codes - storage overhead. Storing 500 files of 4 MB.

reconstruct. This result confirms the measures of the same library in pure decoding shown previously in Figure 3.2. The other libraries perform consistently across the spectrum of parameters, and all operate between 17.15 MB/s and 19.19 MB/s. These results need to be taken carefully into account to decide which is the best fitting library to adopt in a cloud setting.

We performed a breakdown analysis of the computing times for each of the microbenchmarks. The goal of this analysis is to verify that the cost of using a high-level language such as Python did not hinder our results and thus negatively impacted on the observed performances. We exploit the `cProfile` module[6] to profile the execution of the `encode`, `decode`, and `reconstruct` microbenchmarks, and to gather profiling statistics. Indeed, the CPU spends almost the totality of the execution time (always more than 99%) in the native code of the encoding libraries. These results confirm the choice of Python as having near-zero impact on the overall performances, while providing major benefits in ease of programming, deployment, and availability of open-source libraries.

3.7.3 Micro-benchmark — Erasure coding storage overhead

We now take a look at the storage overhead induced by the choice of library and configuration. Figure 3.4 presents our results. In this experiment, the client sequentially stores 500 files of 4 MB each, for a total of 2 GB of data. The baseline results indicate the cost of storing the files without any form of coding. On the y-axis we show the storage overhead normalized against the baseline cost, while for each library we indicate the total space requirements. In our experiments, the Flat XOR erasure codes are on average 8% more demanding than the other codes: they require a total of 3.14 GB of storage space (corresponding to a +63% of the original data).

3.7.4 Micro-benchmark — Secure drivers throughput

Having first experimented with the erasure coding libraries in this evaluation, we now take a look at the security configurations presented in Section 3.6.2. Please note that in light of its portability, consistency and adequate performance we pick `jerasure_rs_cauchy` as our erasure coding library for the remainder of this evaluation. For the sake of brevity, every mention of this library will be shortened to `cauchy`.

[6] `https://docs.python.org/2/library/profile.html`

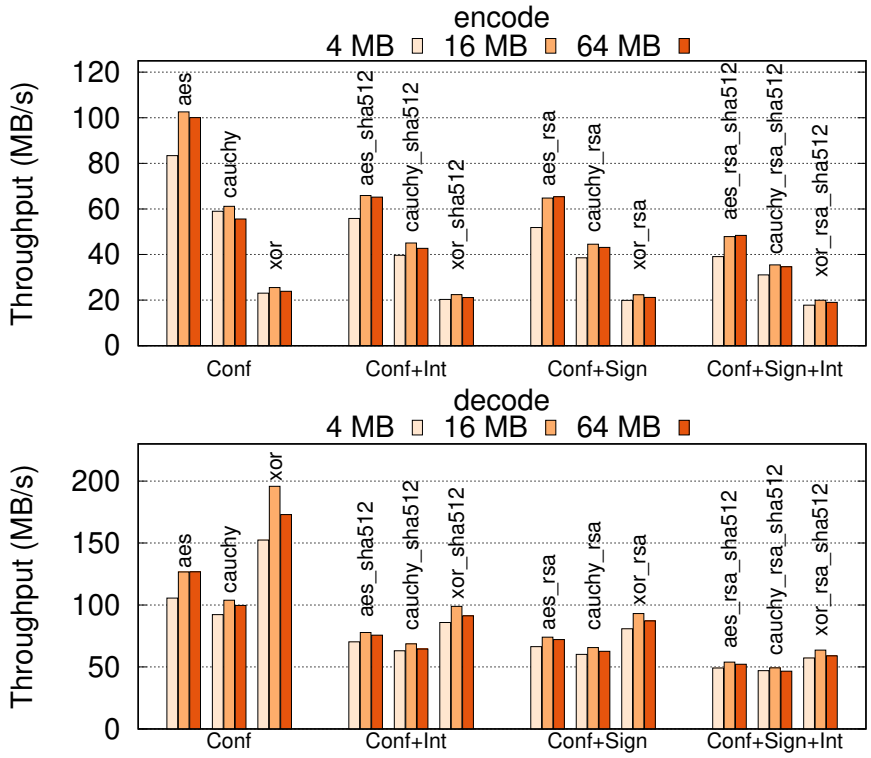


Figure 3.5: Micro-benchmark: Secure drivers - encoding and decoding throughput.

We encode and decode randomly generated binary blocks of increasing size (4 MB, 16 MB, and 64 MB). Encoding and decoding throughput is presented in Figure 3.5 (top) and Figure 3.5 (bottom) respectively. For each configuration, we present the average results observed for encoding/decoding 50 blocks. As expected, the best encoding throughput is achieved with symmetric encryption (*i.e.*, `aes` driver with 102.5 MB/s) since it avoids manipulating multiple data blocks. In fact, the encoding mechanism of the `xor` driver requires generating new random data blocks, a time-consuming operation leading to higher overhead. On the other hand, the decoding operation is reduced to the `xor` operation itself, which is very efficient. Consequently, the `xor` is significantly more efficient in decoding, and it consistently achieves the best decoding performance across the full block size range, and up to 195.7 MB/s for the 16 MB case.

As expected, combining several security options impacts negatively over the throughput. The most secure combination (*Conf.+Int.+Sign.*) consistently performs poorly compared to the other combinations, with throughput slowdown in the order of 2× for encoding and 5× for decoding.

3.7.5 Macro-benchmark — Single-cloud latency

In the rest of this evaluation, we present an extensive set of macro benchmarks, where the full stack of the system is under test. First, we present the observed latency performance of the system. We configure the testbed to operate in single-cloud mode using Dropbox, OneDrive, and GDrive as cloud backends. We include the same results executed against a local Redis storage backend deployed locally on our cluster. For each scenario, we store 250

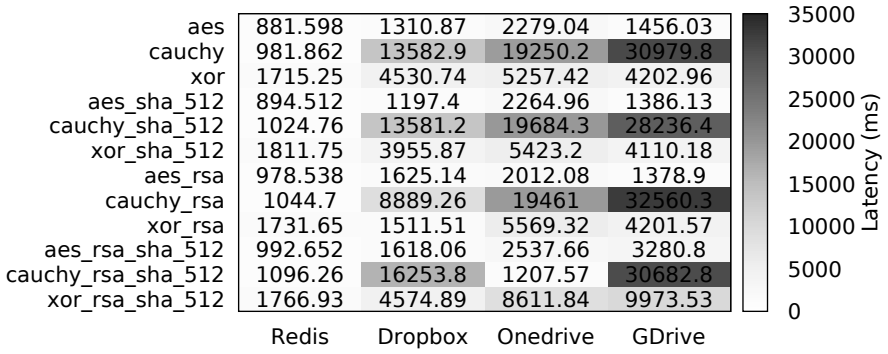


Figure 3.6: Macro-benchmark: Single cloud - average latency per block. Each cell label indicates the average latency for the given pair of driver/cloud provider.

blocks of 1 MB. We measure the insertion latency for each block. Figure 3.6 presents our results. We use a heatmap representation with shades of gray to show at once the observed latencies across the 48 distinct configurations. Each cell of the heatmap shows the average value of the measured latencies for the corresponding configuration.
Three factors influence the reponse time of storage requests:

1. **security measure complexity:** raising the computing time as the number of security measures combined grows

2. **number of blocks generated:** raising the number of connections made to the cloud storage as the number of blocks generated by erasure coding or xoring grows

3. **proxy to cloud storage latency:** raising the overall response time when leaving the local network

Each configuration displayed in the heatmap is a variation on one or more of these factors.

Based on proxy-to-cloud latency, we can split the heatmap in two parts: the first column with Redis in the same cluster and the last three with the remote cloud storage configurations. This distinction highlights or rubs out the noticeable differences of performance between the various security measure combinations. Indeed the impact of computation heavy processing such as `xor` is more significant when running the experiments on the local redis database. In the first column, the response time of any combination using `xor` is systematically longer than the other combinations. But when run over remote cloud storage (the three right-most columns), the impact of the number of blocks to store using erasure coding (`cauchy`) dwarfs the longer computing time in the overall response time. For instance, in the Google Drive scenario, the average latency for the `cauchy_rsa` case is 30.2 s, whereas `aes_rsa` and `xor_rsa` achieve 1.3 s and 4 s respectively. While expected, the overhead of sending a larger number of blocks to components located out the cluster is not the only factor affecting the results. The large variance in response time to each of the providers, in particular Google drive, has previously been discussed [257].

3.7.6 Macro-benchmark — Multi-cloud latency

We evaluate latency of our system in a multi-cloud setting by configuring our testbed to use three distinct public cloud backends at the same time, namely Google Drive, Dropbox, and

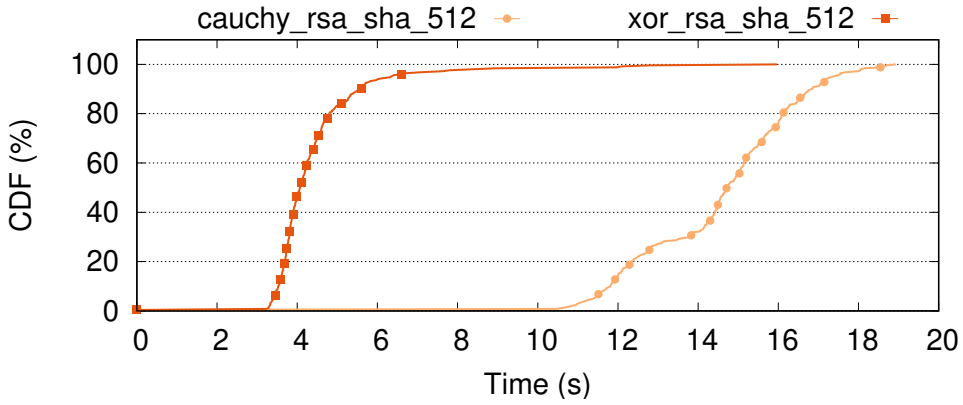


Figure 3.7: Macro-benchmark: Cumulative Distribution Function (CDF) of response time of `cauchy_rsa_sha_512` and `xor_rsa_sha_512` over 3 cloud providers.

OneDrive. We choose the driver combinations that provide the highest degree of security in a multi-cloud deployment: `cauchy_rsa_sha_512` and `xor_rsa_sha_512`. We compare the response time when inserting 250 blocks of 1 MB. We present the cumulative distribution function (CDF) of the results for `cauchy_rsa_sha_512` and `xor_rsa_sha_512` in Figure 3.7.

We observe that the exclusive-or based driver `xor_rsa_sha_512` is considerably faster than the erasure-coding driver `cauchy_rsa_sha_512`. For example, the 50*th* percentile of the former is below 4 s whereas the latter is at 14.7 s. The explanation is twofold. First, in the coding phase, the exclusive-or is a computationally efficient information in particular when compared to erasure coding. And second, in the dispatching phase, the number of blocks to distribute strongly weighs on the total time to store a file, 3 for `xor_rsa_sha_512` and 14 for `cauchy_rsa_sha_512`, in spite of the amount of data to upload, 3× the size of file vs 1.4×. This is partly due to our implementation that uploads blocks sequentially. While parallelizing the distribution of blocks seem like an obvious optimization, it is important to understand that this change could overwhelm the machine hosting the proxy and the potential performance gain could itself be cut down by cloud provider's available bandwidth and/or rate-limiting.

3.7.7 Macro-benchmark — Resource usage

Next, we evaluate the resource requirements (CPU and live memory) for each of the security configurations. These results intend to unveil the hidden costs that clients need to face when using systems that offer such security guaranteed deployed in environments with constrained resources (embedded devices, smartphones, etc.).

Table 3.4 presents a comprehensive survey of our experimentation. We evaluate single- and multi-cloud deployment scenarios, for each of the different combinations of security mechanisms. We evaluate three distinct configurations: Rows 1-4 for single-cloud, Rows 5-8 for the `cauchy` driver on multi-clouds, and Rows 9-12 for the `xor` driver on multi-clouds. The first row of each configuration on the multi-cloud deployments defines the baseline, and the subsequent rows indicate the overhead over the baseline.

The CPU time presents the percentage of time used by the processors when encoding or decoding a block. Memory consumption is presented in absolute numbers (in MB) for the baseline configurations (in bold) while the other configurations display a percentage variation. For the single-cloud deployment we consider `aes` as the baseline configuration, and report the results for the other drivers as a ratio against it. Similarly, for the multi-cloud deployment,

Table 3.4: Macro-benchmark: CPU and memory usage at encoding time of different security mechanisms (single- and multi-cloud). Baseline configurations in bold rows followed by resource usage variations as percentages.

Deployment	Drivers	CPU			Memory Used		
		4MB	16MB	64MB	4MB	16MB	64MB
Single-Cloud	**aes**	**8.26%**	**10.52%**	**11.15%**	**518 MB**	**904 MB**	**2448 MB**
	aes_sha_512	+5.93%	+1.62%	+0.81%	+1.74%	+0.22%	-1.35%
	aes_rsa	+15.25%	+2.38%	+0.90%	+3.47%	+1.44%	-0.74%
	aes_rsa_sha_512	+11.50%	+3.04%	+1.43%	+5.79%	+2.43%	-1.27%
Multi-Cloud	**cauchy**	**9.84%**	**10.81%**	**10.62%**	**532 MB**	**917 MB**	**2413 MB**
	cauchy_sha_512	-1.32%	-1.94%	+1.51%	+0.38%	+0.00%	+0.12%
	cauchy_rsa	-0.20%	-1.11%	+1.98%	+4.70%	+1.64%	+0.87%
	cauchy_rsa_sha_512	+1.12%	+0.00%	+2.82%	+3.76%	+1.09%	+0.62%
	xor	**10.49%**	**11.20%**	**10.91%**	**521 MB**	**887 MB**	**2322 MB**
	xor_sha_512	-1.05%	+0.09%	+0.55%	+3.54%	+2.03%	+1.94%
	xor_rsa	+0.67%	+0.45%	+1.19%	+8.45%	+5.30%	+2.93%
	xor_rsa_sha_512	+0.57%	+0.63%	+1.65%	+5.76%	+4.40%	+3.96%

Table 3.5: Macro-benchmark: CPU and memory usage at decoding time of different security mechanisms (single- and multi-cloud). Baseline configurations in bold rows followed by resource usage variations as percentages.

Deployment	Drivers	CPU			Memory Used		
		4MB	16MB	64MB	4MB	16MB	64MB
Single-Cloud	**aes**	**7.74%**	**9.11%**	**9.32%**	**415 MB**	**462 MB**	**615 MB**
	aes_sha_512	+16.54%	+12.51%	+9.01%	+3.86%	+4.11%	+5.85%
	aes_rsa	+19.90%	+14.27%	+10.09%	+6.51%	+5.84%	+6.99%
	aes_rsa_sha_512	+24.55%	+19.21%	+13.41%	+6.75%	+9.31%	+8.62%
Multi-Cloud	**cauchy**	**8.17%**	**9.57%**	**9.01%**	**439 MB**	**497 MB**	**699 MB**
	cauchy_sha_512	+12.48%	+9.30%	+9.32%	-0.23%	+0.80%	+1.57%
	cauchy_rsa	+14.32%	+10.24%	+10.21%	+1.59%	+3.02%	+2.86%
	cauchy_rsa_sha_512	+19.58%	+14.63%	+14.32%	+2.51%	+4.63%	+4.01%
	xor	**7.20%**	**8.12%**	**7.85%**	**409 MB**	**471 MB**	**705 MB**
	xor_sha_512	+19.17%	+20.81%	+19.36%	+5.13%	+7.43%	+8.37%
	xor_rsa	+23.47%	+23.03%	+20.51%	+12.96%	+16.99%	+12.62%
	xor_rsa_sha_512	+31.94%	+30.91%	+28.66%	+9.05%	+12.95%	+13.33%

we consider `cauchy` and `xor` as the baseline measurements and present the other results by comparison.

CPU usage does not vary significantly between different approaches for the encoding process. However, different security techniques yield contrasting resource usage in the decoding process. The most CPU-demanding encoder component is the `xor_rsa_sha_512`, with an increase of 31.94% for 4 MB blocks during decoding operations.

We also measured the live memory consumption of each configuration. Once again, the `xor_rsa_sha_512` reveals to be the most memory-demanding configuration with an increase of 16.99% over the steady operational mode.

3.7.8 Macro-benchmark — Storage overhead

Finally, we consider storage space requirements. In a multi-cloud scenario, the storage providers need to accommodate more than the source data's original size. The expected impact of the erasure coding techniques, as well as `xor`ing data in terms of storage usage was

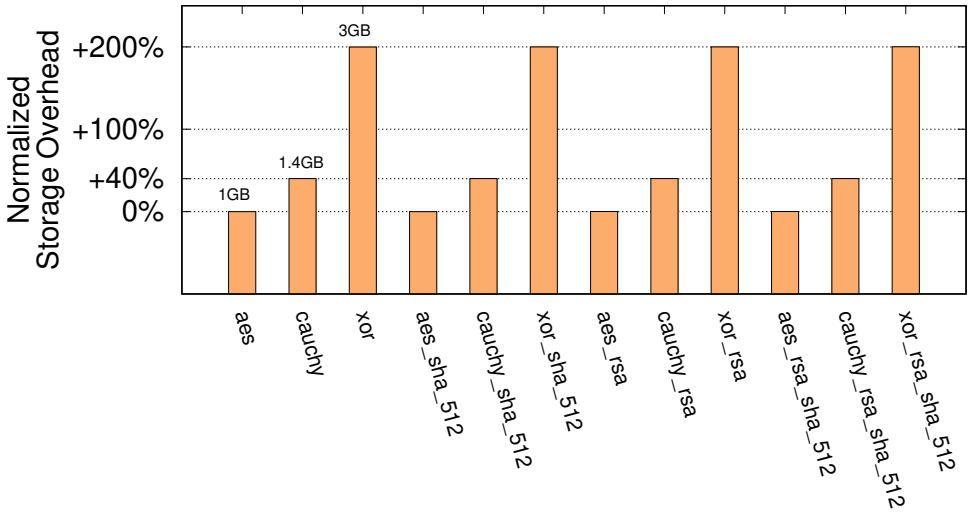


Figure 3.8: Macro-benchmark: Single-cloud - Storage overhead. Storing 500 files of 2 MB in a Redis instance.

presented in Table Table 3.3. The original block size is b. In our implementation, the XOR technique requires n times more space than the original size, with n being the number of storage providers. The erasure-coding drivers will conversely require much less space. For instance, for 64 MB blocks, `cauchy_rsa_sha_512` spends an extra 45.88% of storage space. We confirmed these observations by measuring the effective storage overhead induced by all the drivers when storing 500 files of 2 MB each on a local Redis server. The results are presented in Figure 3.8.

Note that evaluating the storage usage impact of security techniques is of particular relevance since it typically implies additional cost. As a result, storage space is a key variable to take into account in the decision of which security features to add to a particular system.

3.8 Lessons learned

We started our evaluation with a performance study of several open-source erasure coding libraries that are widely used to implement error correction in distributed systems. Our observations notably highlight the importance of specific hardware instructions such as SIMD to improve performance, the negligible overhead of using coding libraries in high-level languages like Python, the good space efficiency of erasure codes for fault-tolerant storage, and the relatively high cost of the reconstruction of missing blocks as compared to regular decoding operations.

Considering the rest of our evaluation experiments, some results stand out. First, the `aes` encryption driver proved to be the most balanced solution with a virtually constant performance across all benchmarks. As seen in Figure 3.5, the encoding and decoding throughput is very similar across benchmarks, only with an average difference of 20 MB/s at most. Additionally, we can observe that the measured latency of block encryption with `aes` is very similar to the latency of uploading a block to a storage provider. This follows from the fact that `aes` encryption does not increase block size and that the latency of the computational part of the algorithm is negligible.

Second, the `cauchy` driver has a similar discrepancy between the encoding and decoding throughput (30 MB/s on average) but, in absolute values, it always exhibits lower throughputs

when compared with `aes` encryption. Moreover, the need to generate 14 blocks has a small impact on the throughout when considering a low-latency deployment such as Redis. Conversely, when using cloud providers, the impact is highly significant. This also has a significant negative impact when using entanglement on the proxy, where in the worst case it has a difference of 8s. We note however that the number of generated blocks can be configured and `cauchy` is the only driver capable of tolerating the failure of a cloud provider.

Third, despite the fact that the `xor` driver has the biggest difference of throughput in the micro-benchmark, it has better performance than the `cauchy` driver on a real deployment. This is due to the fact that only three blocks are generated by the driver, which implies less uploaded data and, consequently, less communication latency. This happens with and without entanglement. Notably, while in the worst case the `xor` driver with entanglement exhibits a latency around 10 s, the `cauchy` driver with entanglement only achieves comparable performance in the best case scenario.

In summary, the `aes` driver protects the users information with minimal overhead, but stores all the information in a single cloud. The `cauchy` driver ensures the privacy of the users data while supporting the failure of a single cloud provider, however this has a significant cost in processing and latency. For a middle ground approach, the `xor` driver protects the users information by dividing the information among multiple cloud providers with a smaller cost on latency but does not support the failure of a cloud. Finally, security measures such as integrity, origin authentication and anti-censorship have a relatively small impact on the latency when considering baseline encryption and provider latencies.

Regarding the latency across cloud providers, if the `aes` encryption on Redis is considered as a baseline, Dropbox exhibits the lowest latency on average, with an increase of 77%. Google Drive increases the latency by 108%. Finally, in our experiments Onedrive had the worst latency, with a latency increase of 192%.

3.9 Summary

In this chapter, we have compared a wide range of security mechanisms that can be used to protect data stored in the cloud. Our experimental study sheds light on the performance and memory overheads incurred with increasing levels of security. Unlike previous studies, we consider the trade-offs of security mechanisms when used in isolation, as well as the security guarantees they provide, so that users can take informed decisions about which ones to use depending on their specific needs.

Our experiments were conducted using a testbed that we built and deployed across several standard cloud-based storage services. Unsurprisingly, we observe noticeable degradation of the throughput of block encoding with increasing layers of security. The impact of security guarantees is mainly visible in terms of CPU usage, which in turn yields increased latency, but we also observe some variations in terms of memory consumption. Furthermore and as expected, throughput is generally higher and more stable in single-cloud deployments.

To sum up, the lessons to take away from our study are that there is no single combination of security measure that performs best for all applications. Instead, users need to carefully chose the minimum set of mechanisms that can match their security requirements. In turn, cloud provides need to provide security measures that can be freely combined instead of proposing a "complete package", as every additional security layer comes with an associated cost. We hope that our experimental results will provide valuable insights to both service

providers and their users, and can be instrumental for improving cloud-based storage systems and developing applications that can best leverage them.

Chapter 4

RECAST: Random Entanglement for Censorship-resistant Archival STorage

4.1 Introduction

Users entrust an increasing amount of data to cloud systems for long-term archival purposes. This archiving comes with expectations for the data to remain available and readily accessible at any time. Hence, an archival system must be highly durable. However, there are many threats to durability, and the longer the lifetime of the content is, the worse it gets.

Broadly speaking, one can distinguish between passive threats (*e.g.*, hardware failures) and active threats (*e.g.*, a censor wishing to delete a specific document). A common countermeasure to mitigate passive threats is redundancy: extra information is stored in the system, with the goal of tolerating node or disk failures (see Section 2.3). However, effectively defending against active attacks still presents open challenges. Various solutions have been proposed, none of which are fully satisfactory. For example, security by obscurity [165, 66] encrypts content before archival so that the attacker cannot distinguish the data to censor, but this severely limits the operation that can be performed on data and the way it can be accessed or shared. Data entanglement [252, 279], on the other hand, attempts to weave together popular and unpopular data so that an attacker is forced to do collateral damage when censoring its target, but in a way that is unfit for deployment in practical systems.

We introduce RECAST, an anti-censorship data archival system based on a random data entanglement strategy proposed in [173] that provides strong guarantees while being readily applicable to real systems. RECAST's design and entanglement principles are based around the following intuition. First, by entangling pieces of data with one another, we enable potentially unpopular and rarely accessed content to benefit from the protection offered to other data in the system. Second, the asymmetry of the construction makes the system easy to repair and hard to corrupt. Indeed, while the randomness component of RECAST's entanglement makes it NP-hard to compute the minimal set of documents to be deleted to censor a target [173], error-correcting codes allow to repair the system recursively without increasing the storage overhead as long as the damage done by the attacker is recoverable.

To assess the security of RECAST, we assume a powerful adversary model where the censor has access to the metadata and thus knows how documents are processed, split, and distributed over the storage nodes. To further protect the system against a powerful attacker targeting metadata, RECAST includes an emergency disaster recovery mechanism that enables the system to rebuild metadata from the data itself. An alternative protection exist in the form of an extension to RECAST that replicates metadata over a set of private blockchain nodes [141] (discussed in Section 4.9.1).

We measure the protection by collateral damage units, that is, the number of additional documents to be deleted by a censor when targeting a specific document. In that respect, we design RECAST so as to offer strong long-term as well as fast short-term protection. We first build upon *uniform data entanglement* [173], which offers very strong long-term protection but leaves recently archived documents poorly protected. To address this limitation, we rely on *normal data entanglement* and temporary replication that provide fast short-term protection, but do not spread entanglement adequately across the archive. To get the best of both worlds, we introduce a hybrid approach, `nu`-entanglement, which exploits the strong long-term protection of uniform entanglement and the fast short-term protection of normal entanglement and temporary replication.

The rest of the chapter is organized as follows. We summarize our design goals in Section 4.2. We discuss previous entanglement schemes in Section 4.3. In Section 4.4 we briefly describe entanglement and the STEP-archival strategy [173]. In Section 4.5 we propose a new practical technique to data entanglement, called `nu`-entanglement, which offers fast short-term and strong long-term protection for all the documents in the archive. We discuss the design and implementation details of the RECAST prototype in Section 4.6 and Section 4.7, and evaluate both security and performance in Section 4.8. We discuss possible extensions and deployment considerations in Section 4.9. Finally, we present our conclusions in Section 4.10.

Please note that the work in this chapter was originally spearheaded by Roberta Barbi and presented as a conference paper [17]. As a result, contributions such as improvements to STEP's pointer selection, reliability modeling and simulation work can be found in her PhD dissertation [16]. While some of these elements are repeated here for the sake of completeness, this chapter focuses on the design and evaluation of the RECAST prototype.

4.2 Design goals

In this work, we aim at designing a long-term censorship-resistant system ensuring content integrity and durability. We understand durability as the ability to eventually retrieve any of the archived documents.

While high availability and confidentiality of data are desirable features of any storage solution, durability is paramount to long-term archival systems. Indeed, maintaining high availability may get expensive over time for documents that are infrequently accessed. Moreover, maintaining data confidentiality and managing the associated encryption keys over a long period of time (*e.g.*, decades) unnecessarily exposes the system as a central point of failure. With time, encryption algorithms are broken and keys once considered large enough prove to be too short. Therefore, confidentiality is best left deferred to the client's discretion. In contrast, the ability to retrieve documents for audit purposes or disaster recovery at a configurable cost answers a more practical need, especially in a context where independent providers operating as storage backends may become unreliable over time or worse, may attempt to censor, tamper or delete user content.

To address this problem, RECAST ensures documents durability by establishing random interdependency links between documents at storage time, which enables recursive reconstruction of data beyond the local capability of the underlying erasure code. The trade-off is that documents cannot be updated nor deleted from the archive. A file update must consist in a new file upload, and we do not offer file removal as enabling this operation

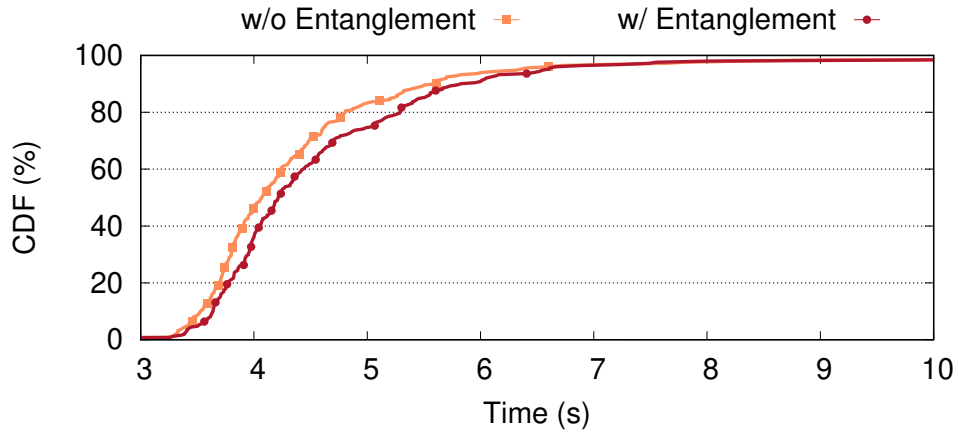


Figure 4.1: Dagster on top of `xor_rsa_sha_512` over 3 cloud providers.

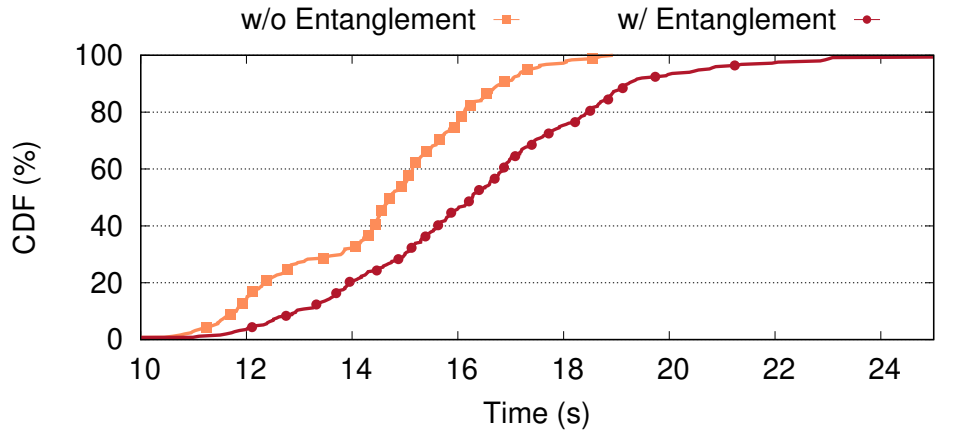


Figure 4.2: Dagster on top of `cauchy_rsa_sha_512` over 3 cloud providers.

for a user would enable it for the censor as well. This means that a system administrator compelled by law to remove content [73] must work as hard as an attacker who wishes to censor a file without causing collateral damage, *i.e.*, recode the entire system starting from the file to delete.

4.3 Data entanglement

The problem of censorship-resistance in distributed storage systems has been studied extensively and, as seen in Section 2.3, can be solved with different approaches. One of these approaches is data entanglement [13]. In this section we expand on the topic of entanglement, the implementation and shortcomings of existing solutions.

Data entanglement is the process of creating interdependencies between pieces of data. In essence, storing a new document creates links to older pieces of data. Upon a read request of the new document, the older pieces it was linked to must also be fetched for successful decoding. If any or some of the linked pieces cannot be retrieved, the read request fails. Data entanglement effectively minimizes the risk of silent corruption by propagating the damage from a single piece of data to a larger part of the archive.

To ensure that effective dependencies are created between documents, the different pieces of data are usually tied together through coding. Following the code chosen, the

new piece of data is combined with older pieces. The resulting codeword is stored and the original piece of data is discarded. To recover the original piece of data, we must then retrieve the codeword and some or all of the older pieces used in the coding. While they differ in the codes used, this high-level description covers the behavior of 2 schemes: Tangler [279] and Dagster [252]. The first one, Tangler uses Shamir secret-sharing [240] to encode new documents. Secret-sharing makes Tangler interesting for environments where some pieces used to generate the codeword may not be available at times. But it remains expensive as it computes codewords using elliptic curves, a form of public key cryptography, and requires the storage and communication of codewords that are several times larger than the original piece of data. To answer the cost issue, Dagster offers a cheaper but more brittle alternative by combining the new and older pieces of data using an exclusive-or (XOR), an inexpensive operation for most modern computers.

Building an entanglement scheme such as Dagster on top of an existing distributed storage system is possible but comes at a cost. To illustrate this, we implemented a Dagster-like scheme in the testbed presented in Section 3.4. In Dagster, the size of documents and blocks is identical. When a new document D must be stored, Dagster randomly chooses c blocks already archived and xor them with D.

We evaluate the overhead of the entanglement by configuring our testbed to use three distinct public cloud backends at the same time, namely Google Drive [98], Dropbox [72], and OneDrive [190]. We chose 2 of the most complex configurations our testbed could provide: `cauchy_rsa_sha_512`, and `xor_rsa_sha_512` (see Table 3.2 in Section 3.7.1). Both configurations are encrypted and signed but the first one is erasure-coded where the second is coded with a one time pad. We compare the latency of inserting 250 blocks of 1 MB with and without entanglement for both drivers. Once blocks are entangled, they are sent to the chosen provider in a round-robin fashion to balance the load. We present the cumulative distribution function (CDF) of the results for `xor_rsa_sha_512` and `cauchy_rsa_sha_512` in Figure 4.1 and Figure 4.2 respectively.

In both cases, we can see that the overhead induced by the entanglement phase is modest. In particular, in Figure 4.2, the entanglement only adds a +18.1% latency overhead for the $95th$ percentile of the blocks. In the Figure 4.1 scenario, this overhead is lowered to +0.3%. These results show that a multi-cloud entanglement scheme can be practically operated by clients with a moderate performance penalty when compared to the default, non-entangled operational mode.

However, latency should not be the only factor to consider. In this entanglement scheme, the corruption of any of the $c+1$ blocks needed for reconstruction makes the document unreadable. It voids the capability of configurations such as `cauchy_rsa_sha_512` that are built to tolerate the loss or corruption of some blocks through erasure coding. What was gained in performance has been lost in fault tolerance. For a balanced approach, we look at STeP-archives in Section 4.4

4.4 STeP-archival

In this section we leverage the technique presented in [173] for creating interdependencies between data to be stored and data already archived in the system. Upon archival, the blocks of a document are entangled with some blocks of documents previously archived in the system. The entanglement builds strong ties between content, preventing silent censorship of

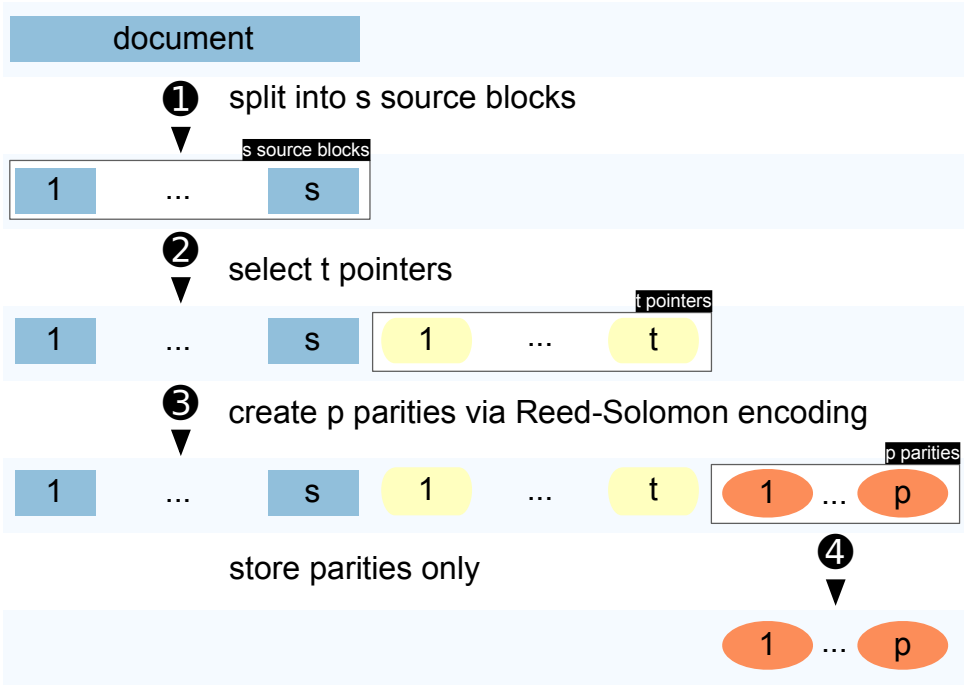


Figure 4.3: Entanglement logical flow with STEP.

rarely accessed data. An attacker wishing to censor a target document must cause collateral damage by corrupting several other archived documents.

More formally, a (s, t, e, p)-archive [173] is a storage system where each archived document consists of a codeword of s source blocks, t pointers or tangled blocks (*i.e.*, old blocks already archived in the system), p parity blocks, and that can correct $e = p - s$ erasures (*e.g.*, missing blocks) per codeword.

The logical flow to archive a document in a (s, t, e, p)-archive is illustrated in Figure 4.3. The document to be archived is split into $s \geq 1$ source blocks (Figure 4.3-❶). From the archive, t distinct old blocks, the *pointers*, are selected (Figure 4.3-❷) and a Reed-Solomon (RS) code is used to create $p \geq s$ parity blocks (Figure 4.3-❸) depending on both source blocks and pointers. In particular, the p parity blocks are computed using a RS$(s+t+p, s+t)$ code [222], which encodes $s+t$ blocks into a $(s+t+p)$-long codeword with the property that any $s+t$ blocks of the codeword are necessary and sufficient to reconstruct all its $s+t+p$ blocks [160]. In the last step, we *only archive the p parity blocks* (Figure 4.3-❹), making the code non-systematic. The t pointer blocks come from the archive, so we do not need to store them again. The choice of not storing the source blocks enhances security [173]. As a trade-off, read performance is degraded both in terms of latency and bandwidth since reading a document from a STEP-archive always requires a decoding operation involving $s+t$ blocks, as opposed to a systematic code where the s alive source blocks can be directly accessed.

STEP-archive asymmetry. To censor an archived document, it is not always sufficient to destroy its code blocks because they can sometimes be recovered recursively. On the one hand, it has been proven [173] that finding the minimal set of documents required to irrecoverably censor a target document is NP-hard. On the other hand, the system can repair recoverable attacks using a simple and efficient reconstruction algorithm: we first scan the archive and build a set C of corrupted document with at most e erased blocks. We pick a document from C, repair it, and update the set of documents with at most e erased blocks

Table 4.1: Notation. For example, we write `u-l-(s,t,e,p)-r`α for the leaping attack running on a (s, t, e, p)-archive with uniform entanglement and keeping α temporary copies for each document in the tail of the archive. The replication parameter is omitted when temporary replication is not in place.

Symbol	Description
u	uniform entanglement
n	normal entanglement with standard deviation $\sigma = 1000$
nσ	normal entanglement with standard deviation σ
nu	normal-uniform entanglement with standard deviation $\sigma = 1000$
nuσ	normal-uniform entanglement with standard deviation σ
l	leaping attack
c	creeping attack
rα	level of temporary replication α

in C. The algorithm stops when C is empty. At this point, either the system is completely repaired or there is a closed set of documents with strictly more than e erased blocks.

Flavors of entanglement. Different flavors of entanglement are studied in [173]. Uniform random entanglement selects pointer blocks uniformly at random. The randomness of this approach is an advantage as pre-planning an attack against such a structure is not feasible. Uniform entanglement has the main drawback of requiring increasingly longer periods of time to protect young documents as the archive increases. For example, Figure 4.4 shows that after inserting 10000 documents, 25% them are either not pointed to or pointed to only once. An attacker wishing to tamper with these documents can thus do so without causing collateral damage. The authors of [173] also show how to speed up the protection of new documents by selecting the pointers to entangled blocks from a sliding window bounded to the recent past, although this has the drawback to bound the entanglement level of the documents in the archive.

Practical resilience to censorship. Since censoring a document optimally is NP-hard, to evaluate the censorship resistance of the system, one can rely on suboptimal heuristics. We exploit the greedy attacks as well as a branch and bound technique introduced in [173]. We include a brief summary and refer the reader to [173] for more details.

We play the attacker and select a target to be censored. Erasing blocks in the target produces erasures in other documents because of the entanglement. To prevent recursive repair we must corrupt those documents as well. So at each step of the attack we face the choice of which block to delete and use one of two greedy heuristics to make the decision:

- The *leaping attack* leverages the fact that it is easier to attack recent documents rather than old ones (because they have fewer incoming pointers).
- The *creeping attack* tries to keep the set of corrupted documents as compact in time as possible by preferring documents having approximately the same archival date, which is very effective against window-constrained entanglement strategies.

It is possible to improve on these greedy heuristics by means of a branch and bound technique: at each step of the computation of the minimal set of blocks to be erased, we retain the best partial solutions up a certain buffer size and expand all of them. Throughout the chapter we adopt the notation summarized in Table 4.1.

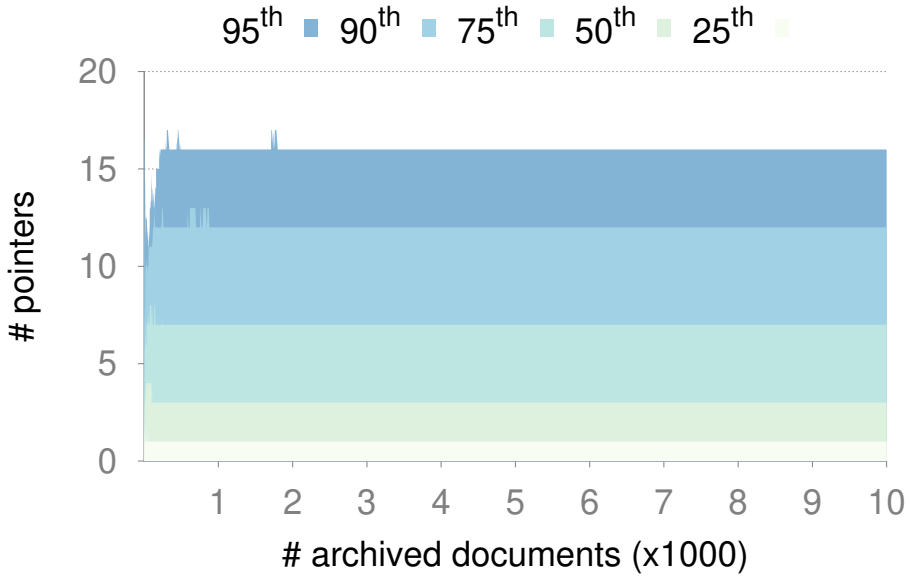


Figure 4.4: Variation in the number of pointers to each document in a u-$(1, 5, 2, 3)$-archive with 10^4 documents.

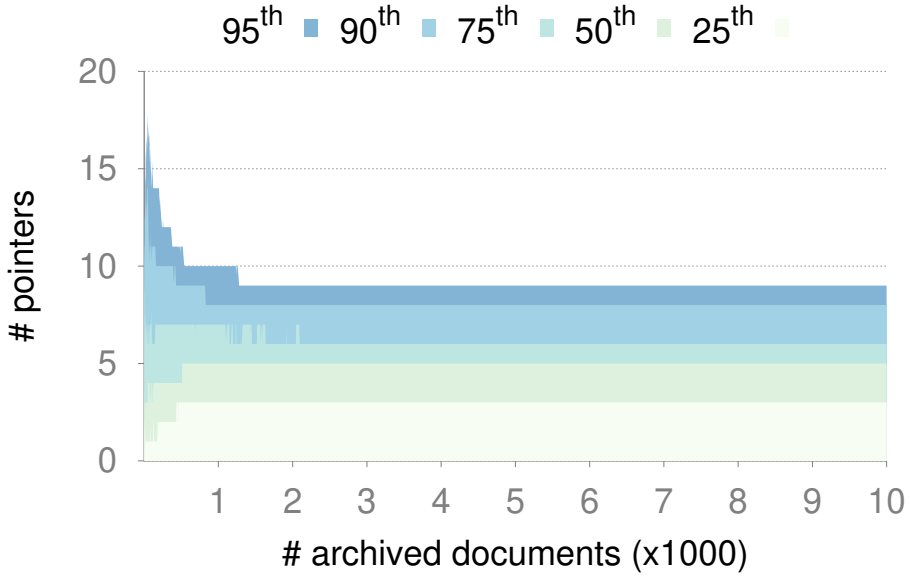


Figure 4.5: Variation in the number of pointers to each document in a n-$(1, 5, 2, 3)$-archive with 10^4 documents.

4.5 Hybrid entanglement

In this section, we leverage the benefits of uniform and window-based normal entanglement strategies. We present a hybrid entanglement technique mixing pointers chosen uniformly at random and pointers chosen from a normal distribution. We further enhance anti-censorship for recently archived documents using temporary replication.

Normal entanglement. To archive document d_{i+1}, we extract t pointer blocks from a left half-normal distribution with standard deviation σ centered in the last archived document d_i. This models a sliding window of about size 2σ, in which the probability to point to a document older than $d_{i-2\sigma}$ is 5%, and the probability to point to a document older than $d_{i-3\sigma}$ is 0.3%. On the one hand, this means that documents are quickly protected: we

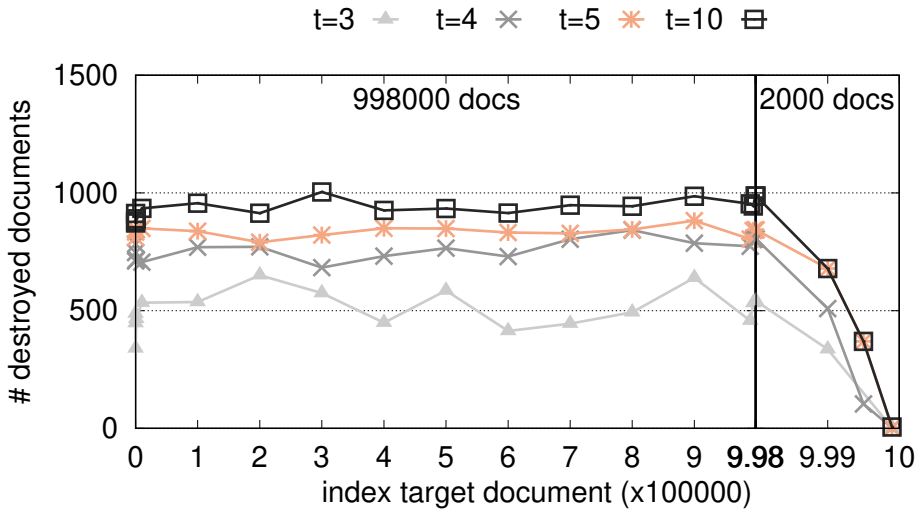


Figure 4.6: Number of corrupted documents to erase a target (x-axis) with a creeping attack in a n-$(1, t, 2, 3)$-archive with $t = 3, 4, 5, 10$. Notation in Table 4.1.

just need to wait for 2σ documents to reach the average protection offered by the archive. Furthermore, this does not depend on the size of the archive but only on the chosen standard deviation σ. Comparing numbers, we see that the first quartile (25%) grows from pointed to at most once under uniform entanglement (Figure 4.4), to pointed to at most three times under normal entanglement (Figure 4.5).

As discussed in the previous section, approximating a sliding window bounds the propagation of the entanglement against which the creeping attack is very effective. To confirm this intuition, we simulate the creeping attack on an archive storing 10^6 documents. As expected, the average number of documents to be censored, *i.e.*, the protection, is constant through the archive and only the last 2000 documents have lower protection (Figure 4.6, right-most side of the x-axis). As expected, increasing the number of pointers t improves the average protection against the creeping attack: in Figure 4.6 the average number of documents to be destroyed to censor the target is about 506 for $t = 3$, 758 for $t = 4$, 835 for $t = 5$ and 935 for $t = 10$.

Hybrid `nu`-entanglement. To overcome the limitations of the entanglement techniques described so far, we blend normal and uniform entanglement together. We select each of t pointers by flipping a coin, so that with probability $\frac{1}{2}$ we have:

- a uniform pointer, providing the good randomness and the strong long-term protection of uniform entanglement,
- a normal pointer, offering the fast short-term protection of normal entanglement.

The best greedy heuristic on a STeP archive with `nu`-entanglement is the leaping attack, indeed the effectiveness of the creeping attack drops thanks to the restored randomness. In Figure 4.7, we observe a greater influence of the number of pointer blocks on the protection. In particular for $t = 5$, the long-term protection grows by almost 3 orders of magnitude with respect to normal entanglement and the short term-protection grows by about 2 orders of magnitude with respect to uniform entanglement.

We compare the three entanglement methods in Figure 4.8 over an archive of 10^6 documents. Using uniform entanglement, around $6 \cdot 10^5$ documents can be censored tampering with only 10 documents. On the other hand, using normal entanglement with $\sigma = 1000$, all

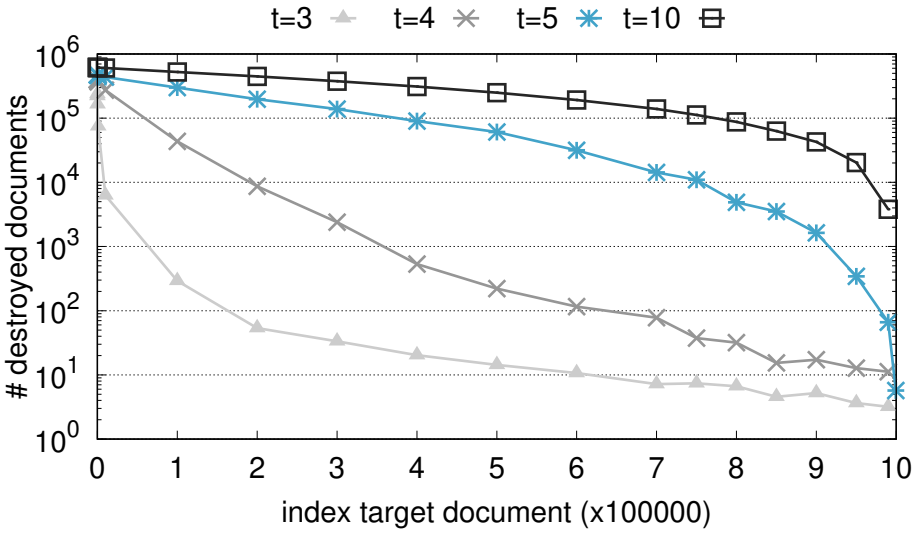


Figure 4.7: Number of corrupted documents to erase a target (x-axis) with a leaping attack in a nu-$(1, t, 2, 3)$-archive with $t = 3, 4, 5, 10$.

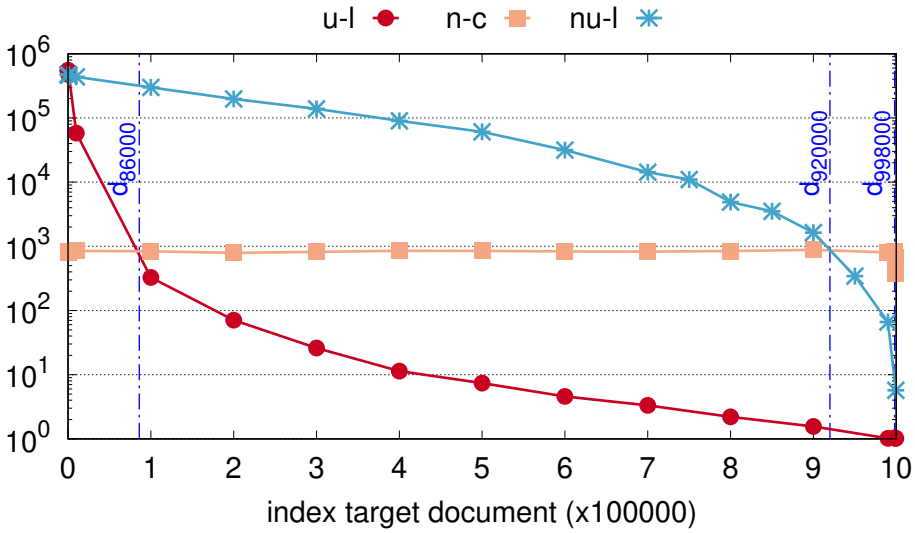


Figure 4.8: Number of corrupted documents by the most effective heuristics on uniformly-, normally- and nu-entangled $(1, 5, 2, 3)$-archives.

the documents have the same protection except for the last $2\sigma = 2000$ documents. Moreover, when normal entanglement is in place, protection is offered fast: to erase d_{998000} (a recent one), a greedy attacker needs to tamper with more than 800 documents. In contrast, using uniform entanglement, d_{998000} can be censored by tampering with one document. As discussed, nu-entanglement mixes the two approaches to gain the best of both worlds: nu-entanglement offers greater protection than uniform entanglement for all the documents older than document d_{86000} and greater than normal entanglement up to document d_{920000}. Hence, nu-entanglement outperforms the other entanglement approaches on more than the 80% of the archive when tested with suboptimal greedy attacks.

Temporary Replication. Regardless of the pointer blocks' selection method, entanglement takes time to provide protection to new documents (*e.g.*, the last archived document is not pointed to). Hence, we use temporary replication until the entanglement kicks-in. The replicas are spread over the various storage nodes. They enable fast recoveries in case of

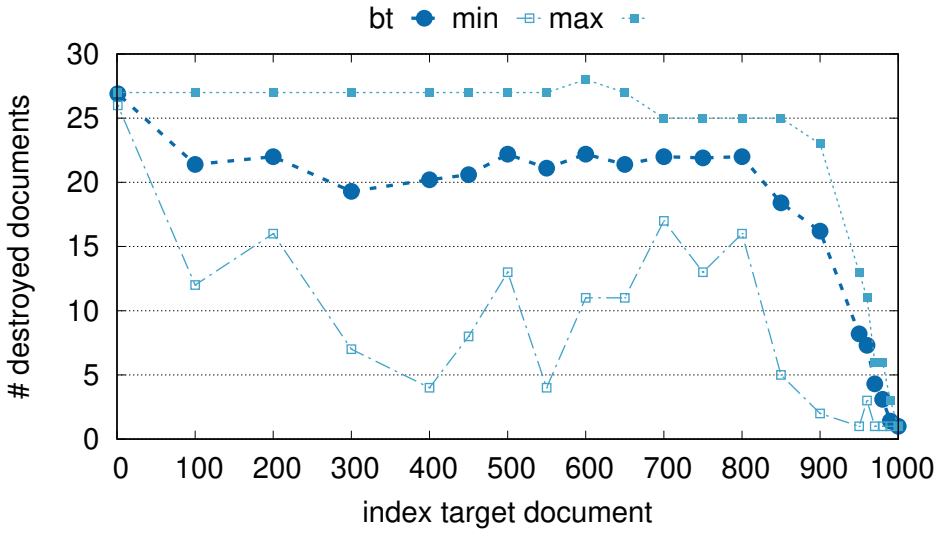


Figure 4.9: Backtracking the creeping attack on a n100-$(1, 5, 2, 3)$-archive of 1000 documents.

node failure, at the cost of increased storage overhead. To reduce such costs, we periodically examine the level of protection of blocks and once a given threshold is passed, RECAST removes the corresponding replicas from the storage nodes.

In the remainder of this section we study how to set such a threshold in a STEP-archive configured with `nu`-entanglement. We conduct this study by means of simulations, in order to establish when a given block's replicas can be safely removed. For uniform entanglement, as the number of blocks increases over time, the random selection of pointers lowers the probability of picking recent ones. Hence, the replication level and the replication threshold are determined by the normal pointers. On a normally-entangled STEP-archive with 1000 documents, we run the creeping attack to get a first estimate, then we backtrack the tree of solutions to find the optimal one, exploiting branch pruning [173] to speed up the execution time. We present the results in Figure 4.9. The average protection offered by $t = 5$ normal pointers selected with standard deviation $\sigma = 100$ is 21.8. It drops with the 800th document, *e.g.*, 2σ documents before the end of the archive. The average number of documents to be deleted is 8.2 on the full archive and 11.6 out of the tail. Hence, to guarantee homogeneous protection to a RECAST system that uses a STEP-archive with $t = 10$ `nu`-pointers when the normal pointers are extracted with standard deviation $\sigma = 100$, we decide to replicate the tail, *i.e.*, 2σ documents, 10 times each.

Summary. Uniform entanglement provides strong long-term protection but needs a massive use of replication to reach an adequate level of short-term protection. To reduce the storage overhead, we study normal entanglement which provides constant long-term protection, fast short-term protection and a level of replication independent on the size of the archive. While the cost of replicas grows linearly with the size of the archive when using uniform entanglement, it goes to zero in the case of normal entanglement. Indeed, as the number of documents that need to be replicated is constant, in the long-term it becomes a negligible fraction of the whole archive. Finally we presented `nu`-entanglement to blend the two approaches, to attain strong long-term protection with uniform pointers, fast short-term protection and a constant amount of temporary replication determined by normal pointers.

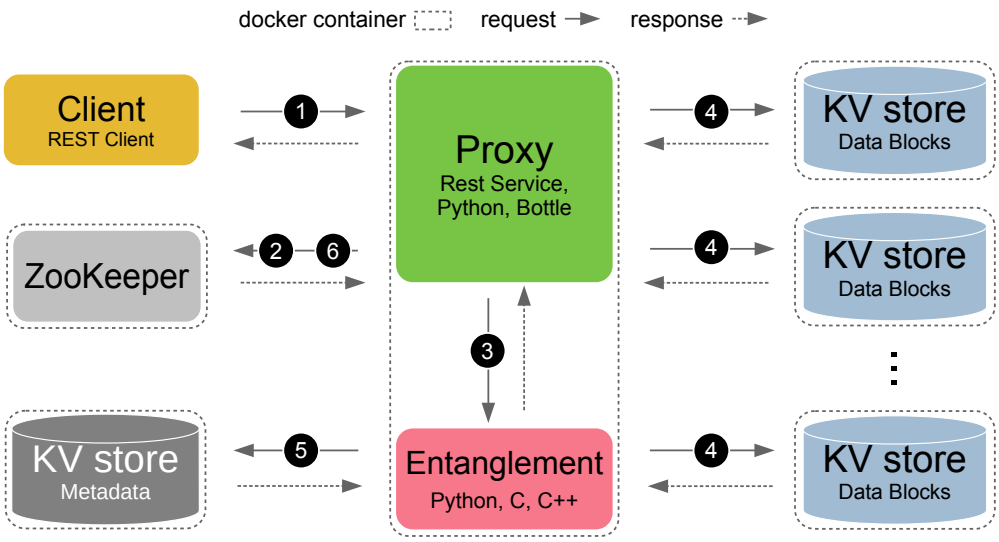


Figure 4.10: The RECAST architecture and execution flow.

4.6 Architecture

This section presents the architecture of RECAST. Figure 4.10 depicts the architecture as well as the typical operating flow for a single document insertion (a document retrieval follows a similar pattern, as described next). The RECAST architecture comprises the following components: a *proxy* that mediates interactions between clients and the RECAST system, an *encoder* that performs the entanglement operation, a *metadata server* that logs the stored documents and their entanglement links, and a set of storage nodes that serve as backends. We assume the storage backends to be deployed over untrusted, possibly malicious but not colluding service providers. The proxy is the gateway to the data and metadata in the system, and we assume it is deployed in a trusted environment, typically on the client-side. The proxy acts as RECAST's front-end coordinating the operations between the actual storage backends and the metadata server. Clients contact the proxy to write or read data through a simple REST interface issuing `PUT` and `GET` requests, similarly to Amazon S3. The configuration system of RECAST allow system deployers to tweak the number and type of storage nodes, the STEP parameters and the factor used for temporary replication.

Write operations. A client sends a write request to the proxy (Figure 4.10-❶). The proxy then acquires a write lock from Zookeeper to handle this document (Figure 4.10-❷). At this point, the proxy can delegate the entanglement operation to a dedicated encoder or handle the process by itself (Figure 4.10-❸). In both cases, the original document is split into s blocks, t random pointers are sourced from the storage nodes and used for entanglement. The resulting p blocks are then synchronously dispatched to the storage nodes (Figure 4.10-❹). Once the entanglement and storage steps are finalized, the proxy persistently saves the required information for future decoding in the metadata server (Figure 4.10-❺). This information includes the location of the parity blocks of the document and as well as the pointers' for entanglement. Finally, the proxy releases the lock and replies to the client (Figure 4.10-❻).

Read operations. Upon a read request, the proxy looks through the metadata server for the document's blocks and its pointers to reverse the entanglement. The proxy fetches the blocks from the storage nodes and delegates the decoding to the encoder (or decodes by itself if configured to do so).

Deployment considerations. The modular RECAST architecture enables the decoupling of the proxy and the component in charge of the entanglement operations. To maximize throughput, these components should be hosted on the same physical node, thus avoiding bottlenecks induced by high pressure on the network stack. For the same practical reason, the metadata server should be deployed on the same machine as the proxy. Note that these considerations do not affect the RECAST's threat model.

Entanglement and replicas management. RECAST uses STEP-based entanglement for long term protection of documents. At the same time, new blocks are replicated to provide a high level of redundancy from the very early moments the document's life in the archive. To mitigate the storage overhead that this approach could generate, RECAST manages the lifetime of the replicas as follows:

1. Upon a write request, pointers are randomly selected and used for the entanglement of the incoming document;
2. A static number of replicas is computed and spread randomly over the storage nodes;
3. As soon as any of the chosen pointers is pointed a *sufficient number* of times, its replicas are permanently removed from the system.

Hence, deployers must decide upon two parameters to configure RECAST: the replication factor and the protection threshold. The former is known by the proxy, which is in charge of computing how many copies of any block it must send to the storage nodes. The entanglement component is unaware of the replication factor, and it only asks for blocks. This design makes the management of replicas simpler and shields the component executing the block coding (or further clients) from having to deal with corrupted replicas or unavailable storage nodes. Moreover, in a configuration with homogeneous storage nodes, this level of indirection enables the proxy server to balance the load of read requests for a block over the different storage nodes holding replicas. The latter, the protection threshold, is also known and used by the proxy. In our current prototype implementation, the threshold is expressed as the number of documents pointing to a block. The choice of this metric rather than the age of the document and its blocks stems from our use of uniformly random selection of pointers. In particular, its deterministic nature prevents from the risk of exposing poorly protected documents to potential threats.

As described in Section 4.4, some old blocks may never be entangled with enough documents to reach the given threshold, which would prevent the system to use an age-based approach. In practice, we deploy a separate process running next to the proxy that periodically scans the metadata and lists blocks that have been pointed to *threshold* times. Once this list is assembled, the system removes the redundant replicas and updates the metadata server.

Metadata management. Our system leverages randomness in the selection of tangled blocks and in the placement of new blocks into storage nodes. If metadata is unavailable, damaged or lost, stored blocks become meaningless. In addition to traditional replication, RECAST implements a mechanism to reduce risks of complete loss of metadata. This procedure allows RECAST to reconstruct the metadata from the data itself. Under the assumptions of available and honest storage nodes as well as pristine data blocks, we scan the storage nodes, examining the hosted blocks and reconstruct the associated metadata (except for the creation date), see Figure 4.14.[1]

[1]Note that the creation date can also be appended, however a partial order can be inferred even without it [173].

This solution is possible because, we prepend the entanglement information to each block before sending them to the storage nodes. More specifically, given a block, this metadata-overhead includes a reference to all the pointers selected during the entanglement of the document. In our prototype, it consists of a fixed 80 Bytes per block (erasure coding information) and the list of t pointer names. As blocks are named according to the document they belong to and their position in the codeword, this average length depends on the naming patterns in the archive. For example, in the case where users do not provide names for the documents they insert, the system defaults to an ASCII version of a `uuid`. This results in average length of 38 Bytes: 36 Bytes of the serialized uuid + 2 Bytes for the position in the codeword.

4.7 Implementation details

Our implementation choices have been largely driven by performance, programming simplicity considerations and constraints from the storage backends interfaces. The proxy and the encoder are implemented in Python [213] (v2.7). In particular, the entanglement code is built on top of PyECLib [212] (v1.3.1), liberasurecode [191] (v1.1.1) and Intel ISA-L [121] (v2.20.0). Clients communicate with the proxy through the REST API implemented using Bottle [27] (v0.12.13) and exposed through uWSGI [269] (v2.0.15). Storage backends can be commercial storage providers such as Google Drive, Dropbox, Microsoft OneDrive and Amazon S3 or self-hosted key-value stores such as Redis [221] and Minio [178]. For the sake of evaluation, we only deploy storage nodes on premises. To coordinate the accesses to the metadata from the proxy, the repair and replica management processes, we use ZooKeeper [263]. Finally, all the components are wrapped and deployed using Docker [68] (v17.05.0-ce).

4.8 Evaluation

In this section, we evaluate RECAST's performance and resilience to data loss. First, we test the entanglement throughput in isolation and as part of the full writing and reading process. Then we look at the variations in metadata storage overhead depending on the number of documents and the STEP configuration. Finally, we look at the metadata reconstruction process and the repair bandwidth use.

4.8.1 Encoding/decoding throughput

We begin by evaluating RECAST's throughput capabilities in terms of raw encoding and decoding operations. These results are obtained on a Intel Broadwell 64-cores machine with 128 GB of RAM running Ubuntu 16.04 (kernel 4.4.0-101). Figure 4.11 presents results for 6 configurations of STEP having different number of pointers and code rate. Please note that some of these configurations, namely $(5, *, 2, 7)$, are too brittle to be used in practice and are only included here to show the impact of STEP parameters tuning. We measure the variations in throughput across these configurations with 3 different document sizes: 4, 16 and 64 MB. While the size of the incoming documents directly affects the performance of any configuration, in particular when the codewords fit in low-level caches (40 for the L3-caches on our servers), the most determinant factor is the storage overhead. Indeed, both in encoding and decoding, results can be grouped according to storage overhead (`nu`-$(1, *, 2, 3)$

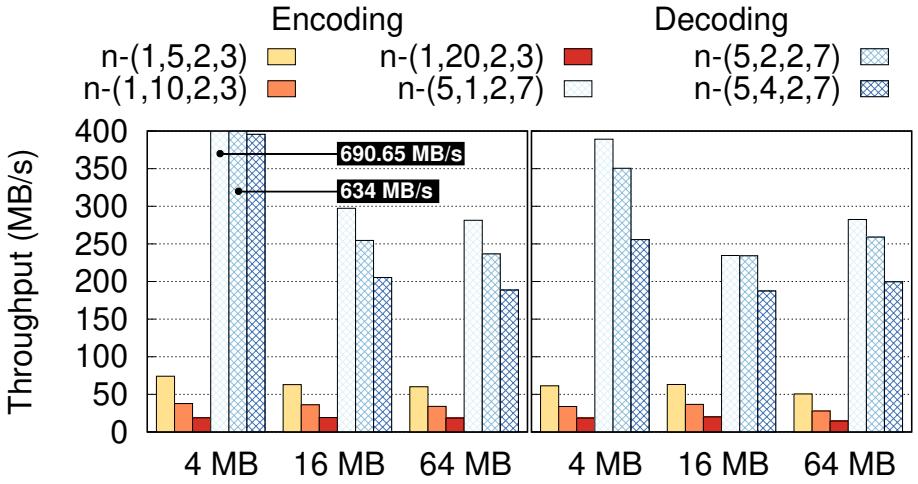


Figure 4.11: Encoding and decoding throughput of the entanglement process in isolation.

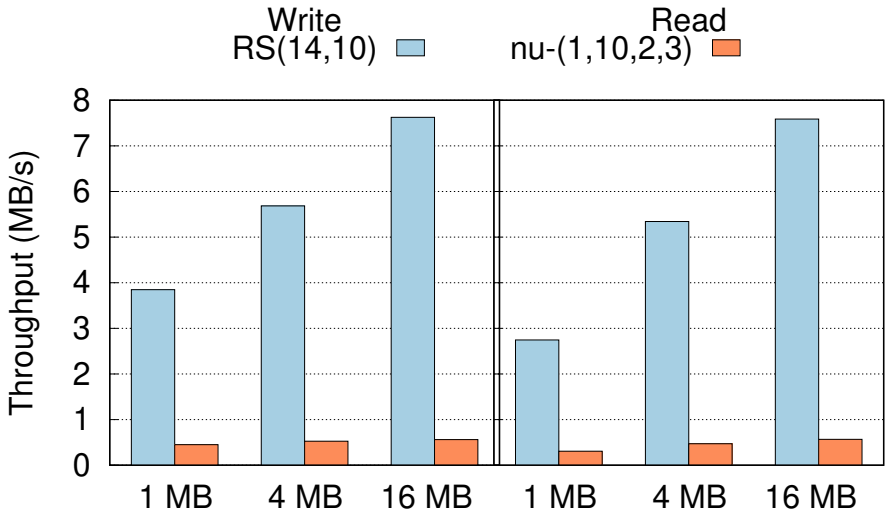


Figure 4.12: Writing and reading throughput of the entanglement process compared to RS(14,10) in a full RECAST instance.

and nu-$(5, *, 2, 7)$). Second to the storage overhead, the number of pointers also affects the throughput with a predictable slowdown as the number of pointers grows. Looking at the first group of configurations, nu-$(1, *, 2, 3)$, we observe that the throughput halves every time the number of pointers doubles. Indeed, to encode a 16MB document, we go from 63 MB/s ($t = 5$), on to 36 MB/s ($t = 10$) and end up at 19 MB/s ($t = 20$). The second group of configurations, nu-$(5, *, 2, 7)$, also endures a predictable slowdown but to a lesser extent (15-20%) every time we double the number of pointers. To encode a 16MB document, we drop from 297 MB/s ($t = 1$), on to 254 MB/s ($t = 2$) and finally stop at 205 MB/s ($t = 4$). This difference in behavior pairs with the rising storage overhead and the sensitivity of the different configurations to these changes as we introduce more pointers. In conclusion, Figure 4.11 shows the trade-offs in choosing a configuration for STEP. It may offer good protection, *e.g.*, nu-$(1, 20, 2, 3)$ at the expense of a low throughput (up to 18.84 MB/s when encoding) or a very fragile alternative, such as nu-$(5, 1, 2, 7)$, reaching up to 690 MB/s.

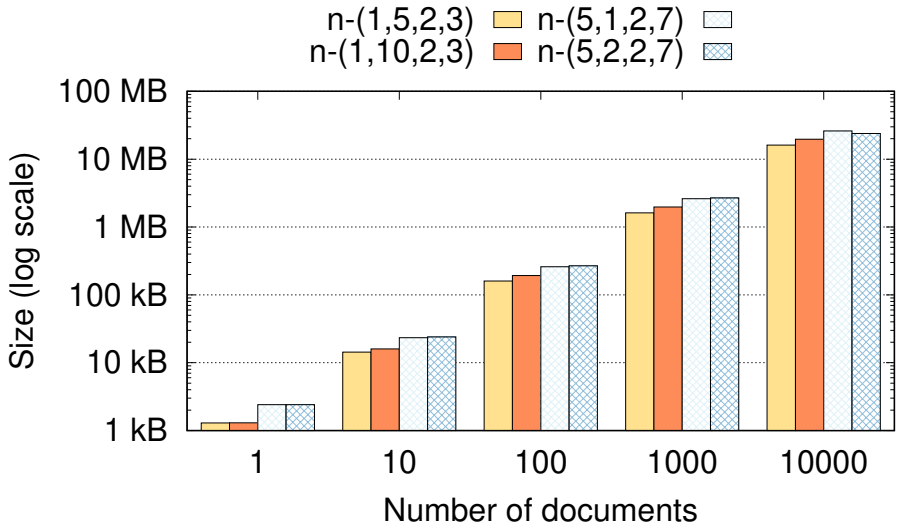


Figure 4.13: Metadata storage overhead.

When integrated in the full system, we compare the performance of our entangled archive against the standard Reed-Solomon (RS) code provided by Intel ISA-L. By means of the YCSB [52] framework, we run two workloads of 1000 operations: a read oriented one based on YCSB's `workloadc` and a pure insert one. Both workloads run 1000 operations, with payloads ranging from 4 to 16MB. We deploy an instance of Recast that runs on a single machine (64 cores and 128GB of RAM) hosting proxy, coder, metadata and 16 storage nodes. We run this experiment with 8 concurrent threads from a remote machine connected to Recast host on 1GB switched network. Considering the throughput measured with these workloads (see Figure 4.12), we observe that the entanglement slows down the operations by 10x. This can be explained by the volume of pointers that flows through the system when reading or writing (10 times the size of the original data) in addition to the necessary manipulations of the pointer blocks.

4.8.2 Metadata storage overhead

Next, we look at the storage overhead introduced by our prototype. While the STeP configuration, the number of documents and their average size affect the storage requirements, the metadata growth (see Figure 4.13) is more dependent on our implementation.

The visible differences across the STeP configurations are due to the varying number of pointer blocks t: the greater t grows, the greater the protection offered by the archive is and as a consequence, the number of entanglement links to maintain increases. The maintenance of those links may incur a lower variation if the metadata database is better normalized.

4.8.3 Metadata reconstruction and documents availability

Figure 4.14 presents the capacity of the system to rebuild the metadata in case of loss of the metadata server. It shows the amount of documents for which enough metadata has been scraped to be served through the proxy. This result shows how Recast behaves in 2 different STeP configurations, each with 2 different replication factors (none or 3-replication). If all blocks are replicated, we can expect to be able to serve all documents before reading from all the storage nodes as depicted by the full-points lines. In contrast, if none of the blocks are

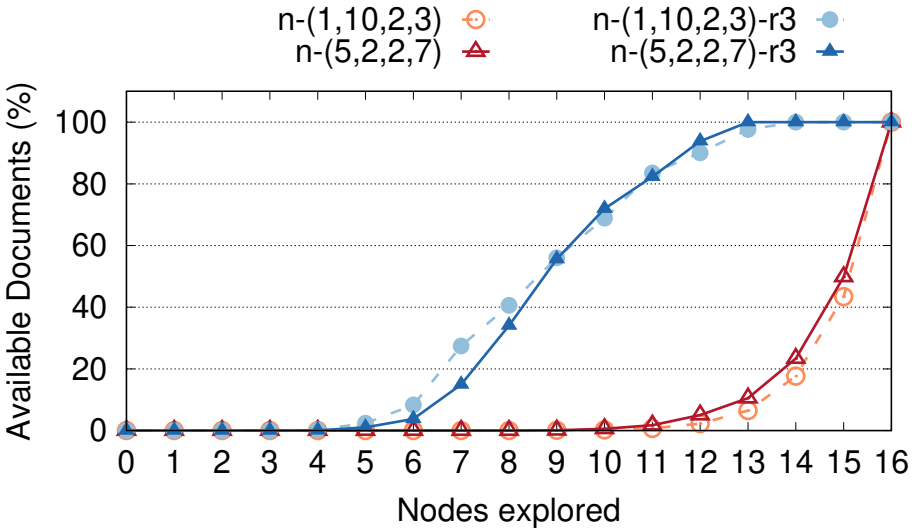


Figure 4.14: Percentage of documents in the archive that can be served as metadata is rebuilt by crawling through the storage nodes.

replicated, a large number of the storage nodes needs to be crawled through until we gather enough knowledge about the entire system. When running an instance of RECAST configured with `nu`-entanglement and replication management enabled, the number of replicated blocks is constant as the archive grows. Following this principle, the number of nodes that need to be explored in order to rebuild a functional archive increases. With respect to the archive history, we can expect the metadata reconstruction process to be fast at first and grow slower over time depending on the number of replicated blocks.

4.8.4 Repair bandwidth costs

Figure 4.15 shows the amount of data in transit through the system in order to repair 1, 2 or 3 missing blocks in RECAST in configuration `nu`-$(1, 10, 2, 3)$. In this instance, RECAST can recover 2 blocks solely relying on Reed-Solomon capabilities ($e = 2$) and 3 exploiting the recursive reconstruction. Performing RS reconstruction of 1 or 2 blocks, 11 blocks of the same document must be read as we use a RS(14,11) to compute the 3 parities. Repairing beyond 2 blocks requires reading from more than 1 document, hence an increase in the number of blocks received for the repair (twice as many). When we have to fix 3 erasures, we repair one of the failed blocks by decoding a document using it as pointer. We are then back to the case where we can rely on the reconstruction capabilities of the RS and recover the last two blocks from the original document itself.

4.9 Discussion

Despite the extensive evaluation of RECAST presented in Section 4.8, some practical issues remain unaddressed bys RECAST's original design. In this section, we go through the possible extensions and considerations that would help with RECAST's deployment.

Figure 4.15: Bandwidth to repair a 64kB document originally stored in an instance of RECAST hosting a n-$(1, 10, 2, 3)$ archive.

4.9.1 Metadata service as a single point of failure

Our prototype only uses entanglement to protect archival data, but not metadata. A powerful censor targeting a document in our system can thus attack its metadata, which is usually significantly smaller and less resilient than the document itself, at a much lower cost. Protecting metadata itself is a challenging task, but since it is much smaller than the data, typically by several orders of magnitude, it is a suitable candidate for massive replication. Most databases, such as Redis, offer clustering and replication features. These schemes are usually good enough to withstand accidental failures. However, to provide proper metadata protection against a powerful attacker, we would need to replicate and ensure consistency over a great number of nodes. As an alternative to database replication, we proposed METABLOCK [141], a blockchain-based metadata protection system.

METABLOCK is built on top of the Ethereum blockhain [288] and stores metadata in smart contracts [255]. It plugs into RECAST by way of message queues and event-based scripts. This design lowers the performance impact on RECAST that only needs to forward metadata to the message queue as new information is stored or modified. Metadata pushed into the queue and stored in RECAST's metadata service remains in a pending state until it is stored by the blockchain at which point it is considered valid. This pending state can then be leveraged at pointer selection time to ensure that new documents are entangled with trusted blocks. For faster processing of the metadata, the consensus protocol is changed from proof-of-work to proof-of-authority [210]. The proof-of-authority consensus protocol implies that the nodes are authenticated and typically deployed in private blockchain.

While storing metadata in a blockchain might seem appealing to benefit from out-of-the-box replication and integrity checking, we found shortcomings in terms of latency, maintenance and storage space [141]. First, time to validate metadata is in the order of seconds which is unsuitable for write-intensive systems. Second, migration of the metadata schema, represented in the smart contract, cannot be done in place and requires a complete re-write of existing metadata. Finally, as the blockchain nodes remains idle, without insertion of new metadata, the ledger size keeps on growing at constant rate.

In conclusion, while securing metadata from passive threats at a reasonable cost is an achievable goal, defending against powerful attackers remains an open problem. METABLOCK

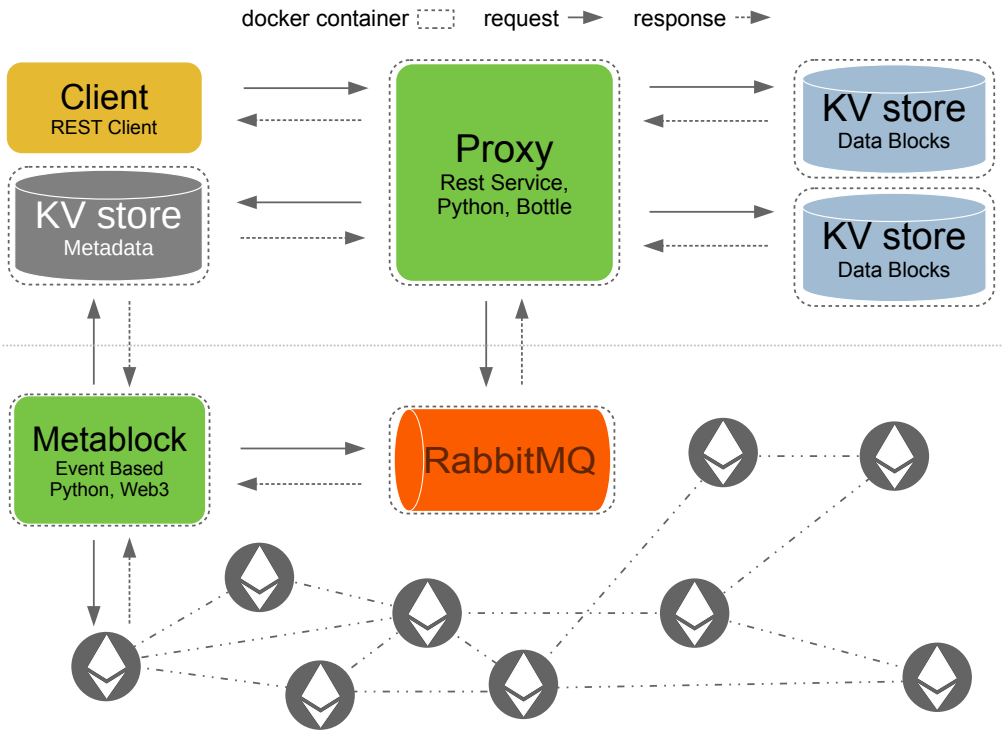


Figure 4.16: Integration of MetaBlock with Recast

may not be sufficient on its own to ensure metadata safety, but a combination of Recast's reconstruction capabilities, database replication and offline backups may be enough to ensure recovery from attacks.

4.9.2 GDPR compliance

Recast is a long-term archival system implementing STeP for better data protection. By design, it does not allow the modification or deletion of documents once they are stored. This design choice conflicts with recent EU legislation such as the European General Data Protection Regulation (GDPR) [277] and especially the rights to erasure and modification. Under such a framework, the data controller, or operator of the system, must be able to remove personal data from the system upon requests from the data subjects. An obligation that may seem difficult to comply considering Recast's design.

With its immutable nature, Recast falls in the same category as blockchain technologies whose compatibility with GDPR has been and is still actively studied [87, 119, 186]. From the literature, we learn that personal data cannot be stored in these systems unless it is highly anonymized. With research on de-anonymization being an active topic and the impossibility to guarantee a high level of anonymity over long periods of time, this type of technology seems inherently limited. Indeed, this reduces the use cases for immutable systems to (*i*) non-personal data and (*ii*) personal data for which the need to ensure integrity and availability exceeds the interest of the data subject to the erasure. In the latter case, data subjects must be explicitly warned before storage and the processing of such data could lead to an infringement of the GDPR.

However, the definition of erasure in article 17 can be left to interpretation [87]. Erasure could simply be understood as marking the data as deleted or removing it from search indexes. But preventing external access to erased data is not enough as the data

controller must guarantee that said data is no longer processed. A possible workaround in an immutable system is to encrypt the personal data before storage and destroy the keys upon erasure requests from the data subjects [148]. But, similarly to the de-anonymization issue, no guarantee can be provided that the encryption scheme used will resist to future results.

In conclusion, the use of systems such as RECAST, that build their strength on immutability, in environments subject to GDPR-like regulation should be preceded by a serious study of the type of data that must be stored. Management of personal data should be avoided when possible even if a laxer interpretation of the text allows for practical solutions to be implemented. While developers should be mindful of the legislative framework they work in as the systems they develop today remain at the mercy of future research results, rulings and court decisions, domain specific knowledge should first inform on potential for successful adoption.

4.10 Summary

Archival systems are designed to support long-term storage of documents. They can leverage the same mechanisms that classical data stores use, notably cryptography and redundancy, to protect against common types of attacks. It should however be resilient against more subtle attacks that would threaten the long-term integrity of the archived data, in particular offering strong protection against attackers that covertly tamper with the data or delete specific documents (*i.e.*, censors).

In this chapter, we presented RECAST, a novel anti-censorship archival system based on random data entanglement [173]. It exploits erasure codes to generate redundant blocks combining content from multiple documents, old and new, in order to protect them from both failures and malicious attacks. As opposed to prior work, RECAST allows efficient recursive repair while requiring censors to do an increasingly large amount of work over a large number of storage nodes as the archive scales, which is a highly desirable property. Entanglement is performed in such a way that documents become more resilient as they stay longer in the system, and the level of interdependency makes it quickly impossible to delete or tamper with a single document without causing collateral damage to a large number of other documents. RECAST uses a hybrid strategy for data entanglement designed to offer fast short-term and strong long-term protection for all the documents in the archive. While our implementation benefits from additional support in order to protect its metadata [141] and should find success with the proper knowledge domain, its model has not only validated the resilience of this strategy via simulations, but also through a complete prototype implementation.

Chapter 5

SafeFS: A Modular Architecture for Secure User-Space File Systems

We have seen so far how we can leverage cloud storage to securely and durably store our data despite the often dreaded loss of control. To mitigate the loss of control over distributed data, we have seen how to protect its privacy using a combination of cryptographic tools (see Chapter 3) and resistance against powerful censorship attempts using data entanglement (see Chapter 4). However, the migration of data from local storage to the cloud still presents at least 2 challenges.

First, the data that we wish to outsource is often read and modified by business critical applications that may not be cloud-aware. However, high-availability requirements, lack of access to the source code or costly expertise could prevent companies from upgrading their software. In this case, settling on the right application-facing interface for a cloud storage system is not only a competitive advantage but a necessity. Second, there is no "one-size-fits-all" solution that would provide the right level of safety or performance for all applications and users, and it is therefore necessary to provide mechanisms that can be tailored to the various deployment scenarios.

In this chapter, we address both challenges by introducing SafeFS, a modular architecture based on software-defined storage principles featuring stackable building blocks that can be combined to construct a secure distributed file system.

5.1 Introduction

The paradigm shift from local storage to the cloud has not changed the expectations of immediate and ubiquitous access to data. As we keep producing more information, which is expected to reach 175 *zettabytes* by 2020 [223], the limit between local and remote storage becomes blurrier.

This paradigm shift raises two important challenges. First, client applications need to handle heterogeneous and incompatible interfaces. Second, client applications have to balance working with data of various sensitivity while maintaining an acceptable level of performance.

The first problem can be partially solved by providing well-known and extensively used abstractions on top of third-party interfaces. One of the most widespread and highest level abstractions offered atop storage systems is the file system. The practicality and ease of use of such abstraction has spurred the development of a plethora of different file system solutions offering a variety of compromises between I/O performance optimization, availability, consistency, resource consumption, security, and privacy guarantees for stored data [268, 244, 289, 22]. Additionally, developers can leverage the FUSE [94] (file system in

user-space) framework to implement a POSIX-like [280] file system on top of a multitude of local and remote storage systems in a fairly straightforward manner. Nevertheless, each file system implementation is different and specifically designed for certain use cases or workloads. Choosing which implementations to use and combining them in order to take advantage of their respective strong features is far from trivial [300].

The second problem, securely outsourcing sensitive data, has also been the subject of intensive work. There are several file system implementations providing privacy preserving mechanisms [289, 107, 82, 175]. However, similarly to storage systems, a single approach does not address the specific privacy needs of every application or system. Some require higher levels of data privacy while others target performance at the price of lower privacy guarantees. Furthermore, these approaches lack a clear separation between privacy preserving mechanisms and the file system implementation itself. This prevents an easy combination of different privacy preserving mechanisms with other file system properties (*e.g.*, caching, compression, replication, etc).

In this chapter we tackle both challenges simultaneously. Inspired by software-defined storage (SDS) design principles [265, 7], we introduce SafeFS, a novel approach to secure user-space stackable file systems. We advance the state of the art in two important ways by providing **two-dimensional modularity** and **privacy à la carte**.

First, the SafeFS two-dimensional modular design allows easy implementations of specialized storage layers for security, replication, coding, compression and deduplication, while at the same time allowing each layer to be individually configurable through plug-in software drivers. SafeFS layers can then be stacked in any desired order to address different application needs. The design of SafeFS avoids usual pitfalls such as the need for global knowledge across layers. For instance, for size-modifying layer implementations (*e.g.*, encryption with padding, compression, deduplication), SafeFS does not require a cross-layer metadata manager to receive, process, or redirect requests across layers [297].

Second, SafeFS design allows us to easily combine any FUSE-based file system implementation with several cryptographic techniques and, at the same time, to leverage both centralized [247, 228] and distributed storage back-ends [22, 180]. For example, it is easy to integrate an existing FUSE-based file system with secret sharing on top of distributed storage back-ends using SafeFS simply by adapting the system APIs. To the best of our knowledge, SafeFS is the first user-space file system to offer this level of flexibility.

To show the practicality and effectiveness of our approach, we implemented a full prototype of SafeFS that, similarly to other proposals [20, 180, 145, 272], resorts to the FUSE framework. With thorough experimental evaluation, we compare several unique configurations of SafeFS, each combining different privacy-preserving techniques and cryptographic primitives. We evaluate the performance by resorting to state-of-the-art benchmarks and show that SafeFS is within the performance range of state-of-the-art FUSE-based file systems while providing higher security guarantees. SafeFS is open-source[1] and available to ease the reproducibility of our evaluation. Our evaluation shows that SafeFS is within the performance range of state-of-the-art FUSE-based file systems while providing higher security guarantees.

The remainder of this chapter is organized as follows. Section 5.2 illustrates the design goals of SafeFS, while Section 5.3 details its architecture. The implementation details are given in Section 5.4. The configurations put under test are presented in Section 5.5. Section 5.6 presents our extensive evaluation of the SafeFS prototype, before concluding with the road ahead.

[1] `https://github.com/safecloud-project/SafeFS`

5.2 Design goals

SafeFS is a framework for flexible, modular and extensible file system implementations built atop FUSE. Its design allows to stack independent layers, each with their own characteristics, optimizations, etc. These layers can then be integrated with existing FUSE-based file systems as well as restacked in different order. Each stacking configuration leads to file systems with different traits, suitable to different applications and workloads. Keeping this in mind, the four pillars of our design are:

- *Effectiveness.* SafeFS aims to reduce the cost of implementing new file systems by focusing on self-contained, stackable, and reusable file system layers.
- *Compatibility.* SafeFS allows us to integrate and embed existing FUSE-based file systems as individual layers.
- *Flexibility.* SafeFS can be configured to fit the stack of layers to the applications requirements.
- *User-friendliness.* From a client application perspective, SafeFS is transparent and usable as any other FUSE file system.

5.3 Architecture

Figure 5.1 depicts the architecture of SafeFS. The system exposes a POSIX-compliant file system interface to the client applications. Similar to other FUSE systems, all file system related operations (*e.g.*, `open`, `read`, `write`, `seek`, `flush`, `close`, `mkdir`, etc.) are intercepted by the Linux FUSE kernel module and forwarded to SafeFS by the FUSE user-space library. Each operation is then processed by a stack of layers, each with a specific task. The combined result of these tasks represents a file system implementation.

We identify two types of SafeFS layers serving different purposes. Upon receiving a request, *processing* layers manipulate or transform file data and/or metadata and forward the request to the next layers. Conversely, *storage* layers persist file data and metadata in designated storage back-ends, such as local disks, network storage, or cloud-based storage services. All layers expose an interface identical to the one provided by the FUSE library API, which allows them to be stacked in any order. Requests are then orderly passed through all the layers such that each layer only receives requests from the layer immediately on top of it and only issues requests to the layer immediately below. Layers in the bottom level must be of storage type, in order to provide a functional and persistent file system.

This stacking flexibility is key to efficiently reuse layer implementations and adapt to different workloads. For example, using compression before replicating data across several storage back-ends may be acceptable for archival-like workloads. In such settings, decompressing data before reading it does not represent a performance impairment. On the other hand, for high-throughput workloads it is more convenient to only apply compression on a subset of the replicated back-ends. This subset will ensure that data is stored in a space-efficient fashion and is replicated to tolerate catastrophic failures, while the other subset will ensure that stored data is uncompressed and readily available. In these scenarios, one storage stack would use a compression layer before a replication layer, while a second storage stack would put the compression layer after the replication and only for a subset of storage backends. Layers must be stacked wisely and not all combinations are efficient. An obviously bad design choice would be to stack a randomized privacy layer (*e.g.*, standard AES cipher) before a compression layer (*e.g.*, gzip): by doing so, the efficiency of the compression layer would be

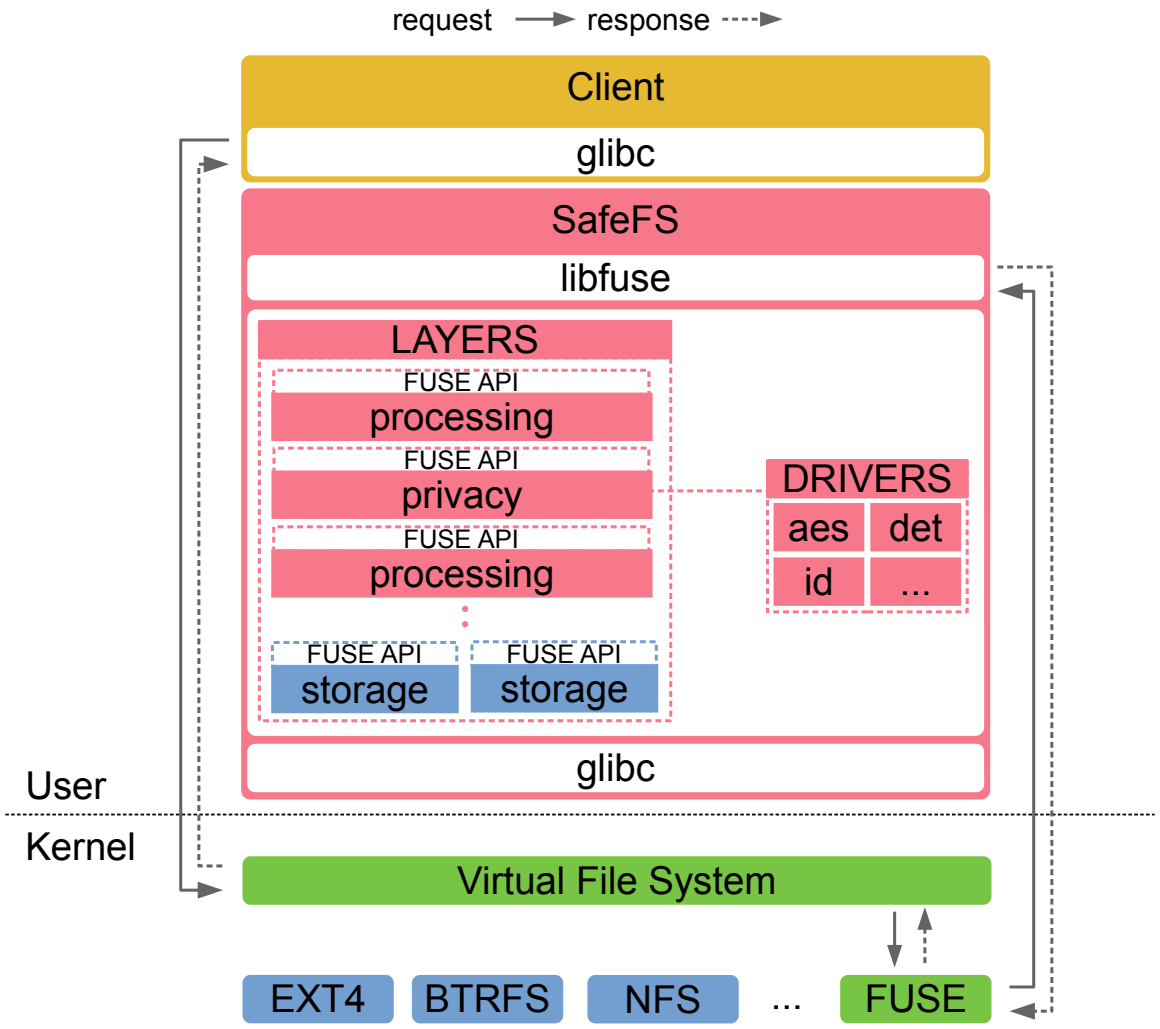


Figure 5.1: Architecture of SafeFS.

highly affected since information produced by the above layer (the randomized encryption) should be indistinguishable from random content. While such malformed scenarios can indeed happen, we believe that *with great power comes great responsibility* and operators who deploy the system need to take care of the appropriate layer ordering. Finally, the SafeFS architecture allows us to embed distributed layers. This is depicted in Figure 5.1, where the lowest layer of the stack is a split of two that can be leveraged for replication or any other redundancy purposes. These final layers can then branch off to a local or remote file system to ensure durability and availability in case of failure of the final storage medium. SafeFS supports redirection of operations toward multiple layers, while at the same time maintaining these layers agnostic from the layer above that transmits the requests.

5.3.1 A day in the life of a write

To illustrate the I/O flow inside a SafeFS stack, we consider a `write` operation issued by the client application to the virtual file system (`read` operations are handled similarly). Each request made to the virtual file system is handled by the FUSE kernel module (Figure 5.2-❶) that immediately forwards is to the user-space library (Figure 5.2-❷). At this point the request reaches the topmost layer of the stack (Figure 5.2-❸), called *Layer 0*. After processing the request according to its specific implementation, each layer issues a write operation to the

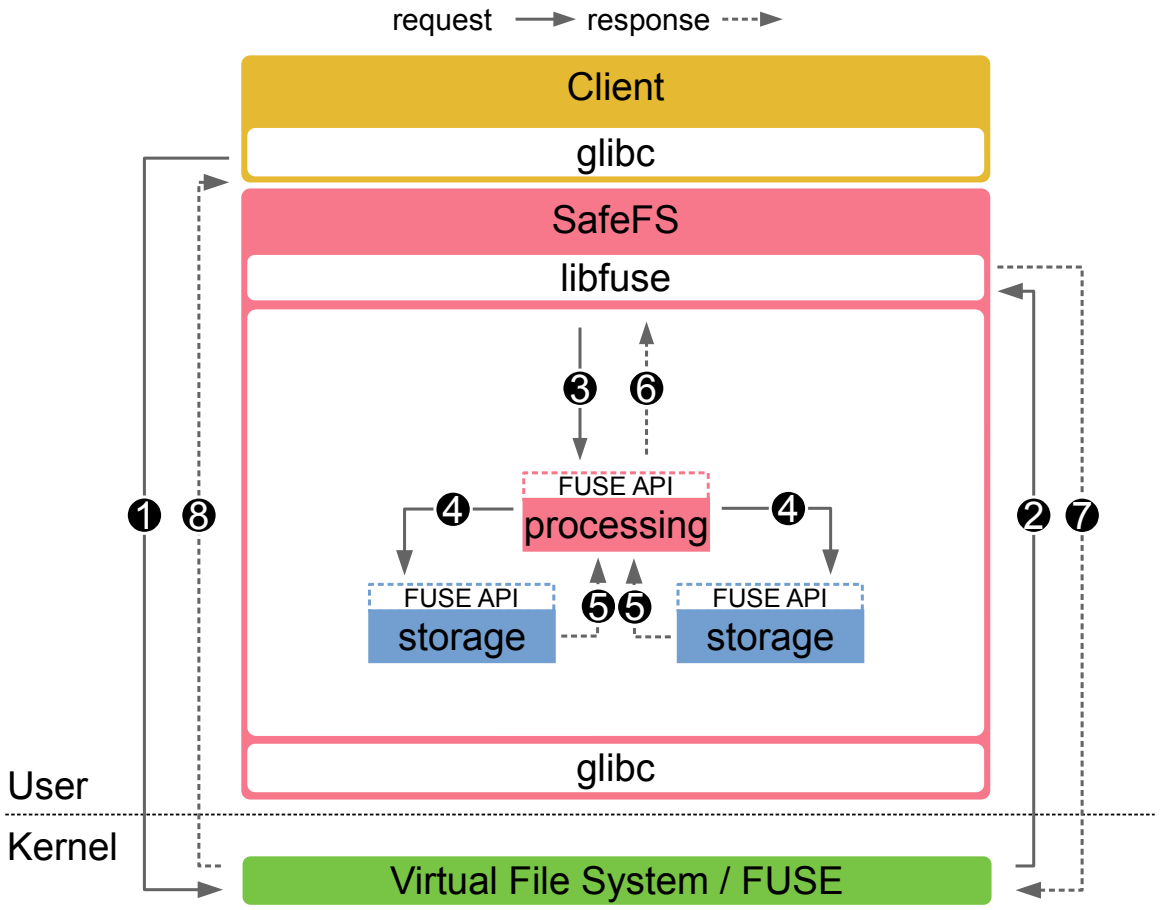


Figure 5.2: Flow of a `write` request.

following layer. For example, a privacy-preserving layer responsible for ciphering data will take the input data, cipher it according to its configuration, and emit a new `write` operation with the encrypted data to the underlying layer. This process is repeated, according to each layer implementation, until the operation reaches a *storage* layer, where the data is persisted into a storage medium (Figure 5.2-❹). The reply request stating whether the operation was successfully executed or not takes the reverse path and is propagated first to *Layer 0* (Figure 5.2-❺), and eventually backward up to the application (Figure 5.2-❽).

When using distributed layers (*e.g.*, with replication), `write` operations are issued to multiple sublayers or storage back-ends. These distributed layers can break some of the assumptions made by the applications. For instance, `rename` and `sync` operations must be atomic. To ensure correct semantics of the operations, a distributed layer should contain a synchronization mechanism that ensures that an operation is only committed if successful in every backend. Otherwise, the operation fails and the file state must not be changed. A possible solution would be a block cache that stores blocks before any operation is applied.

We have discussed so far how layers modify data from read and write operations. The behavior for layers that modify the attributes and permissions of files and folders is similar. For instance, a layer providing access control to files shared among several users will add this behavior to the specific FUSE calls that read and modify the files. This design paves the way for layer reuse and for interesting stacking configurations. Individual layers do not need to implement the totality of the FUSE API: if a layer only needs to manipulate files, it only needs to wrap the FUSE operations that operate over files. FUSE operations over folders can be ignored and passed directly to the next layer without any additional processing.

Layers can support the full FUSE API or a restricted subset, and this allows for a highly focused layer development cycle.

5.3.2 Layer integration

Besides the standard FUSE API, each SafeFS layer implements two additional functions. First, the *init* function initializes metadata, loads configurations, and specifies the following layer(s) in the specific SafeFS stack. Second, the *clean* function frees the resources allocated by the layer during its runtime.

The integration of existing FUSE-based implementations in the form of a SafeFS layer is straightforward. Once *init* and *clean* are implemented, a developer simply needs to link its code against the SafeFS library instead of the default FUSE. Additionally, for a layer to be stacked, delegation calls are required to forward requests to the layers below or above. The order in which layers are stacked is flexible and is declared via a configuration file.

Finally, SafeFS supports layers that modify the size of data being processed (*e.g.*, compression, padded encryption) without requiring any global index or cross-layer metadata. This is an advantage over previous work [289], further discussed with concrete examples in Section 5.4.

5.3.3 Driver mechanism

Some of the privacy-preserving layers must be configured with respect to the specific performance and security requirements of the application. However, these configurations do not change the execution flow of the messages. From an architectural perspective, using a DES cipher or an AES cipher is strictly equivalent.

With this observation in mind, we further improved the SafeFS modularity by introducing the notion of *driver*. Each layer can load a number of drivers by respecting a minimal SafeFS driver API. Such API may change according to the layer specialization and characteristics, as further discussed in the next section. Drivers are loaded according to a configuration file at file system's mount time. Moreover, it is possible to change a driver without recompiling the file system, to re-implement layers, or to load new layers. Naturally, this is possible provided that the new configuration does not break compatibility with the previous one. For instance, introducing different cryptographic techniques will prevent the file system from reading previous data.

Consider the architecture depicted in Figure 5.1, with a privacy-preserving layer having two drivers, one for symmetric encryption via AES and another for asymmetric encryption with RSA. The driver API of the layer consists of two basic operations: `encode` and `decode`. In this scenario, the cryptographic algorithms are wrapped behind the two operations. When a `write` request is intercepted, SafeFS calls `encode` on the loaded driver and the specific cryptographic algorithm is executed. Similarly, when a `read` operation is intercepted, the corresponding `decode` function is called to decipher the data. In order to change the driver it is sufficient to unmount the file system, modify the configuration file, and remount the SafeFS partition.

The driver mechanism can be exploited by layers with diverse goals, such as those targeting compression, replication, or caching. In the next section we discuss the current implementation of SafeFS and break down its driver mechanism in further details.

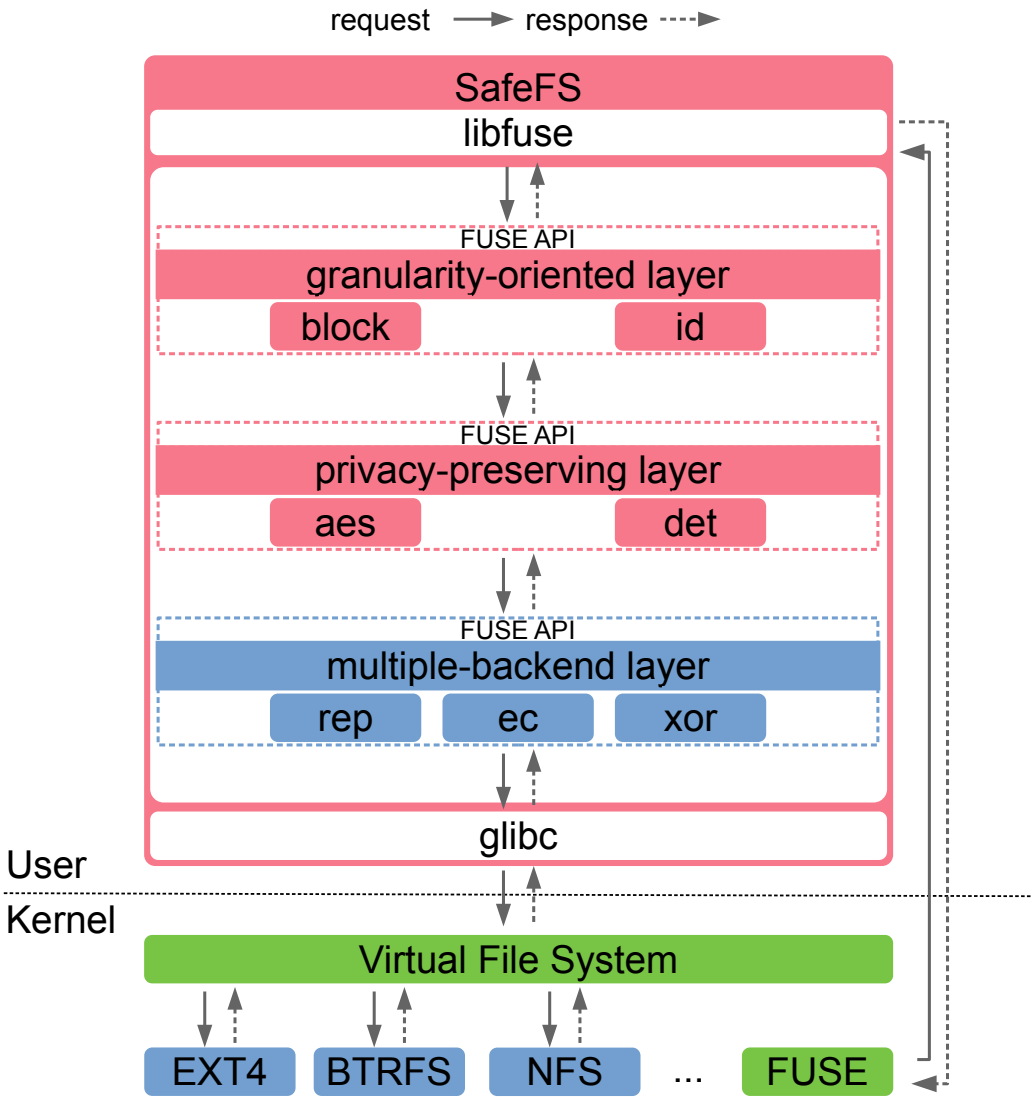


Figure 5.3: SafeFS: chain of layers and available drivers.

5.4 Implementation details

We have implemented a complete prototype of SafeFS in the C programming language. Currently, it consists of less than 4,200 lines of code (LOC), including headers and the modules to parse and load the configuration files. Configuration files are used to describe what layers and drivers are used, their initialization parameters, and their stacking order. The code required to implement a layer is also remarkably concise. For example, our cryptography-oriented layer only consists of 580 LOC. SafeFS requires a Linux kernel that natively supports FUSE (v2.6.14). To evaluate the benefits and drawbacks of different layering combinations, we implemented three unique SafeFS layers, as depicted in Figure 5.3. These layers are respectively concerned with data size normalization (*granularity-oriented*), enforcing data privacy policies (*privacy-preserving*) and data persistence (*multiple backend*). Since they are used to evaluate SafeFS, we detail them in the remainder of this section.

5.4.1 Granularity-oriented layer

It is important to be able to stack layers that operate on data at different granularity levels, *e.g.*, with different block sizes. For example, one might need to stack a layer that reports dynamic sizes for file write and read operations over a layer that only works with fixed-sized blocks ([289, 82]).

As a more concrete example, the FUSE user-space library reports file write and read operations with dynamic sizes. Yet, many cryptographic algorithms only work with fixed-size block granularity and hence require a *block size translation* mechanism. Such translation is provided by the *granularity-oriented* layer. This layer opens the way to exploit block-based encryption, instead of whole-file encryption, which is more efficient for many workloads where requests are made only for small portions of the files. For instance, if only 3 bytes of a file are being read and the block size is 4KB, then only 4KB must be deciphered while a whole-file approach could require the entire file to be deciphered only to recover those same 3 bytes of data.

In more details, the translation layer creates a logical abstraction on top of the actual data size being read, written, or modified. This is achieved by processing data write and read requests from the upper layer and manipulating the offsets and data sizes to be read/written from underneath layers. The manipulation of the offsets and sizes is done using two functions: `align_read` and `align_write`. The drivers of the layer must implement both function calls to define distinct logical views for read (for `align_read`) and write (for `align_write`) operations. Operations on directories or file attributes are redirected to adjacent layers pristine.

Our prototype implements two drivers for the translation layer: a *block* and an *identity* driver. The *block* driver creates a logical block representation for both file write and read requests, which will be used transparently by the following layers. This block abstraction is fundamental for layers whose processing or storage techniques rely on block-based requests (*e.g.*, block-based encryption, de-duplication, etc.) [145]. Block size is configured on driver initialization. On the other hand, the *identity* driver does not change the offset or the buffer size of the bytes read or written. We use this driver as a baseline to understand the overhead of our block-oriented approach and the layer itself.

5.4.2 Privacy-preserving layer

The goal of this layer is to protect sensitive information in a transparent way for applications and other layers of SafeFS. As explained in Section 5.3.3, file data being written or read is intercepted by the layer and then ciphered (`encode`) or deciphered (`decode`). We implemented three drivers: standard AES encryption (*AES*), deterministic encryption (*Det*), and *Identity*.

The *AES* driver leverages the OpenSSL's own AES-128 block cipher in *CBC* mode [88]. Both key and initialization vector (`IV`) size of the AES cipher are parameters defined during the initialization of the driver. Our design follows a block-based approach for ciphering and deciphering data. Hence, the *block* driver of the *granularity-oriented* layer is crucial to transparently ensure that each `encode` and `decode` call issued by the *AES* driver receives the totality of the bytes for a given block. Each block has a random IV associated, generated in the *encode* function, that is stored as extra padding to the cipher text.[2]

[2]The `IV` is important for decoding the cipher-text and returning the original plain-text but keeping it public after encryption does not impact the security of the system [196].

The *Det* driver protects the plaintext with a block-based deterministic cipher (AES-128). This cipher does not need a new random `IV` for each encoded block and is hence faster than randomized encryption. Despite compression algorithms being more efficient in plain-text, this driver helps detect data redundancy over cipher-text, otherwise impossible to find with a standard randomized encryption scheme.

Both drivers resort to padding (16 bytes from the AES padding plus 16 bytes for storing the `IV`). For example, a 4KB block requires 4,128 bytes of storage. Manipulating block sizes must be done consistently across file system operations. Every size modifying layer must keep track of the real file size and the modified file size so no assumption is broken for the upper and lower layers. For instance, if a layer adds padding data, it only reports the original file size without the extra padding to the previous stack layer.

Finally, we implemented an *Identity* driver, which does not modify the content of intercepted file operations and is used as an evaluation baseline, similarly to the *granularity-oriented* Identity driver. We note that drivers for other encryption schemes (*e.g.*, DES, Blowfish, or RSA) could be implemented similarly.

5.4.3 Multiple back-end layer

The storage layers directly deal with persisting data into storage back-ends. In practice, these back-ends are mapped to unique storage resources available on the nodes (machines) where SafeFS is deployed. The number of storage back-ends is a system parameter. They may correspond to local hard drives or remote storage back-ends, such as NFS servers accessible via local mount points. The drivers for this layer follow the same implementation pattern described previously, namely via `encode` and `decode` functions. The encode method, upon a write request, is responsible for generating a set of data pieces to be written in each storage back-end from the original request. The decode implementation must be able to recover the original data from a set of data pieces retrieved from the storage back-ends.

Our evaluation considers three drivers: replication (*Rep*), one-time pad (*XOR*), and erasure coding (*Erasure*). The *Rep* driver fully replicates the content and attributes of files and folders to all the specified storage back-ends. Thus, if one of the back-ends fails, data is still recoverable from the others. The *XOR* driver uses one-time pad XOR to protect files. The driver creates a secure block (secret) by applying the one-time pad to a file block and a random generated block. This operation can be applied multiple times using the previous new secret as input, thus generating multiple secrets. The original block can be discarded and the secrets safely stored across several storage back-ends [196]. The content of the original files can only be reconstructed by accessing the corresponding parts stored across the distinct storage back-ends. Finally, the *Erasure* driver uses erasure codes such as Reed-Solomon [285] to provide reliability similar to replication but at a lower storage overhead. This driver increases data redundancy by generating additional parity blocks from the original data. Therefore, only a subset of the blocks is sufficient to reconstruct the original data. The generated blocks are stored on distinct back-ends, thus tolerating some unavailability without any data loss. As erasure codes modify the size of data being processed, this driver requires to a metadata index that tracks the offsets and sizes of stored blocks on a per-file basis. The index keeps the size-changing of erasure-codes contained within the layer.

Table 5.1: The different SafeFS stacks deployed in the evaluation. Stacks are divided in three distinct groups: Baseline, Privacy, Redundancy. The table header holds the three SafeFS layers. Below each layer we show the respective drivers. For each stack, we indicate the active drivers (the $\surd$ symbol). Layers without any active drivers are not used in the stack. The indices for Multiple-Backend drivers indicate the number of storage backends used to write data.

		Granularity		**Privacy**			**Multiple-Backend**		
Groups	Stack	Block	Id	AES	Det	Id	Simple	XOR	Erasure
Baseline	FUSE	✗	✗	✗	✗	✗	✓,1	✗	✗
	Identity	✗	✓	✗	✗	✓	✓,1	✗	✗
Privacy	*AES*	✓	✗	✓	✗	✗	✓,1	✗	✗
	Det	✓	✗	✗	✓	✗	✓,1	✗	✗
	XOR	✗	✗	✗	✗	✗	✗	✓,3	✗
Redundancy	*Rep*	✗	✗	✗	✗	✗	✓,3	✗	✗
	Erasure	✓	✗	✗	✗	✗	✗	✗	✓,3

5.4.4 Layer stacks

The above layers can be configured and stacked to form different setups. Each setup offers trade-offs between security, performance, and reliability. The simplest SafeFS deployable stack consists of the *multiple back-end* layer plus the *Rep* driver with a replication factor of 1 (file operations issued to a single location). This configuration offers the same guarantees of a typical FUSE loopback file system.

Increasing the complexity of the layer stack leads to richer functionalities. By increasing the replication factor and the number of storage back-ends for the simplest stack, we obtain a file system that tolerates disk failure and file corruption. Similarly, replacing the *Rep* with the *Erasure* driver, one may achieve a file system with increased robustness and reduced storage overhead. However, erasure coding techniques only work in block-oriented settings thus requiring the addition of the *granularity-oriented* layer to the stack.

When data privacy guarantees are required, one simply needs to include the *privacy-aware* layer into the stack.[3] Note that when *AES* and *Erasure* are combined, the file system stack only requires a single *block* oriented layer. This layer provides a logical block view for requests passed to the *privacy-aware* layer. These requests are then automatically passed as blocks to the *multiple back-end* layer.

Using the *XOR* driver provides an interesting privacy-aware solution, since trust is split on several storage domains. This driver exploits a bitwise technique not dependent on previous bytes to protect information, thus it does not require a block-based view as *privacy-aware* drivers.

5.5 Configurations

Our evaluation (Section 5.6) comprises a total of 7 different stack configurations (Table 5.1). Each one of these exposes different performance tradeoffs and allows us to evaluate the different features of SafeFS. The chosen stacks are divided in three groups: baseline, privacy, and redundancy.

[3]Due to the block-based nature of the *Rep* and *Erasure* drivers, the *granularity-oriented* is also required.

The first group of configurations, as the name implies, serve as baseline file system implementations where there is no data processing. The FUSE stack is a file system loopback deployment without any SafeFS code. It simply writes the content of the virtual file system into a local directory. The *identity* stack is an actual SafeFS stack where every layer uses the *identity* driver. It corresponds to a pass-through stack where the storage layer mimics the loopback behavior. These two stacks provide means to evaluate SafeFS framework overhead and individual layer overhead.

The privacy group is used to evaluate SafeFS's modularity and measure the tradeoffs between performance and privacy guarantees of different the different techniques. In our experiments we used three distinct techniques. The *AES* stack and *Det* stacks correspond respectively to a standard and a deterministic encryption mechanism. The *AES* stack is expected to be less efficient than *Det* as it generates a different IV for each block. However, *Det*'s performance gains also yield weaker security guarantees. The third stack, named *XOR*, considers a different trust model where no single storage location is trusted with the totality of the ciphered data. Data is stored across distinct storage back-ends in such a way that unless an attacker gains access simultaneously to all back-ends, it is impossible to recover any sensitive information about the original plaintext.

Finally, the two remaining stacks deal with data redundancy. The *Rep* stack fully replicates files into three storage back-ends. In our configuration, two out of three back-ends can fail, while still allowing the applications to recover the original data. The *Erasure* stack serves the same purpose but relies on erasure codes for redundancy instead of traditional replication. Data is split into 3 fragments (2 data + 1 parity) over 3 back-ends for a reduced storage overhead of 50%, with respect to replication. This erasure coding configuration supports the complete failure of one of the back-ends. These stacks provide an overview of the costs of two different redundancy mechanisms.

5.6 Evaluation

This section presents our comparative evaluation of the SafeFS prototype. First, we present the third-party file systems used for comparison in Section 5.6.1. Second, we list the setup and the benchmarking tools used for the evaluation in Section 5.6.2. Finally, we present the results in the following order: a performance evaluation using a database benchmarking tool in Section 5.6.3, a second performance evaluation using workload simulations in Section 5.6.4 and a breakdown of the time spent in each layer of the different stacks when running database benchmarks in Section 5.6.5.

5.6.1 Third-party file systems

Since our SafeFS prototype implementation focuses on privacy preserving features, we deploy and run our suite of benchmarks against 4 open-source file systems with encryption capabilities. We include 3 user-space file systems: CryFS [54], Metfs [175], and LessFS [145] as well as eCryptfs [107], a kernel-space file system available in the Linux mainstream kernel. We selected those for their wide availability, adoption by the community, and the different security tradeoffs they offer. Their characteristics are summarized in Table 2.3 (Section 2.4.4) but repeated here for context.

CryFS [54] (v0.9.6) is a FUSE-based encrypting file system that ensures data privacy and protects file sizes, metadata, and directory structure. It uses AES-GCM for encryption

and is designed to integrate with cloud storage providers such as Dropbox. For evaluation purposes, we configure CryFS to store files in a local partition.

EncFS [82] (v1.7.4) is a cross-platform file system also built on top of FUSE. This system has no notion of partitions or volumes. It encrypts every file stored on a given mounting point using AES with a 192-bit key. A checksum is stored with each file block to detect corruption or modification of stored data. In the default configuration, also used in our benchmarks, a single `IV` is used for every file, which increases encryption performance but decreases security.

MetFS [175] (v1.1.0) uses a stream cipher (RC4) for encryption and stores its data as a single blob file when unmounted.

LessFS [145] (v1.7.0) supports deduplication, compression (via QuickLZ used in our experiments) and encryption (BlowFish), all enabled by default.

Finally, eCryptfs [107] (v1.0.4) includes advanced key management and policy features. All the required metadata are stored in the file headers. Similar to SafeFS, it encrypts the content of a mounted folder with a predefined encryption key using AES-128.

5.6.2 Evaluation settings

The experiments run on virtual machines (VM) with 4 cores and 4GB of RAM. The KVM hypervisor exposes the physical CPU to the guest VM with the `host-passthrough` option [266]. The VMs run on top of hard disk drives (HDD) and leverage the `virtio` module for better I/O performance. We deployed each file system implementation inside a Docker (v1.12.3) container with data volumes to bypass Docker's AUFS [69] and hit near-native performance.

We conducted our experimental evaluation using two commonly used benchmarking programs: `db_bench` and `filebench`. The `db_bench` benchmark is included in `LevelDB`, an embeddable key-value store [146], as well as its successors such as `RocksDB` [226]. This benchmark runs a set of predefined operations on a local key-value store installed in a local directory. It reports performance metrics for each operation on the database. The `filebench` [86] tool is a fully-fledged framework to benchmark file systems. It emulates both real-world and custom workloads configured using an workload modeling language (WML). Its suite of tests includes simple micro-benchmarks as well as richer workloads that emulate mail- or web-server scenarios. We leverage and expand this suite throughout our experiments.

We ran several workloads for each considered file system (4 third-party file systems and 7 SafeFS stacks). The results have been grouped according to the workloads. First, we present the results of using `db_bench`, then `filebench` and, finally, we describe the results of running latency analysis for SafeFS layers.

5.6.3 Database benchmarking throughput

We first present the results obtained with `db_bench`. We pick 7 workloads, each executing 1M read and write operations on `LevelDB`, which stores its data on the selected file systems. The *fill100K* test (identified by ①) writes 100K entries in random order. Similarly, the entries are written in random order (*fillrandom*, ②) or sequentially (*fillseq*, ③). The *overwrite* (④) test completes the write-oriented test suite by overwriting entries in random order. For read-oriented tests, we considered 3 cases: *readrandom* (⑤), to randomly read entries from the databases, *readreverse* (⑥) to read entries sequentially in reverse order, and finally *readseq* (⑦) to read entries in sequential order.

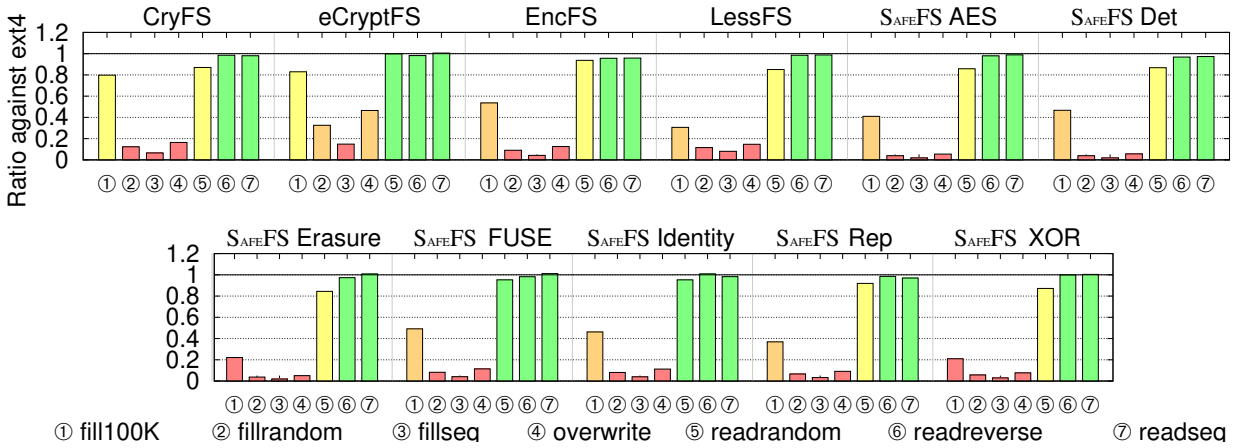


Figure 5.4: Relative performance of `db_bench` workloads against `native`.

Figure 5.4 presents the relative results of each system against the same tests executed over a `native` file system (ext4 in our deployment). We use colors to indicate individual performance against native: red (below 25%), orange (up to 75%), yellow (up to 95%), and green (>= 95%). Please note that MetFS results were discarded due to a locking issue which prevented the database to initialize correctly.

We observe that all systems show worse performance for write-specific workloads (①–④) while performing in yellow class or better for read-oriented workloads (⑤, ⑥, and ⑦). The results are heavily affected by the number of entries in the database (*fill100K* ① vs *fillrandom* ②). As the size of the data to encrypt grows, the performance worsens. For instance, the SafeFS `XOR` configuration (the one with the worst performance) drops from 21% to 0.5%. The same observation applies for CryFS (the system with best performance) that drops from 79.78% to 12.33%.

The results for the *fillseq* ③ workload require a closer look as they have the worst performance in every file system evaluated. Since `db_bench` is evaluating the throughput of `LevelDB` which is storing its data on the evaluated file-systems, it is necessary to understand an important property of `LevelDB`. The database is optimized for write operations, which results for *fillseq*, in high throughputs on native file systems contrasting with the selected file systems, where the throughput is significantly lower. As a matter of example, comparing throughputs for *fillseq* vs *fillrandom* on `native` (17.4 MB/s vs. 7.74 MB/s) and CryFS (1.14 MB/s vs. 0.94 MB/s) shows how much of the initial gains provided by `LevelDB` are lost.

While the processing of data heavily impacts the writing workloads, reading operations (⑤, ⑥ and ⑦) are relatively unaffected. The results for *readrandom* range from 85.06% with LessFS up to 99.81% for eCryptFS. Moreover, in experiments that switch the reading offset, the results are even better. In more details, the results never drop below 95.67% (*readreverse* on EncFS) independently of whether the reading is done from the beginning or the tail of the file.

Overall, the different SafeFS stacks perform similarly for the different database operations. The privacy stacks (see Table 5.1) perform comparably to the other file systems on most operations. Only the *fill100K* test shows significant differences, in particular against CryFS and eCryptFS. As expected, the deterministic driver provides a better performance (46.69%) against *AES* (40.96%) and *XOR* (20.98%). The redundancy stacks perform similarly. The erasure driver is slightly less efficient (22.11% of the native performance) due to the additional coding processing.

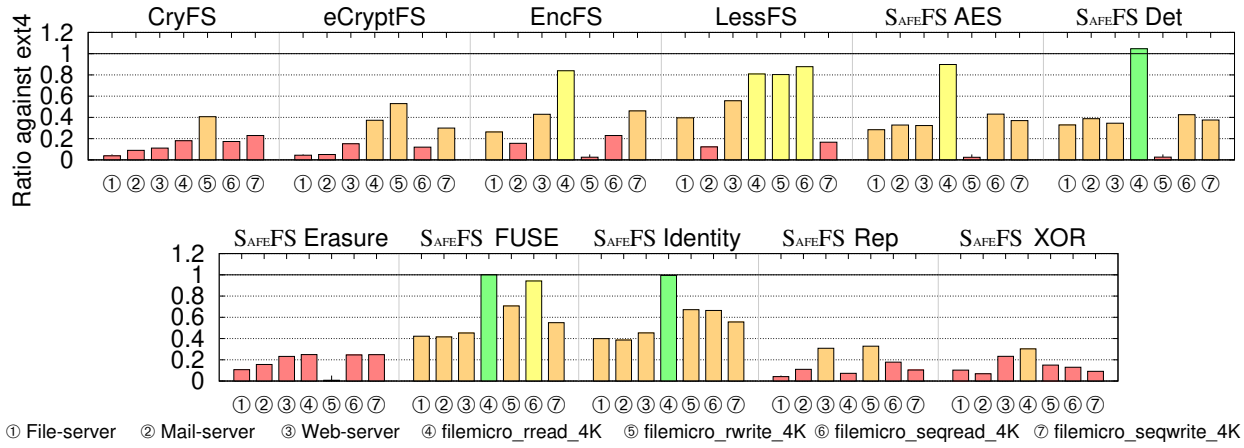


Figure 5.5: Relative performance of `filebench` workloads against `native`.

5.6.4 Workload simulation throughput

Next, we look at the relative performance of various workloads from `filebench`. Figure 5.5 shows our results. We re-use the same color scheme used for `db_bench`: red (below 25%), orange (up to 75%), yellow (up to 95%), and green (>= 95%). The seven workloads, executed over the different file systems and configurations, can be separated in two sets: application emulations (file-server ①, mail-server ②, web-server ③) and micro-workloads (④ to ⑦). This classification also introduces 3 major differences.

First, the application benchmarks last for 15 minutes, while the micro-workloads terminate once a defined amount of data is read or written. Second, the number of threads interacting with the system is respectively set to 50, 16, and 100 for the three workloads ①, ②, and ③, while micro-workloads are single-threaded. Third, the focus of micro-workloads is to study the behavior of a single type of operation while the application emulations usually run a mix of read and write operations. We discard LessFS results from this part of the evaluation as the system exhibits inconsistent behavior (*i.e.*, unpredictable initialization and run time ranging from minutes to hours) leading to timeouts before completion.

In the micro-workloads (④–⑦), we observe the performance of the tested solutions in simple scenarios. Reading workloads (④ and ⑥) are most affected by the reading order. Surprisingly, our implementation performs better than the baseline with `Det` at 104.68% on random reads. These observations contrast with the results obtained for sequential reads where the best performing configuration in this case is SafeFS `fuse` (94.24%). On the writing side, micro-workloads ⑤ and ⑦ also display different results. For sequential writes (⑦), SafeFS `Identity` tops the results at 55.56% of the native performance. On the other end of the scale, SafeFS `XOR` stalls at 9.14%. The situation does however get a little better when writing randomly: `XOR` then jumps to 14.98%. An improvement that contrasts with the case of erasure coding (that has to read all the existing data back before encoding again) where the performance dramatically drops from 29.93% to 0.7% when switching from sequential to random writes.

On the application workloads side, the mixed nature of the operations gives better insights on the performance of the different systems and configurations. The systems that make use of classical cryptographic techniques consistently experience performance hurdles. As the number of write operations diminishes, from ① to ③, the performance impact decreases accordingly. Another important factor is the use of weaker yet faster schemes (such as the deprecated stream cipher RC4 for MetFS or re-using IVs for SafeFS `Det`). As expected,

those provide better results in all cases. Indeed, MetFS reaches 39.56% for ① and 55.65% for ③, and `Det` tops at 38.74% for ②. Resorting to more secure solutions can still offer good results with SafeFS `AES` (①: 28.40%, ②: 32.32%, and ③: 32.79%) but the need for integrated access control management should not be neglected for an actual deployment. Figure 5.5 presents the remaining SafeFS stacks for redundancy. These also exhibit signs of performance degradation as the data processing intensifies.

Beyond the specifics of the data processing in each layer, the performance is also affected by the number of layers stacked in a configuration. As evidence, we observe that the `Identity` stack has a small but noticeable decrease of performance when compared with other `FUSE` stacks.

Overall, the privacy-preserving stacks of SafeFS with a single back-end have a better performance than the other available systems across the workloads. Only MetFS is capable of providing a better performance in some workloads. This benchmark suggests that user-space solutions, such as those easily implementable via SafeFS, perform competitively against kernel-based file systems.

5.6.5 Runtime breakdown across layers

In addition to using `db_bench` to study the performance degradation introduced by SafeFS, we use some of its small benchmarks (*fill100K*, *fillrandom*, *fillseq*, *overwrite*, *readrandom*, *readreverse*, and *readseq*) to measure the time spent in the different layers as the system deals with read and write operations. To do so, SafeFS records the latency of both operations in every layer loaded in the stack. The results obtained are presented in Figure 5.6 where we show the percentage of time spent reading (top) and writing (bottom) for each workload and configuration. We recall that for a read or write operation issued by `db_bench`, the call will be relayed to the layers in the order described in Table 5.1 (*i.e.*, `granularity`, `privacy` and finally `backend`). We note that for all these benchmarks, the initialization phase is part of the record and that the time stacks show the sum of a layer's inherent overhead and the time spent in the underlying layers.

As expected, the time spent in each layer varies according to the tasks performed by the layers. The 3 most CPU-intensive stacks (`AES`, `Det` and `Erasure`) concentrate their load over different layers: `privacy` for the first two and `backend` for the last one. Indeed, more time is spent in `backend`, the lowest layer, in non-privacy-preserving configurations. Another noticeable point is the increase in time spent reading data back for `Erasure` in the `backend` layer (11.03% for *fill100K*, 21.19% for *fillrandom*, and 21.53% for *fillseq*) compared to the decrease for `Rep` (respectively 8.02%, 1.93%, and 0.05%) and `XOR` (4.03%, 2.59%, and 0.05%) stacks. This is again explained by the fact that `Erasure` needs to decode an entire file before re-encoding it and does so for every new write. However, a simple improvement would be to encode at the block level, to avoid the reading of entire files, which should both increase the throughput of the stack and decrease the read amplification. And with the layer independence provided by SafeFS, this implementation change could easily be integrated with little to no impact to the other layers.

Overall, this breakdown shows us that reading and writing to the final medium remains the most expensive part of each operation. We also see that the use of an intermediary layer that essentially acts as a no-op (see `Fuse` and `Identity`) weighs significantly on total runtime. But while no realistic implementation would run with such unnecessary layers, we see this breakdown as a useful tool to identify software bloat and wasteful implementations.

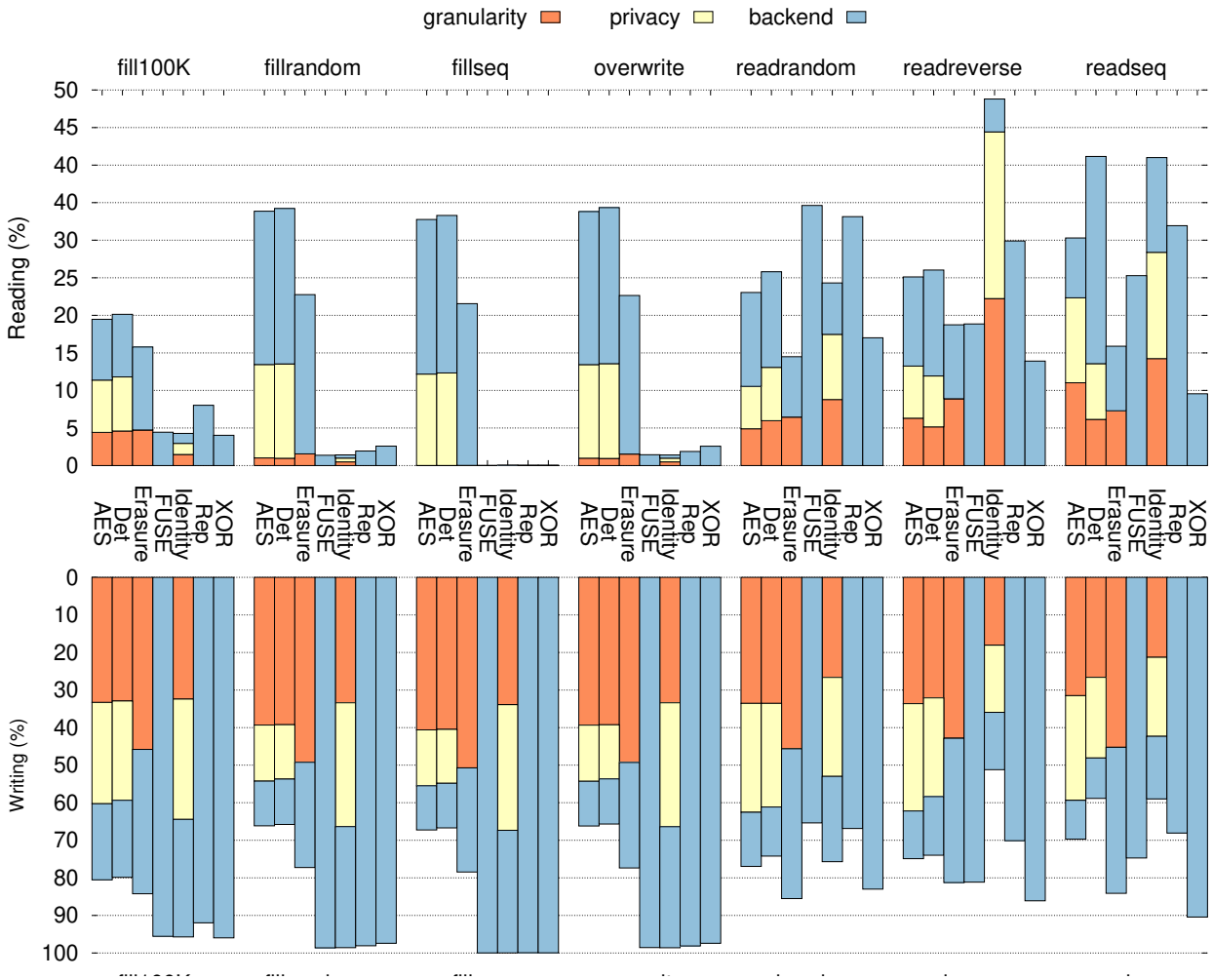


Figure 5.6: Breakdown of runtime spent between the different layers of SafeFS configurations. The total sum for stacks a given configuration, for reading (top) and writing (bottom), always equals 100%.

5.7 Discussion

The design, implementation, and evaluation of file systems is a complex problem as applications using them have different requirements in terms of fault-tolerance, data privacy, or pure performance. In this section, we discuss improvements and possible avenues for the future development of SafeFS.

5.7.1 Access control and key management

While the SafeFS stacks used in our evaluation are fully functional and the results presented are representative of secure file systems, two aspects of such systems were left out: access control and key management. However, CryFS, eCryptfs and other systems in the literature offer solutions for these problems.

Access control is a common feature that ensures that actions performed by user on the file system are authorized. In the case of a file system, these authorized actions range from reading to modifying and/or deleting objects such as files or directories. A file system can check whether the user running a program over certain paths has the right to interact

with it. Restrictions can be applied using tools such as SElinux [4] but a fine-grained solution can easily get complex to understand and tedious to maintain. FUSE, however, partly solves this problem out of the box with a drastic design choice. By default, FUSE ensures that the content of a mount point is only accessible to the user that started the file system going as far as excluding the root user. This restriction can be eased up when starting the file system to allow all users, root or none other to access the mount point. While this level of configuration might seem too coarse for systems with a large number of users that share the same resources, we believe it is sufficient for a user hosting their own private files. Key management, on the other hand, remains a challenging problem. For convenience, the different SafeFS stacks used in our experiment derive the key from a plain-text passphrase in a configuration file. The passphrase in this configuration file could also be hashed and salted for better protection. But that would not prevent an attacker from extracting the file and discovering the original passphrase through sheer brute force. Another approach is to prompt the user to input the passphrase at start-up time. This is the way CryFS, EncFS, LessFS and MetFS work. Unfortunately, this method adds a lot more friction to a user's routine as they must remember, or keep track of, the passphrase and input it on every restart. A preferable approach is to tie the file system's decryption to the user login. This is the method chosen by Ubuntu's integration of eCryptfs. As the user logs in, the passphrase is combined with an encryption key from its kernel-space daemon *ecryptfsd* to decrypt the home directory. Regardless of the method chosen, a practical must provide a balanced compromise between user-friendliness and security.

5.7.2 Data processing in trusted execution environments

We have evaluated and discussed protection schemes for data and encryption keys. The methods presented offer good solutions against unauthorized users trying to access the data when mounted or at rest. However, some threat models also consider processes with higher privileges or the operating system (OS) itself as threats. With this in mind, encryption keys or decrypted files living in our FUSE file systems' memory remain at risk. To prevent this kind of issue, Trusted Execution Environments (TEE) offer tooling to run computation and store data securely from processes with higher privileges. In practice, TEEs enable developers to split their applications between a trusted and an untrusted part. The trusted part runs inside a secure world, where computations and secrets cannot be accessed or tampered with by other processes. One example of such TEEs are Intel Software Guard eXtensions (SGX) [53]. The trusted part of the application runs in a so-called enclave where secrets can also be kept during the lifetime of the application. The enclave code can be signed by an authority and an attestation mechanism allows operators to verify that the code running inside is indeed the one that was signed. As the secure code is loaded in the enclave, it represents an ideal location to encrypt, decrypt and handle sensitive data. Finally, when the file system is unmounted or the process needs to be terminated, the sensitive data can be encrypted using SGX's sealing capabilities. The sealing mechanism enables developers to encrypt data using a key derived from keys in the processor's fuse array that are unique to the enclave or signer of the enclave and cannot be accessed by the OS. In order to measure the cost of running this kind of computation inside an enclave, we built Sgx-Fs [33].

While dissociated from SafeFS, as it works as an independent FUSE file system, Sgx-Fs makes it easier to measure the impact of crossing the enclave boundary by keeping all data in memory. We built 3 variants of the system: Ram-Fs where all the data resides

[4] `https://github.com/SELinuxProject/selinux`

unencrypted in memory, SGXRAM-FS where data is first encrypted by the enclave and kept sealed in memory and SGX-FS where all the data is kept unencrypted in the enclave. On shutdown, SGX-FS seals all in-memory data and store it on disk. RAM-FS and SGXRAM-FS simply write the data from memory to disk without any further transformation. On the next restart, the data can be loaded from disk, and in the case of SGXRAM-FS and SGX-FS, with the insurance that it has not been tampered with by processes with higher privileges.

From this work, we learned two things. First, the startup time of the file system is mostly impacted by the cost of enclave initialization rather than the unsealing or reading of data (from 100 ms for RAM-FS to 2.4 s for SGXRAM-FS amd SGX-FS). While this was tested by loading data within the memory limit of the enclave (below 128 MB), there is very little difference in file system setup whether the data to load is 32, 64 or 96 MB. Second, the cost of writing a file through/to the enclave multiplies the round trip time by 3x/10x respectively for SGXRAM-FS and SGX-FS. This is partly due to the way FUSE tends to make systematic `getattr` calls before safely opening, reading or writing to a file. This call requires the lookup of information that is found in unprotected memory for SGXRAM-FS and inside the enclave for SGX-FS. While the overhead comes with better safety and security guarantees, this performance penalty may not be acceptable for all workloads.

But beyond our observations, it is important to remain cautious when integrating SGX to an existing solution. First, numerous attacks on SGX have managed to run malicious code within the enclave [239] or leak enclave data through side-channels [152, 270, 238, 235, 181, 271, 236, 234]. A part of these vulnerabilities leverage speculative execution of enclave code, and as a consequence, patches issued by Intel often come with significant performance penalty. Second, it should be noted that due to the limited amount of memory in SGX enclaves, managing what data or metadata should stay unencrypted within these boundaries requires clever memory and caching management on top of running a consistent file system. Porting SGX-FS to other TEEs such as ARM Trustzone [11] or AMD Secure Encrypted Virtualization [129], that are not subject to performance altering patches and rigid memory bounds, could offer interesting results and avenues for future development.

5.7.3 End-to-end encryption

In order to fully protect our data from the prying eyes of unauthorized users and processes with higher privileges, it still may not sufficient to run our computation inside an enclave. Indeed, file systems in user-space rely on the FUSE infrastructure to bridge client requests to their daemon. Traveling from the systems calls enabled by their local flavor of the libc through the kernel to the Virtual File System (VFS) gateway, the FUSE kernel module, the FUSE user-space library and finally reaching the user-space daemon, client requests are seen in the clear by all these components. Unfortunately, modifying and loading kernel parts is often beyond the capabilities offered to regular users.

At network-scale, extensions of SAFEFS such as TRUSTFS [83], already ensure that work on encrypted data can be delegated to remote machines in distributed storage settings by leveraging Intel SGX. However, this does not prevent the OS and privileged processes from snooping on the data in transit between components on the local machine.

A potential solution to solve this issue would be to have the client and the user-space daemon load a modified version of the `libc` and `libfuse`. By encrypting all information in the call forwarded, apart from the mountpoint required by VFS and FUSE kernel module, we can limit information leakage. This could be achieved by having the two libraries instantiate secure enclaves, perform a key exchange and encrypt the content of their requests and

responses from that point on. This method is similar to the one used in Obliviate [6] to limit the leak of information by combining Intel SGX and oblivious RAM. Another benefit of this approach is that it does not require re-compiling the file system or the client application. By using `LD_PRELOAD` to load our modified libraries with these applications, we ensure that they communicate over secure channels with our file system.

5.8 Summary

In this chapter we presented SafeFS, a modular FUSE-based architecture that allows system operators to simply stack building blocks (*layers*), each with a specific functionality implemented by plug-and-play *drivers*. This modular and flexible design allows extending layers with novel algorithms in a straightforward fashion, as well as reuse of existing FUSE-based implementations.

We compared several SafeFS stacking configurations against industry-battled alternatives and demonstrated the trade-offs for each of them. Our extensive evaluation based on real-world benchmarks hopefully shed some light on the current practice of deploying custom file systems and will facilitate future choices for practitioners and researchers.

We envision to extend SafeFS along three main directions. First, we plan to smooth the efforts to integrate any existing FUSE file system as a SafeFS layer, for example by exploring Linux's LD_PRELOAD mechanism, thus avoiding any recompilation step. Second, we envision a context- and workload-aware approach to choose the best stack according to each application's requirements (*e.g.*, storage efficiency, resource consumption, reliability, security) leveraging SDS control plane techniques that enforce performance, security, and other policies across the storage vertical plane stack [265]. Finally, we intend to improve the driver mechanism to allow for dynamic, on-the-fly reconfiguration.

Chapter 6

MinervaFS: User-Space File System with Generalised Deduplication

So far we have described a world where a single cloud storage provider cannot be trusted on their own. In Chapter 3, we made use of different encryption, integrity checking and information dispersal schemes in order to preserve the confidentiality, integrity and availability our data before sending it to the independent providers. In Chapter 4, we implemented a scheme that enhances the durability of archived data over a set cloud-storage providers by means of random data entanglement. And finally in Chapter 5, we set up a framework that facilitates the composition feature-rich storage systems using the most common storage abstraction used by applications: file systems.

While the propositions mentioned above contribute to the overall safety and security of users' data in the cloud, they also tend to increase the storage overhead and thus its storage cost. In addition, all these contributions make it seem like innovation can only be expected from the client-side of distributed storage. However, considering the amount of data their users generate and the quality of service these users expect, cloud storage providers must strive to offer good service at a manageable operational cost. One of the option they can pursue to lower the cost of running such services is a reduction in storage overhead. Common data reduction techniques such as compression and deduplication have long been studied (see Section 2.5) but we propose explore a little further.

In this chapter, we implement a recently introduced data reduction scheme known as Generalised Deduplication and embed it in a user-space file system to test its effectiveness and practicality.

6.1 Introduction

In the last decade, we have witnessed a stark increase in data generated by various sources (*e.g.*, social networks, Internet of Things) and persistently stored on public clouds with recent estimations accounting for 175 ZB of data by 2025 [223]. A large portion of this increase is due to the rise of connected Internet of Things (IoT) devices. Notably, sensors of different nature, *e.g.*, weather stations, biomedical or smart grid meters, produce continuous streams of data for collectors in charge of aggregating and possibly storing them for subsequent data analytics. A typical example of such scenario is Amazon AWS IoT analytics [15]. Moving and aggregating data from IoT devices on cloud-based storage systems puts enormous pressure on the storage backends of cloud providers, as they must deal with the need of continuously expanding their storage infrastructure to follow such ever-growing demand for storage needs. There exist different techniques, such as compression [295], erasure coding [117,

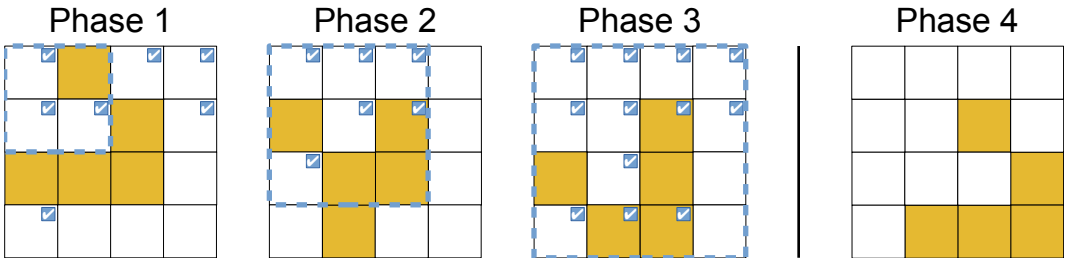


Figure 6.1: Effect of chunk size on deduplication potential on the transitions of a glider in the game of life. Identifying the redundant blocks in phases 1 to 3 against phase 4. Dashed boxes indicate chunk perimeter (from 2×2 to 4×4) and ☑ indicate common bits with the right-most glider.

203], deduplication [214, 306] or a combination of these [170], to reduce the storage footprint and, coincidentally, to also increase the dependability of the storage system. In particular, data deduplication [292, 199] processes and identifies duplicates across multiple data chunks for different granularities [133], *i.e.*, files or blocks, for fixed or dynamically [193] chosen chunk sizes. A critical issue with deduplication is that near identical chunks of data (even differing by a single bit) will be identified as completely different (*e.g.*, each chunk hashes to a different value), depending on the chosen chunk size. Consider for instance the simple case of 16-bit images (Figure 6.1), *e.g.*, four *game of life*'s gliders [51] representing ordinal numbers. For different sizes of the chunk (in dashed lines, from 2×2 up to 4×4) used to check common blocks of data, each of three left-most gliders share a different number of bits with the right-most one, hence providing more opportunities for deduplicating those bits. State-of-the-art deduplication implementations (*e.g.*, BtrFS's `bedup` [18] or ZFS [302]) would identify the three left-images as completely different from the right-most one, and hence without any opportunity to deduplicate. Furthermore, as additional content is added to a given dataset, the more opportunities for deduplication are created. To illustrate this, we analysed the logs from a real-world HDFS deployment [307, 293]. We report the evolution of the serial correlation coefficient [159] (SER) while new entries are added to the log. As seen in Figure 6.2, after an initial spike, the value of SER converges around 0.585 (when random content would converge towards zero).

Recently, *Generalised Deduplication* (GD) [275] has been proposed as a novel deduplication technique that systematically identifies data similarities (without the need to compute additional hashes in contrast to Rabin fingerprints, *e.g.*, [71, 140]) using transformation functions that map each chunk to a basis (common to a large number of potential chunks) and a deviation (indicating exactly the change with respect to the basis). Then, GD deduplicates each basis. GD is theoretically optimal for compression as sufficient chunks are fed into the system [275] and was shown to converge to this optimal point multiple orders of magnitude faster than classical deduplication, *i.e.*, gains are seen with far less data.

This chapter presents MinervaFS, a user-space file system that demonstrates how GD can be efficiently used as building block for large-scale storage systems exposing a classical POSIX-based file system interface. We implement MinervaFS using the FUSE framework [149], as it allows the easy prototyping of efficient file systems in user-space [272]. We evaluate our full MinervaFS prototype against synthetic workloads and real-world datasets.

MinervaFS is a non-trivial extension of the concepts proposed in [275]. Previous work provided a construction to evaluate the ideal compression performance of GD, *e.g.*, convergence

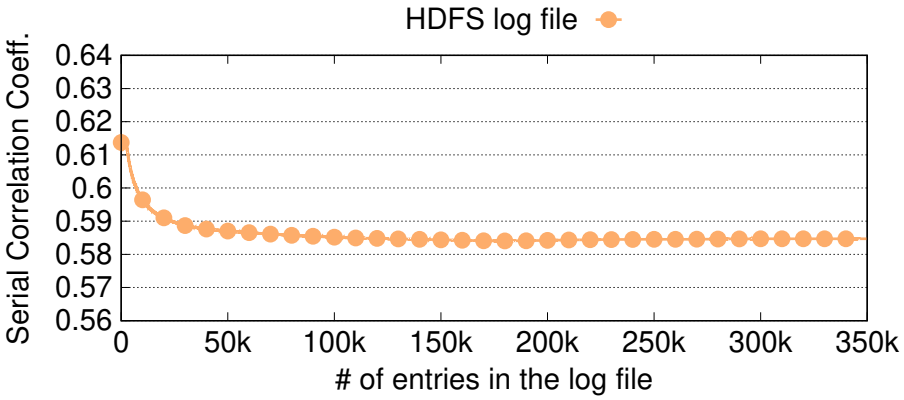


Figure 6.2: Evolution of the serial correlation coefficient in the first 350,000 entries of the HDFS dataset [307, 293].

to source entropy, convergence speed, with a simple data source. This construction is good for illustrating the benefits of the transformation function. However, it does not allow for random access of data chunks or edits of previously written data, which are key features in a file system. In addition, [275] did not consider implementation challenges, *e.g.*, processing speed, metadata, durability, or real datasets.

Figure 6.3 shows the storage savings that MinervaFS can bring when compared to other common space-saving techniques, such as Gzip compression, or state-of-the-art deduplication implementations (such as the one used in ZFS [302]). In particular, it shows its benefits on a different set of input datasets: (1) HDFS logs [307, 293], (2) real-world flat images from the RAISE repository [61], (3) a pansharpened version of RAISE datasets (a technique used for instance by Google Maps to increase the quality of images by merging images of different resolutions), and (4) Virtual Machine (VM) images from various Linux distributions in VDI format [195]. As we observe, MinervaFS offers good storage savings (*e.g.*, 19.29% and 63.53% savings for HDFS logs and VDI, respectively) for both small and large data sets, while ZFS provides no or very limited savings.

In developing MinervaFS, we faced (and proposed solutions for) the following problems. First, we go beyond the state-of-the error correcting and coding libraries needed to transform chunks into basis and deviation pairs in order to reach our target throughputs. We achieve this by exploiting AVX instructions available in Intel processors, to speedup processing to 8 GB/s, effectively being limited by our underlying hardware (see Section 6.4 for details). More importantly, we maintain the speedup over a large span of chunk sizes, from 512 bytes to 128 KB. Second, deduplication libraries are typically limited by costly read/write maintenance operations on the registry, as well important memory overhead. For instance, ZFS requires a 5 GB registry alone for the storage of 1 TB. As we explain in Section 6.4, we leverage ext4 to store the registry on disk, significantly reducing MinervaFS's memory footprint and unbinding it from a linear growth with the data stored. We also show that the size of the on-disk registry is an order of magnitude smaller than ZFS' for the same chunk size. Finally, we show that our first MinervaFS implementation is 16 % faster for read-heavy workloads and 42 % slower in write-heavy ones when compared to ZFS. We believe this to represent a significant improvement over ZFS, reducing the memory requirements while providing higher compression gains and maintaining high-performance in read/write operations.

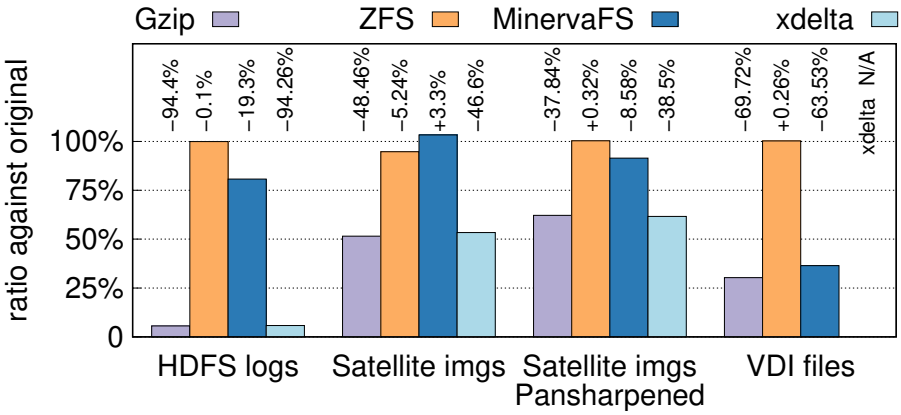


Figure 6.3: Compression ratios of Gzip, ZFS and MinervaFS on different types of datasets.

The rest of this chapter is organized as follows. Section 6.2 describes GD and its theoretical foundations. We describe the architecture of our system in Section 6.3 and in-depth implementation details in Section 6.4. MinervaFS' performance under multiple workloads and datasets is analysed in Section 6.5. Section 6.6 discusses the results and potential future work. Finallu, Section 6.7 closes the chapter with a summary of this work.

6.2 Generalised Deduplication

Data deduplication is a common technique used to reduce the redundancy in data before storage or transmission. In storage systems, the focus is mostly on the storage overhead of data at rest. Classical deduplication accomplishes this by splitting the data into chunks, uniquely identifying the content of each piece and ensuring redundant chunks are only stored once by checking them against a registry. If an identical chunk is not found in the registry, the new chunk is stored in the system and can later be referenced when new files enter the system. Conversely, if a similar chunk exists, a new reference is added to the existing chunk in the registry and the new chunk is not stored. While simple in principle, classical deduplication implementations can vary in many ways (*e.g.*, fixed or variable size for chunks, choice of hashing schemes to identify chunks, registry implementation). However, a common issue to classical deduplication, and in most implementations, is that if two chunks of data deviate by a single bit they will be considered as completely different. Generalised Deduplication (GD) solves this issue by identifying similarities between chunks further than exact matches.

In practice, GD achieves this by using transformation functions (T) to create a *Many-to-One mapping* from different chunks to a common basis. A transformation function $T(c)$ is a partial deterministic function which maps a chunk (c) to a basis (b) and a deviation (d) pair where d describe how c deviates from b. As the system grows, new pairs are added to the sets of bases $\mathcal{B}$ and deviations $\mathcal{D}$. As multiple chunks map to the same bases, we apply classical deduplication to the bases but store the deviations as they are. To identify duplicates we create fingerprints over the set of bases ($\mathcal{B}$) and maintain a registry $\mathcal{R}$ keeping track of the list of all files stored in the system, the chunks they are made of, the list of all bases in $\mathcal{B}$ and their fingerprint.

As shown in [275], GD achieves optimal compression asymptotically. By separating each chunk in two parts, one to be deduplicated, and one to be stored directly, GD allows deduplication of similar chunks. GD has been evaluated in IoT applications [273], with a Reed-Solomon code as the transformation function, and in [274], with a permute-and-truncate

transformation function. For the latter, the authors showed that the proposed scheme could reach similar or even superior compression ratios than state-of-the-art compression techniques, *e.g.*, Gzip, bzip2, 7z, while providing random access to data without the need to decompress more than a few samples of the IoT stream. This is done for small chunk sizes (≤ 18 bytes), compatible with compression of individual or groups of files, but not necessarily with the characteristics expected from a file system. MinervaFS focuses on chunk sizes orders of magnitude larger and uses a simpler and faster transformation function based on Hamming codes [284]. Thus, while the compression ratio of MinervaFS is not expected to achieve the same gains as standard compression techniques, as we show later in the evaluation, it outperforms state-of-the-art file systems using deduplication.

6.2.1 A brief description of Generalised Deduplication

Generalised Deduplication (GD) starts by splitting a file f_i into n-bit long *chunks*. These chunks are referred to as $c_{(i,j)}$, where j indicates their sequence number in f_i, and form a structure describing f_i named C_i. From this point, GD applies its transformation function $T(c)$ on all $c_{(i,j)} \in C_i$.

The function $T(\cdot)$ transforms a chunk $c_{(i,j)}$ into a pair $(b_{(i,j)}, d_{(i,j)})$, where $b_{(i,j)}$ is a *basis* and $d_{(i,j)}$ is a *deviation* describing how $c_{(i,j)}$ deviates from $b_{(i,j)}$. From an n-bit long chunk, $T(\cdot)$ produces a k-bit long basis and an m bit long deviation, such that $n = k + m$.

We define $\mathcal{P}_i$ as the set of basis-deviation pairs associated to the transformation of chunks in C_i, *i.e.*, $\mathcal{P}_i = \{(b,d) : T(c) = (b,d), \forall c \in C_i\}$. We also define $\mathcal{B}_i = \{b : T(c) = (b,d), \forall c \in C_i\}$ as the set of bases generated after the transformation and $\mathcal{D}_i = \{d : T(c) = (b,d), \forall c \in C_i\}$ as the set of deviations generated after the transformation. A valid $T(c)$ is deterministic and with an inverse $T^{-1}(\cdot)$, such that $T^{-1}(b,d) = c$. The goal is to create a systematic *Many-to-One mapping* of the bases, *i.e.*, bring together all chunks with the same basis b regardless of their deviations.

GD then performs deduplication on all bases in $\mathcal{B}_i$. We define $\mathcal{B}_{(i-1)} = \cup_{j=1}^{i-1}\mathcal{B}_j$ as the set of all bases in the system prior to the reception of file f_i. That is, if $b_{(i,j)}$ is in $\mathcal{B}_i$ and not in $\mathcal{B}_{(i-1)}$, the system will have to store $b_{(i,j)}$. Otherwise, if there is a copy of $b_{(i,j)}$ in the system used, the system will link to the existing basis like classic deduplication.

A registry $\mathcal{R}_g$ is still needed, but the GD entries will change with respect to those in classic deduplication. Specifically, for each entry to the registry, the algorithm uses a file identifier per file f_i linked to an ordered set of pairs for each chunk containing the identifier or fingerprint to its basis, *i.e.*, $h^b_{(i,j)} = H(b_{(i,j)})$ and the deviation associated to that chunk, $d_{(i,j)}$. We denote this ordered set $\mathcal{I}_i = [(h^b_{(i,1)}, d_{(i,1)}), ..., (h^b_{(i,K_i)}, d_{(i,K_i)})]$. As in classic deduplication, the system keeps a registry of identifiers to bases and the location on persistent media of the stored bases.

6.2.2 Generalised Deduplication of the first file

Finally, we provide a complete step-by-step example on processing a *first* file, using Hamming code $(n,k) = (7,4)$, with the $\mathcal{P}$ basis-deviation and the $\mathcal{R}_g$ registry empty. Figure 6.4 illustrates this process.

The system ingests file f_1, composed of 3 chunks $c_{(1,1)} = 1010010$, $c_{(1,2)} = 1011110$, and $c_{(1,3)} = 1111101$ (Figure 6.4-❶). The next step is the transformation (Figure 6.4-❷): $c_{(1,1)}$ and $c_{(1,2)}$ are mapped to the same basis $b_{(1,1)} = b_{(1,2)} = 1010$ and two different deviations $d_{(1,1)} = 110$ and $d_{(1,2)} = 010$. Chunk $c_{(1,3)}$ is mapped to another basis $b_{(1,3)} = 1101$ and

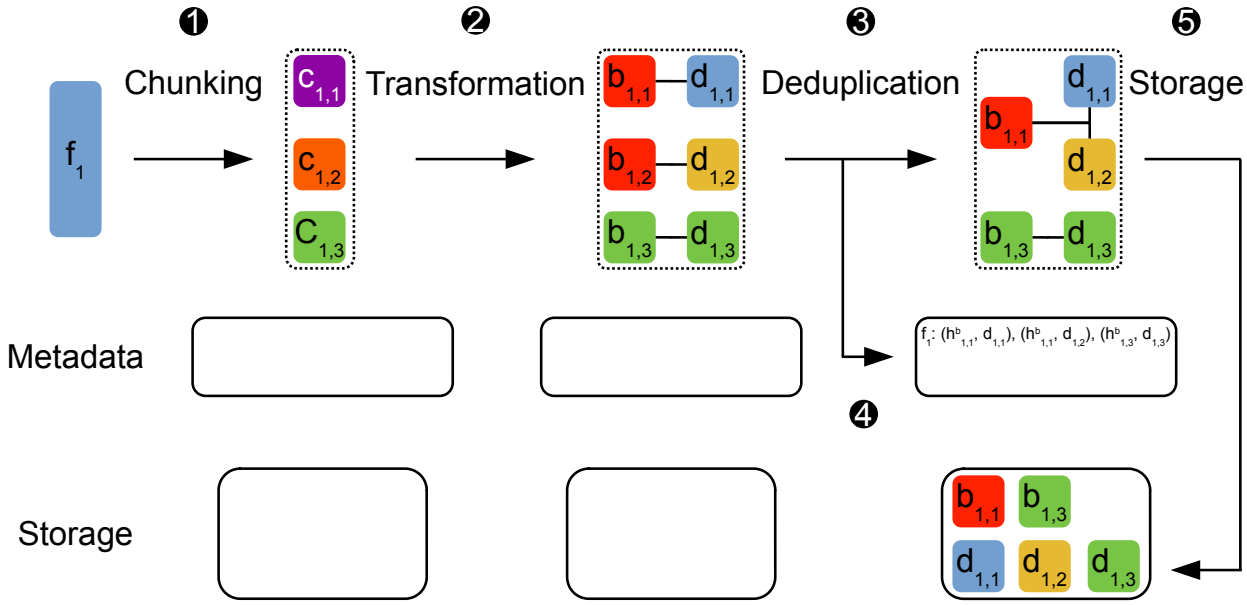


Figure 6.4: Generalised deduplication for the first file.

a deviation $d_{(1,3)} = 010$. Hence, $\mathcal{P}_1 = \{(b_{(1,1)}, d_{(1,1)}), (b_{(1,1)}, d_{(1,2)}), (b_{(1,3)}, d_{(1,3)})\}$ describes f_1. $\mathcal{P}_1$ is passed to the deduplication step (Figure 6.4-❸). For the first pair, $(b_{(1,1)}, d_{(1,1)})$ we know that $\mathcal{B} = \emptyset$, so we generate the hash $h^b_{(1,1)} = H(b_{(1,1)})$ and add it to $\mathcal{B}$. For the next pair, $(b_{(1,2)}, d_{(1,2)})$, $h^b_{(1,2)} = H(b_{(1,1)}) = h^b_{(1,1)}$, which will be found in $\mathcal{B}$ and, thus, the system will not add it again. Finally, $(b_{(1,3)}, d_{(1,3)})$, $h_{(1,3)}$ is not yet in $\mathcal{B}$. Once added we obtain $\mathcal{B} = \{h^b_{(1,1)}, h^b_{(1,3)}\}$. $\mathcal{R}$ is updated (Figure 6.4-❹) with an entry for f_1 to $\mathcal{R} = \{(f_1, \{(h^b_{(1,1)}, d_{(1,1)}), (h^b_{(1,1)}, d_{(1,2)}), (h^b_{(1,3)}, d_{(1,3)})\})\}$. Finally (Figure 6.4-❺), bases $b_{(1,1)}$ and $b_{(1,3)}$ are persisted on disk.

The process is then repeated for each file ingested by the system. In the next section, we look at how to design a system that performs generalised deduplication in the context of a file system.

6.3 Architecture of MinervaFS

MinervaFS is built around four major components laid across three layers. Figure 6.5 presents these layers and their interactions. Looking at the layers from top to bottom, we now describe the responsibilities covered by each one.

Minerva File System layer (MFL). The MFL acts as the interface between the client application and the underlying layers. It exposes the system with a POSIX-like interface by implementing typical file system operations, *e.g.*, `open`, `read`, `write`, `truncate`, `close`, `mkdir`, `flush`. It also orchestrates the exchange of data between the various components.

Generalised deduplication layer (GDL). The GDL provides the *transformation functions* (TF) and *deduplication* components and handles the GD process. The main responsibility of TF is to transform data into basis b and deviation d pairs, and back into the original data by respectively using the T and T^{-1} transformation functions. This includes chunking data before applying T, as well as reassembling chunks after the usage of T^{-1}. The deduplication component handles the fingerprinting of bases, maintaining and updating $\mathcal{R}$ and $\mathcal{B}$ as new chunks enter the system for deduplication purposes.

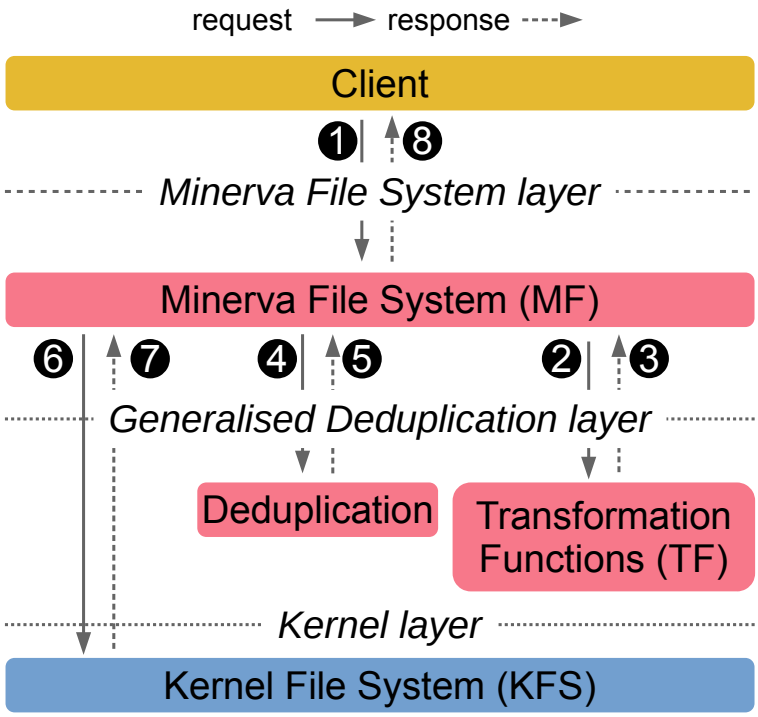


Figure 6.5: MINERVAFS: layered architecture and interaction across the components.

Kernel layer (KL). This layer hosts the underlying *kernel file system* (KFS). The KFS is the file system used as the backend storage. Once data reaches this layer, the KFS can write and read it to/from disk.

6.3.1 Storing and loading data

Next, we describe the execution flow of two key operations, *i.e.*, storing and loading files. Figure 6.5 depicts such flows using white circled numbers and black-filled circled numbers, respectively for writing and reading.

Storing files. The client sends a file path and data to the MF component (Figure 6.5-❶). MF sends the data to TF (Figure 6.5-❷), which chunks it and applies the transformation function on each of the generated chunk. TF then creates a data structure containing the transformation function configuration (T_{config}) as well as the basis and deviation pairs before returning it to MF (Figure 6.5-❸). Note that T_{config} keeps track of the transformation function and the function parameters used to process the file. In our evaluation, we only consider a transformation based on Hamming codes with a single configuration parameter, namely, the m parity bits. As explained previously, this m determines the sizes of chunks, bases, and deviations. Even though, we keep the value of m static during MINERVAFS' runtime, the systematic storage of this information facilitates backward-compatibility and makes it possible to use variable chunk-sizes within the same system.

Next, MF sends the basis and deviation pairs to the deduplication component (Figure 6.5-❹). For each basis and deviation pair, the deduplication component creates a fingerprint of the basis. It also checks if such a fingerprint has been computed in the past, tagging the new ones to be stored in the system. The list of computed fingerprints, bases and deviations are returned to MF (Figure 6.5-❺). For each of such basis, MF writes to the KFS using the computed fingerprint as the path for the basis (Figure 6.5-❻). The KFS returns either a write success or failure message to MF (Figure 6.5-❼). In case of failure, the error is propagated

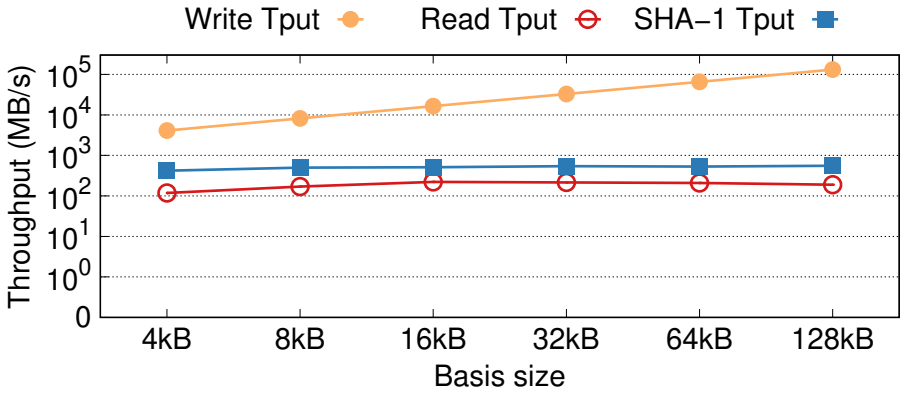


Figure 6.6: Performance trade-offs (reading, writing, basis hashing throughput) for increasing basis size.

back to the client (Figure 6.5-❽). If all bases are successfully written on the KFS, T_{config} and the fingerprint-deviation mapping are stored on disk (Figure 6.5-❾).

Reading files. The process of reading files follows a different path, by first reading T_{config} and the fingerprint-deviation mapping from KFS. MF updates the mapping by loading the bases from KFS replacing the fingerprints listed in the mapping. MF then passes the updated mapping and T_{config} to TF, which applies the inverse transformation function T^{-1} and reassembles the original data. Finally, MF returns the reassembled data to the client. Note that for optimization purposes, MF internally keeps track of the bases already fetched from the backing KFS, with the goal of reducing the number of system calls to the underlying file system.

In the next section, we take a look at how we can implement this design in a FUSE-based prototype.

6.4 Implementation

Based on the architecture described in the previous section, we developed a prototype implementation to test the practicality and performance of Generalised Deduplication within a file system. This particular prototype was written in C+17 using the FUSE framework (v3.9) [149]. During this development, we made specific implementation choices based on micro-benchmarking and arbitrary research-based objectives. Namely, these choices can be summed up as follows: (1) we set the chunk size to 4 kB, (2) chunks are transformed only when a file is closed and (3) our registry is fully disk-based. In the remainder of this section, we motivate some of these implementation choices and the lessons learned while making them.

6.4.1 Processing

Based on the design proposed in Figure 6.5, we first take a look at the processes of transformation and deduplication.

Fingerprinting of bases. We generate the SHA-1 [60] basis fingerprints using OpenSSL (v1.1.0) [88]. While SHA-1 is deprecated for cryptographic use, its ability to produce hashes at high throughput with low collision rates is still interesting for deduplication. We use the fingerprint as the path to a basis, converting the 20 bytes hash into its corresponding

hexadecimal representation. With the hashing algorithm settled on, let us now take a look at the size of the bases that must be handled by the hashing fingerprinting time.

Transformation function and basis size. One important decision that system maintainers must settle on is the specific transformation function. While our design is modular and allows to plug-in alternative implementations, we currently only support Hamming codes (see Section 6.2). In this specific case, one must choose a proper value for message length m. To decide upon this value, the following trade-offs must be considered, as shown in Figure 6.6: (1) a small m is desirable, as it increase the probability of bases matching; (2) however, a small m also increases the total number of bases in the system, which requires additional read and write operations when storing and retrieving files; and (3) with a large m, the SHA-1 fingerprinting becomes a bottleneck.

In MinervaFS, to choose a proper initial value for m, we evaluate the throughput for reading and writing bases to the system for varying sizes (from 4 kB to 128 kB, *e.g.*, the biggest chunk size in ZFS), alongside the throughput for generating SHA-1 fingerprints for bases of the same size. As expected, smaller bases lead to lower read/write throughput. From 32 kB to 64 kB, there is only an increase of 1.15× and 1.18× in throughput for reading and writing, respectively. Additionally, we observe a greater performance penalty (5.9×) in reading throughput by increasing the basis size from 64 kB to 128 kB. From these results and to match the typical block read/write requests issued to file systems, we settle on 4 kB.

While our transformation functions work on 4 kB chunks, our current prototype only triggers these functions when a file is closed. More specifically, when a new file is open for writing, it is first created in a staging area where all blocks are written as-they-are without any transformation. At closing of the file, the transformation is triggered and all its chunks are processed before deduplication. When an existing file is open for reading or writing, it is fully decoded, read and/or modified in the staging area and re-encoded in case of modification. While this process is sub-optimal when reading small parts of large files, it also provides better caching opportunities.

Speeding up transformations with multi-threading and AVX. The current implementation of MinervaFS supports Hamming codes as transformation functions. To improve the throughput from our initial implementation, we made several optimisations: avoiding the resizing of data containers, reducing random access in matrices, parallelizing the computation with multi-threading and enabling instruction multiple data (SIMD) computation by way of Advanced Vector Extensions (AVX) [153]. These optimisations make it possible to increase or decrease the chunk size without significant performance degradation in the computation of the transformation function.

6.4.2 Storage

With the transformation and deduplication handled, we now have a look at the way data is persisted on the backend. We have seen earlier that GD, like other deduplication techniques, requires the maintenance of a registry $\mathcal{R}$ (see Section 6.3). For MinervaFS, we make the arbitrary choice of reading/storing both data and metadata on disk. More specifically, we design a registry fit to work on top of an ext4 file system. For this to work, we set up a backend directory `.minervafs` (location configurable) where we store both the data and metadata. In the remainder of this section, we describe how to build such a registry leveraging the strengths of ext4 and working around some of its issues [200].

Bases As mentioned earlier, we identify bases using their SHA-1 fingerprint (see Section 6.4.1). The choice of SHA-1 implies the possibility of having to store up to 2^{160}

different bases. A naive approach to this problem would be to store all of these in separate files within a flat directory. However, a known issue in the ext family of file systems is a heavy performance penalty when handling large directory indexes [200]. To avoid this issue, we store our bases in a directory structure functioning as follows.

Starting from the top-level of our indexing directory, we take a look the first byte of the basis' fingerprint. The first byte determines the *major group* of the basis and one of the 256 possible directory to which it is assigned. For each of this major group directories, we take a look at the second byte of the fingerprint. This second byte gives us the *minor group* of the basis and one of the 256 possible directory to which it is assigned. Finally, we store the basis within the corresponding minor group directory. With this two-step breakdown, we already have divided the fingerprint space by 2^{16} and can easily find the existence of a basis by constructing the path to the basis.

We exploit the directory-based indexing to identify already existing bases. By constructing a unique path for a basis, consisting of the major group, the minor group and the fingerprint of the basis, it is trivial to test if the basis exists on disk with a single system call. As will be shown in Section 6.5, these system calls are fast enough and make it unnecessary to keep the registry that maps the bases to their physical location in memory. Hence, MinervaFS memory requirements are significantly lower than state-of-the-art deduplication file systems.

As a consequence of this multiplication of directories to host the indexing structure, we must look at the storage overhead beyond the bases themselves. Indeed, the size of the major and minor group directories in MinervaFS will vary depending on the number of files they contain. With an initial size of 4 kB, the directory size increases over time when the number of elements exceed what can be represented within the initial 4 kB. Assuming our ext4 file system stores directory entries in a hash-tree [110], we can make the following estimations about the cost of this hierarchy. For each directory, the hash-tree requires 40 bytes of storage. Each entry in the hash-tree requires a mandatory 8 bytes in addition to the bytes needed to represent the file name. As the basis filename is 40 bytes long, adding a new basis to the system expands the hash-tree by 48 bytes. We recall that in worst-case deduplication scenario, where a minimal number of basis require the creation of the 2^{16} major and minor group directories, the cost of storage on the backend is $2^{16} * 40$ bytes or 2.5 MB for directories. Thus, we can define the fixed size of the registry as $\mathcal{R}_{MFS} = 48 \cdot N + 2^{16} \cdot 40$ bytes where N is the number of bases in the system. The total disk space required for storing the bases is $N \cdot k$ bits, where k is the size of a basis. Given a target registry size, $|\mathcal{R}_{MFS}|$ in bytes, then $N = \frac{|\mathcal{R}_{MFS}| - 2^{16} \cdot 40}{48}$ is the number of bases that can be stored in MinervaFS. This result is useful in order to compare the registry size with respect to the total data stored as bases. Knowing that in the worst case, each chunk maps to a unique basis (*i.e.*, different from all other bases in the system).

In Table 6.1, we compare the amount of data that can be indexed by (a) MinervaFS depending on the chunk size (from 4 to 128 kB) and (b) ZFS with 128KB chunks (default) given a total size of the registry (1, 5, 20 GB). For a chunk size of 4 kB and a registry of 5 GB, MinervaFS can store 2.34× fewer bytes (worst case) than ZFS using 128 kB chunks. If we increase the chunk size in MinervaFS to 16 kB or above, we can store at least the same amount of data as ZFS, and often more, with the same registry size. In fact, for the same chunk size (128 kB) MinervaFS can store 13.65× more data for the same registry size. Given that MinervaFS has better compression potential than ZFS, the total raw data that can be stored for the same registry size will be much larger in MinervaFS than shown in Table 6.1. Finally, given that a smaller basis/chunk in MinervaFS has a higher probability

Table 6.1: Amount of data that can be indexed given a certain chunk-size and total registry size for MinervaFS and ZFS. Greater is better.

	MinervaFS						**ZFS**
Chunk	4 kB	8 kB	16 kB	32 kB	64 kB	128 kB	128 kB
Registry	Storage on disk (GB)						
1 GB	81	170	340	680	1362	2724	200
5 GB	426	853	1706	3412	6823	13647	1000
20 GB	1706	3413	6826	13652	27303	54607	4000

to find matches, we use smaller chunks sizes than the default for ZFS, *e.g.*, 4 kB. Please recall that while we deliberately choose a full on-disk registry in this implementation, partly to measure the performance impact, MinervaFS can easily be adapted to run its registry partially or completely in memory.

Metadata and deviations To keep track of the information that represents a file, we use *placeholder files*. For each file stored in the system, we create a placeholder file containing the path of the file f_{id}, the transformation configuration T_{config} and the ordered list of basis-deviation pairs that make up the file I_i. We have seen that bases are stored in their own separate files, so we only need to store their fingerprints. However, deviations are stored in full within the placeholder file. To avoid repetitions of basis fingerprints and full deviations in the placeholder files, we first number each unique element, be it a basis or a deviation, and replace any re-occurrence in the sequence with their number. Additional information regularly queried by `stat` calls, such as the file size, can also be added to the file.

To save the file, we use the path given by the client f_{id} as the storage path in our backend, thus mirroring the hierarchy seen by the client. While some overhead might be expected from having to move the file around on `rename` calls, this also makes it easier to list the content of a directory. The placeholder file is serialized on disk in a pre-defined binary format that enables us to quickly lookup information within the file without having to decode it in its entirety.

Durability To ensure the durability of data written to MinervaFS, we leverage our choice of a 4 kB basis and rely on the behaviour of ext4. On an ext4 file system, a 4 kB basis size ensures that the storage of a basis only requires the use of a single `inode`. MinervaFS considers that the storage of a basis is successful only if the creation of the inode, writing of the data and linking of the inode succeed. If the inode creation fails, there is no corruption of the file system. If the inode is successfully created but not linked, due to a failure in writing or linking, the orphaned inode will be deleted automatically upon partition remount.

As described in Section 6.3.1, the process to store a file first writes bases as separate files on the file system. After all bases are successfully stored, the metadata associated to the file is stored. The successful execution of these steps, in this precise order, is mandatory for MinervaFS to acknowledge the data as written to the system.

6.5 Evaluation

This section presents our in-depth evaluation of MinervaFS. We compare against state-of-the-art systems and techniques commonly used to reduce storage overhead, a primary goal of deduplication systems.

Table 6.2: Details of the datasets used for the evaluation of MinervaFS

Dataset	Size (GB)	# files
HDFS logs [307, 293]	17.526	32
Satellite images [197]	1.3	69
Satellite pansharpend [197]	3.3	69
Virtual Disk Images [195]	671,21	1420

6.5.1 Evaluation settings

Unless stated otherwise, we deploy MinervaFS on a node with Intel(R) Xeon(R) CPU E5-1620 v4 at 3.50GHz, 10MB of L3 cache, 32GB of DDR4 RAM, running Ubuntu 18.04.1. Experiments are deployed on a 4TB Seagate IronWolf HDD with 64MB cache clocking at 5900 RPM. The HDD is formatted with an ext4 file system to serve as the KFS (Section 6.3). Using a spinning HD allows us to stress the seeking operations required to operate on the basis and deviations.

Disk usage is measured with the `du` system command with options `-sb` [102]. For ZFS we use two additional commands: `zfs` and `zpool` both with the option `list` [166, 167]. The difference between these two commands is that `zpool` reports the usage and deduplication ratio for a given pool while `zfs` reports the full storage usage for the disk, including space reserved for internal ZFS accounting, *i.e.*, ZFS' counterpart to MinervaFS registry. In our analysis, we use the results of the `zfs` command as it makes for a fair comparison against MinervaFS' usage including registry.

For our evaluation, we rely on real-world datasets, shown in Table 6.2. The HDFS log dataset [307, 293] comprises a collection of large text files. The visual and pansharpened [197] satellite images are in TIFF [4] format [201]. The virtual disk images (VDI) is a collection made of a variety of Linux distributions and versions for virtual machines [195].

6.5.2 Storage usage

We begin with a side-by-side comparison of the storage required for the mentioned datasets when using ZFS, Gzip, xdelta [126, 127], MinervaFS, using ext4 as our baseline file system. We evaluate the gains for each system as new files are added to the dataset.

Figure 6.7 (top) shows that Gzip offers the greatest compression gain compared to the others for all datasets. This is not surprising as Gzip uses a more involved compression mechanism. Gzip's results can be considered as a heuristic for the compression potential of each dataset. However, using Gzip in a file system comes at the expense of direct data access: data has to be fully decompressed before it can be accessed. Gzip is closely followed by xdelta, where the difference in compression is insignificant. Similar to Gzip, xdelta only operates on individual files and is not scalable for larger storage systems.

For the HDFS logs, visual and pansharpened images, ZFS provides little to no compression gain. Namely, 5.52% gain for the visual images, less than 1% gain for the HDFS logs, and *increased* storage usage of 0.32% with the pansharpened images. With the same data sets, MinervaFS provides a 19.29% and 8.58% reduction in storage usage for HDFS logs and the pansharpened images, respectively, which constitutes a significant improvement with respect to ZFS. For the visual dataset, MinervaFS requires a 3.33% storage overhead compared to ext4.

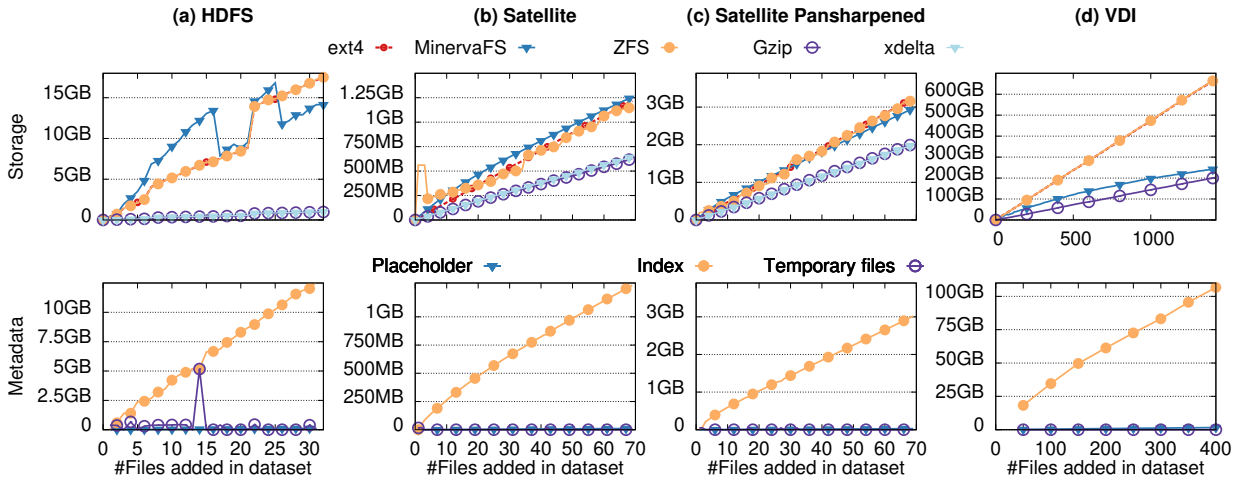


Figure 6.7: Storage (top) and metadata (bottom) overhead.

Figure 6.7 (top-**a, b, c**) show that the storage usage of MinervaFS starts high when compared to ext4 and ZFS, but quickly drops as more data is added to the system. However, it is important to remember that this storage usage for MinervaFS also includes the registry (built directly on disk), while ZFS' registry is represented in memory and not included in these results. ZFS will pay an added cost to store its registry persistently.

For VM images [195] in Figure 6.7 (top-**d**), MinervaFS achieves a 63.53% reduction in storage usage compared to ext4. This means a 2.74x deduplication ratio with all costs (incl. data, registry) taken into account. This is remarkably close to Gzip, which provides a 69.72% reduction in storage usage. Although `zpool` reports that a ZFS storage pool usage of 423GB and a deduplication ratio of 1.59x after all files are added, the `zfs` tool, which gives the full storage usage of ZFS including the storage pool, reports a total usage of 672GB (or 249GB more than `zpool`). For this dataset, we were unable to collect data for xdelta. Due to an error in its internals, xdelta would inconsistently crash when handling files of a larger size. Despite our best efforts to split the data into smaller chunks of 256, 145, and 128MB, xdelta would still fail too often and inconsistently for us to report its results here.

In Figure 6.7 (top-**a, b**) we can observe MinervaFS's storage overhead fluctuating through rises and drops. This is due to temporary files being kept on disk before encoding, temporarily increasing the storage usage of MinervaFS. The reason this is not observed in Figure 6.7 (top-**c, d**) is because of the timing between temporary file deletion and disk usage sampling.

6.5.3 Metadata overhead

To understand how the various MinervaFS metadata elements contribute to the observed storage overhead, we analyze the split between indexing, placeholder files and temporary files (Figure 6.7-bottom).

As expected, the indexing elements are the heaviest contributor, with up to 99.88% in the case of the pansharpened satellite images. These results also confirm that temporary files can indeed cause transient spikes in storage consumption, as we see in the HDFS case (Figure 6.7 (bottom)-**a**).

6.5.4 System throughput

Next, we study the throughput of MinervaFS compared to other file systems capable of deduplication, *e.g.*, BtrFS and ZFS. To better highlight the differences in terms of performance, we run the following experiments on top of an SSD. Specifically, we first run a synthetic workload using filebench [260, 261] and compare the results against our ext4 baseline. In this workload, filebench emulates a web-server under constant load of 100 threads for a duration of 15 minutes serving web pages. Upon each thread request, the file system must reads 10 files (1.9 MB) and appends to a log file (16 KB). Each system is made of a single partition, created and mounted just before the beginning of the experiment.

The results are presented in Figure 6.8 (left). On the x-axis we report the file systems, on the y-axis the relative throughput compared to the ext4 baseline. Our ext4 baseline achieves an average throughput of 514 MB/s. As expected, ZFS and BtrFS perform very similarly to the baseline but so does MinervaFS. BtrFS emerges as the best candidate under this workload with an average of (501 MB/s) but it should be noted that it does not perform in-band deduplication out of the box and requires some extra patching and tweaking in order to be exactly comparable. For this reason, we show in Figure 6.8 the results for ZFS with and without (*i.e.*, ZFS n.d.) deduplication achieving respectively 407 and 482 MB/s. Looking at MinervaFS's results, its throughput averages 472 MB/s, around 16% faster than ZFS with deduplication. We explain these results by the fact that all read and write operations are performed on the decoded versions of the files boiling down to results close to our baseline. The small penalty is due to the time spent encoding and decoding, *i.e.*, performing the transformation into bases and deviations. Additionally, the small writes to the log file partially affect the results. In fact, due to its shared nature, the log file is maintained in cache in its decoded form during most of the workload (see Section 6.4). Note that the performance of MinervaFS could further be improved by disabling the `direct_io` option and leveraging the system cache to decode the file on disk but read its content from the cache. For the sake of fairness in our comparison against the other systems, we did not implement this. We plan to analyse this effect in future work. From these results, MinervaFS appears to be a sound solution file system for read-heavy workloads.

Finally, we evaluate MinervaFS under a write-heavy workload, as shown in Figure 6.8 (right). We replay a real-world dump of Wikipedia (*i.e.*, the `SCN` dialect [286]), consisting of 36 377 pages and 355 367 revisions overall. The revisions are replayed according to the global timestamp of each edit issued by its Wikipedia editor. Each revision consists of a new `html` page, stored into a MinervaFS partition (or alternative systems) alongside all the previous partitions, the same way an archival service would do. The impact of inband deduplication is significant. We observe that MinervaFS achieves only 35% of the ext4 throughput, while ZFS reaches 65%. We explain this by (a) the choice of smaller chunks written in MinervaFS than in ZFS, and (b) the need to use sequential write of bases (not multi-threaded) for consistency/durability needs.

6.6 Discussion

In this section, we discuss the shortcomings of our prototype MinervaFS and present opportunities for improvement and future work.

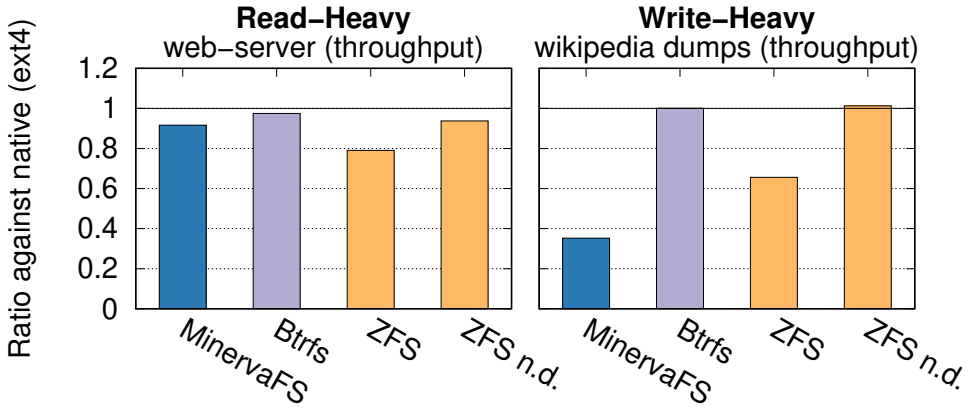


Figure 6.8: Read-heavy (left) write-heavy (right) workloads. We show the relative results vs. the ext4 baseline

6.6.1 Granular encoding and decoding

As discussed in Section 6.4, the current version of MinervaFS triggers the transformation of chunks on closing of the files. This choice of implementation has consequences on the way files are read and written to the system.

From a reading standpoint, a client application must wait for the file to be fully decoded before it can read. A simple GNU utility such as `file`, reading the first bytes of a given file to determine its type would need to wait for the entire blob to be decoded before returning with an answer. While this approach is wasteful in terms of computation and storage, it has the benefit of providing some nice caching effects for files that are left long open. However, with an uncontrollable variety of applications running on top of it, MinervaFS should be able to provide per-chunk decoding at a reasonable cost. This tweak requires the matching of a given offset to a chunk and the decoding of that single chunk. The matching of the offset can already be computed at the file system layer. The decoding of the single chunk should be possible as our decoding function already performs this work for each chunk in the file.

In terms of writing, waiting for the file to be closed by the client application to transform the data ensures a consistent writing throughput but compounds the deduplication and transformation to the end of the process. A per-chunk approach could again help distribute the encoding load over time, making each write slower but closing the file almost instantly. However another approach is possible: asynchronous encoding. With asynchronous encoding, the writing process remains similar to our current implementation but only changes in the way files are closed. Specifically, we acknowledge the closing of the file to the client application and proceed to the deduplication and transformation of the file while the client application returns to its regular activity. But what this approach offers in terms of responsiveness, it takes away in terms of durability guarantees (see Section 6.4.2).

To summarize, the performance and efficiency of MinervaFS could further be enhanced with tweaks to its encoding and decoding strategies.

6.6.2 Garbage collection of unused bases

MinervaFS, like many other systems, allows the deletion of chunks or complete files. In our system, this translates to the deletion of bases, deviations and placeholder files. Since deviations are stored within placeholder files (see Section 6.4.2), removing them from the

placeholder file or deleting the placeholder file itself is pretty straightforward. Unused bases, however, require finer handling. Indeed, to maintain the storage reduction gained from GD, we must be able to identify unused bases and safely remove them from the system. To solve this issue, we propose the following design for a Garbage Collection (GC) component. We consider two different strategies for GC of unused bases, depending on the type of registry used.

First, we look at GC for an in-memory registry where the file system and the garbage collector run within the same address space. In this case, a counter tracks how many files reference a basis. The counter is updated as follows: when a new file references a basis, the counter is incremented; conversely, when a file referencing a basis is deleted, the counter is decremented. The basis can be safely discarded when the counter reaches zero. The collection can then be done by pausing the entire system or keeping the system running while locking the targeted bases and marking them as deleted. Using this in-memory approach implies a larger memory overhead at runtime and requires on-disk persistence at unmount time.

The second strategy is adapted to an on-disk registry. In this scenario, we help the identification of unused bases by maintaining a deletion log. Every time a file or a chunk is deleted in MinervaFS, the related bases are added to the log. Upon GC time, the log is read, unused bases are identified, deleted and an up-to-date compacted version of the log finally is persisted. This approach does not incur the permanent memory overhead of the first approach but lengthens GC's indexing and completion time. It also requires the full file system to be paused for the GC to complete safely.

While the first approach is more suitable for a system with frequent updates to existing data, the second might be a better fit for archival systems where new data is added at a regular rate but not often updated. In the end, the choice of one approach over the other depends on one's use of the file system.

6.6.3 Moving Generalised Deduplication up and down the stack

Broadly speaking, for a client application to benefit from deduplication, a software deduplication scheme can be integrated at three different levels: (a) between the client application and the file system, (b) within the file system or (c) between the file system and the final medium of storage. Throughout this chapter, we had a look at systems that deduplicate at the level of the file system (b) but it might be worth taking a look at integration at other levels.

Looking at the integration of Generalised Deduplication (GD) between the application and the file system, we can already exclude the idea of modifying all existing applications. But proposing modifications at the border of these applications remains a possibility. Indeed, we have discussed earlier the idea of injecting functionality within a system by substituting libraries at runtime (see Section 5.7.3). However, unlike the proposition discussed earlier, blending deduplication behind library calls goes beyond modifying the format of the data exchanged. In order for deduplication to work properly, metadata must be added and maintained beyond the runtime of the application. In the case where multiple applications leverage the same library for deduplication, maximizing data reduction among processes that were not designed to coordinate together can be a daunting task. In short, beyond the difficulty of providing a semantically equivalent library to read and write data to the file system, the need to centralize metadata and properly coordinate its management make it impractical to integrate at the application level.

At the bottom of the stack, we consider the storage medium, related controller and driver out of scope. But that does not prevent a deduplication system from setting up shop

close to the metal. Red Hat's Virtual Disk Optimizer [62], or VDO, sets up as an abstracton layer between the file system and the block device. Split between an indexing and a layer kernel module, VDO provides zero-block elimination, deduplication and compression. The combinaton of deduplication allows for better data reduction than regular deduplicating systems. VDO aggregates block devices to build logical volumes where the apparent size can be set to appear greater than the total storage space assembled. But despite the good compression gains and comprehensive tooling, the system remains complex to administer across mulitple layers, limited to a number of file systems (ext4 and XFS) and only available for Red Hat distributions.

To summarize the issues at hand, a deduplication engine should remain close to the file system it works for, possibly within the same unit, and centralize all information to achieve good compression gains. Consequently, the last option would be to move the whole of MinervaFS down the stack, in the kernel, to benefit from the same quality of data reduction while reducing the number of context switches in system calls. We see this change as an engineering challenge, as MinervaFS heavily relies on the C++17 standard library and other external dependencies. We envision this push of MinervaFS down to the kernel as part of future work.

6.6.4 Selecting type relevant transformation functions

Starting from our motivational example in Figure 6.3, we can see that picking an efficient data reduction scheme that fits all datasets is not simple. Indeed, where pne might expect, for all datasets, to see compression gains, even minimal, we sometimes observe a growth in terms of storage overhead (see the satellite images for instance). To solve this issue, we see an opportunity in selecting a specific transformation function depending on the type of file being processed.

In theory, this should be feasible in MinervaFS as the function and parameters used for transformation are systematically saved as part of the metadata. We mentioned earlier that work IoT generated data using Reed-Solomon codes [273] and a transform-and-truncate function [274] yielded good results on smaller chunks of data. In particular, with the current implementation of MinervaFS waiting for the file to be closed before transforming it, identifying its type right before picking the right function offers a good opportunity for improvement. However, a differentiated treatment according to file type should also take into consideration original file length, appropriate chunk size and compatibility with client application read/write patterns. A question lies in whether great compression gains justify long processing time if a file is always read sequentially from start to finish or read and modified in random patterns. It might also be a good opportunity to detect file formats and containers that are notorious for their lack of redundancy and store them in their original form or limit their processing to classic deduplication. But to accommodate this flexibility in terms of processing, some engineering effort will be required to provide a flexible and suitable backend storage organization.

In the end, whether information aviut file type is used to decide how to properly transform data, objectives must be clearly defined to strike the right balance between good compression gains and reasonable encoding/decoding time.

6.7 Summary

In this chapter, we presented a prototype integration of Generalised Deduplication (GD) at the file system level. Building on the design, implementation and evaluation of our user-space prototype MinervaFS, we showed the suitability of GD for various types of data and workloads. Our evaluation with synthetic and real-world datasets shows significant storage compression gains compared to state-of-the-art classic deduplication file systems, *e.g.*, up to 63.73% compression gain when compared to ZFS. In fact, for the VM image dataset we only use 20 % more storage than Gzip, while using 2.75x less storage space than ZFS. Future work will focus on granular encoding/decoding of files, efficient garbage collection, general performance improvements and alternative transformation functions. In particular, we plan to identify specific transformation functions matching the features of the content types of processed documents, *e.g.*, optimized for images, sound, VM images. Finally, our future work will explore the potential of using standard compressors, *e.g.*, Gzip, xdelta, to compress individual bases in MinervaFS. Given the large deduplication gains on the VM image dataset (2.75x) and the large file compression gains of Gzip (3.3x), we conjecture that the resulting combination will result in a much larger compression gain as high as 9x ($2.75 \times 3.3 \approx 9$).

Chapter 7

Conclusion

We started this work by focusing our efforts along 3 angles. Throughout the different chapters, we built and evaluated a series of prototypes, breaking down different parts of a cloud-based storage system improving on security, reliability and storage overhead. In this chapter, we look the lessons learned from these efforts and outline the directions for future work.

7.1 Summary

Secure and reliable cloud storage from the client-side

The first question we asked was how we could empower clients to securely and reliably store data in the cloud. To answer this, Chapter 3 looked at solutions to build on top of existing public cloud storage providers such as Dropbox, Google Drive and Microsoft OneDrive. To do so, we started with the study and evaluation of erasure coding libraries ensuring that documents uploaded to the cloud could be recovered in case of failure on the part of the storage provider. To guarantee the confidentiality of documents before they were offloaded to the cloud, we built a processing pipeline enabling the combination of encryption, hashing and signature. For completeness, we added a primitive XOR-based secret-sharing scheme to leverage distribution to build confidentiality. From our evaluation of this setup, we learned that while the seamless combination of multiple schemes is feasible, it translates to a significant increase in CPU and memory usage. On the flip side, it does offer the possibility to store data over free-tier services without having to sacrifice on privacy or reliability. But providing users with a solution to maintain ownership of their data through an obscure REST API may not help with the adoption.

Another approach to integrate feature-rich systems is to change the interface users interact through to something more traditional like the POSIX file system API. This is the path chosen by multiple storage solution wrapping their unconventional interfaces in a file system. One of the most efficient ways to leverage an existing solution is to turn to FUSE. In Chapter 5, we looked at how the implementation requirements of user-space file systems developed with FUSE could be leveraged to build a modular solution encouraging the reuse of existing implementations. By building a layered system where each layer can be powered by different drivers, SafeFS enables the quick configuration of the file system with no need to re-compile. We built a series of security-oriented configurations and compared their throughput against other monolithic file systems in user-space. Our main observation is that, in addition to encouraging the reuse of existing implementation source code, SafeFS performance fairs comparably to monolithic solutions.

In summary, to improve client-side control of security and reliability in cloud storage, we measured the impact of different security configurations and proposed SafeFS, a modular framework to build feature-rich user-space file systems.

Resilient archival storage in the cloud

With the question of secure and reliable cloud storage from the client-side tackled, we then looked at how the protection of data archived in the cloud could be improved beyond the use of standard erasure coding schemes. To answer this question, we introduced Recast in Chapter 4, a fully functional implementation of STeP [14]. STeP blends in erasure coding and data entanglement by picking older blocks randomly and inserting them into the input of the coding function to create interdependency links that multiply collateral damage in case of corruption but also enable block recovery beyond the erasure code's capability. Building on the knowledge of erasure codes gathered in Chapter 3 and expanding on the content of the original STeP proposal, we were able to implement an improved version that offers a faster long-term protection to recent documents.

We evaluated the throughput, storage overhead, repair bandwidth and metadata reconstruction process offered by Recast. Our observations are consistent with the ones seen in the preceding chapter with a linear correlation in the throughput decrease as well as the increase in bandwidth use and storage overhead. But despite the considerable improvement in reliability for the data archived in Recast, the system is not perfect. Recast's random nature requires the maintenance of a metadata service. While the backend storage is itself resilient to attacks, the loss of metadata can make the decoding of archived data impossible. For this reason, replication of the metadata service is necessary. In addition, its permanent nature makes it complicated to delete documents form the archive making the system unsuitable for GDPR-compliant systems (though tweaks are possible).

In summary, to increase the protection of archival data in the cloud, we proposed Recast, a system that significantly improves resilience at the cost of increased processing complexity but manageable storage overhead.

Practical and efficient data reduction for file systems

Most of the measures and improvements proposed of this work tend to increase the storage overhead. However, storage space can get costly, be it locally or in the cloud, can be costly. In order to keep the storage costs under control, Chapter 6 studies the use of Generalised Deduplication (GD) in a user-space file system delivering MinervaFS. Distinguishing itself form classic data deduplication by finding reduction opportunities beyond exact similarity, GD is a prime candidate to improve storage efficiency.

To put GD to the test, we built MinervaFS, a FUSE-based file system using GD for data reduction, measuring its storage use (both in terms of metadata and data) and its throughput. From the results, we could draw observations that are both implementation specific and generally useful for the understanding of GD. Generally speaking, GD offers great compression gains on the data when compared to other deduplication enabled systems. However, what GD gains in data use it pays for in processing and metadata complexity. In

addition to the processing complexity, the potential compression gains on some datasets are canceled out by the increase in metadata. In the case of MinervaFS, our prototype implementation, as files are only deduplicated at closing time, most of the processing is compounded at delayed but still blocking time. But this is merely an engineering issue that also has the benefit of bringing some new insights for the future integration of GD in other systems.

In summary, to tackle the issue of storage overhead growth introduced by security and reliability measures, we integrated Generalised Deduplication to a user-space file system in order to improve compression gains at the expense of increased processing and metadata complexity.

7.2 Perspective - Piecing it all together

Trying to piece these contributions together, a clear idea emerges: a trustworthy and reliable use of cloud storage requires strong client-side control. Whether data is uploaded to be shared and modified, or is simply stored online for archival purposes, users should feel confident that their files are safe. Starting with the mindset that remote storage nodes are simple, unsophisticated and unreliable buckets of data helps in the design of secure and resilient storage systems. It only stands to reason that setting up a client-side pipeline that leverages the best of coding techniques while accounting for the potential failures in a distributed setting is the best approach to build qualitative cloud storage solutions. However, we have seen that improvements in confidentiality and reliability are often paid for by an increase in resource usage (CPU, memory or bandwidth). For this reason, the choice of a complex combination of primitives must be made with the knowledge of the overhead it might put upon the user's workflow. Indeed, a combination of the most secure SafeFS configuration with data deduplicated using Generalised Deduplication, distributed and entangled over a set of remote storage nodes is probably too complex and too disruptive to most workflows.

A better approach would have us suggesting configurations based on generic scenarios and threat models. However, choices in storage configuration are not solely driven by technical and strategic motives. The storage of information for commercial, medical or telecommunication services is also bound to legislative and industry specific requirements. In particular, the need for some systems to allow complete deletion and de-referencing of data owned by customers while guaranteeing the retention of metadata for a minimum amount of time may make the use of our contributions complicated. Specifically, the prohibitive cost of removing files from Recast makes it complicated to comply with the European right to be forgotten. It may be fine for organization such as the Internet Archive to host public domain items in such a convoluted system. But in a system where documents are more likely to be contested or modified over time, the cost of removal may be too high.

Rather than closing this work with mixed feelings over the integration of our contributions, let's take a final look at where they can be used. While cloud storage is virtually limitless, especially if you happen to be sitting on a bottomless pot of gold, most customers are only familiar with the free-tier options of these services. The combination of a secure and redundant client-side configuration bundled in SafeFS coupled with Generalised Deduplication can save users worries and money. Providing effective security and data reduction tools that integrate seamlessly is one way to prevent crossing restrictive storage quotas while

maintaining some amount privacy. And in a world where the boundaries between systems keep getting fuzzier and no configuration fits all deployment scenarios, equipping users with the right tooling can help them make informed decisions and maintain ownership of their data.

7.3 Directions for future work

The development of distributed storage systems is a difficult endeavor. In particular, conciliating security, reliability and storage efficiency can be a tricky balancing act. Through the work done is manuscript, we have seen that building complex and feature-rich systems is feasible at the expense of growing resource consumption (be they time, memory or computation). The search for a perfect or universal solution may not be a wise use of time. However, with some engineering and research efforts some significant improvements could be made over the contributions proposed in this work.

Providing the right tooling to craft customized solutions is a step forward to empower users to maintain ownership of their own data. To that end, making the process of building said solutions easier would add significant value to a proposal like SafeFS. Specifically, avoiding the addition of setup or tear down functions to an existing FUSE implementation and leveraging binary rather than source compatibility would be beneficial to this adoption. Short of that, tooling generating sample code could be enough to ease FUSE developers into using SafeFS.

Another aspect of distributed storage systems that was left unaddressed in this work is proof of storage. While most systems actively check the integrity of data fetched from remote storage nodes, this method of integrity checking may prove to be too bandwidth hungry for some settings. To fix this, proofs of storage allow the system to send a challenge to storage nodes in order to prove that they are effectively storing the data they claim to host. Unlike Recast, most systems do not benefit from an error propagation mechanism that allows them to detect that data hosted remotely has been corrupted, deleted or was actually never stored. Building such a system is complicated as it involves extending the trust in a remote machine from simple storage to running some computation and returning a result. Setting aside the need for public storage providers to implement compatible APIs, Recast and similar systems could try to integrate with remote servers sporting Intel SGX based on existing proposals [59].

The use of a data reduction scheme such as Generalised Deduplication has shown us that potential compression gains are often left on the table. But it has also shown us that no practical data reduction method is guaranteed to save us space on all types of files. A study of a system combining Generalised Deduplication with other compression techniques could offer some interesting insights in its behaviour in more complex pipelines. A tiered approach to data reduction, where data could be standing at different layers uncompressed, deduplicated and compressed, would also be interesting to study when put under different workloads and caching or prefetching policies.

Finally, most of the contributions made in this work have taken the form of storage middleware but none have been integrated as part of existing file systems, databases or big data solutions' backend. Integration within such systems would certainly yield interesting insights into the versatility and ease of use of our contributions.

Appendix A

Publications

2016

1. Dorian Burihabwa, Pascal Felber, Hugues Mercier, Valerio Schiavoni.
   A Performance Evaluation of Erasure Coding Libraries for Cloud-Based Data Stores - (Practical Experience Report)
   Distributed Applications and Interoperable Systems (DAIS 2016),
   Heraklion, Crete, Greece, June 2016, pp 160–173.
   doi: 10.1007/978-3-319-39577-7_13

2. Dorian Burihabwa, Rogerio Pontes, Pascal Felber, Francisco Maia, Hugues Mercier, Rui Oliveira, João Paulo, Valerio Schiavoni.
   On the Cost of Safe Storage for Public Clouds: An Experimental Evaluation
   35th IEEE Symposium on Reliable Distributed Systems (SRDS 2016)
   Budapest, Hungary, October 2016, pp 157–166.
   doi: 10.1109/SRDS.2016.028

3. Dorian Burihabwa.
   PlayCloud: A Platform to Experiment with Coding Techniques for Storage in the Cloud
   35th IEEE Symposium on Reliable Distributed Systems (SRDS 2016)
   Budapest, Hungary, October 2016, pp 215–216.
   doi: 10.1109/SRDS.2016.037,

2017

4. Rogério Pontes, Dorian Burihabwa, Francisco Maia, João Paulo, Valerio Schiavoni, Pascal Felber, Hugues Mercier, Rui Oliveira
   SafeFS: a modular architecture for secure user-space file systems: one FUSE to rule them all
   Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR 2017)
   Haifa, Israel, May 2017.
   doi: 10.1145/3078468.3078480

2018

5. Roberta Barbi, Dorian Burihabwa, Pascal Felber, Hugues Mercier, Valerio Schiavoni.
RECAST: Random Entanglement for Censorship-Resistant Archival STorage
48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2018)
Luxembourg City, Luxembourg, June 2018, pp 171–182.
doi: 10.1109/DSN.2018.00029.

6. Dorian Burihabwa, Pascal Felber, Hugues Mercier, Valerio Schiavoni.
SGX-FS: Hardening a File System in User-Space with Intel SGX
2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2018)
Nicosia, Cyprus, Dec 2018, pp 67–72.
doi: 10.1109/CloudCom2018.2018.00027

2019

7. Arnaud L'Hutereau, Dorian Burihabwa, Pascal Felber, Hugues Mercier, Valerio Schiavoni.
Blockchain-Based Metadata Protection for Archival Systems.
IEEE 38th Symposium on Reliable Distributed Systems (SRDS 2019).
Lyon, France, October 2019, pp 315–323.
doi: 10.1109/SRDS47363.2019.00044

2020

8. Lars Nielsen, Dorian Burihabwa, Pascal Felber, Daniel E. Lucani, Valerio Schiavoni.
MinervaFS: A User-Space File System for Generalized Deduplication
Submitted to USENIX FAST'21.

Bibliography

[1] *7-Zip*. URL: https://www.7-zip.org/ (visited on 09/07/2020).

[2] C. M. Adams and S. E. Tavares. "Designing S-boxes for ciphers resistant to differential cryptanalysis". In: *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography, Rome, Italy*. Jan. 1993, pp. 181–190.

[3] M. Adler. *madler/zlib*. original-date: 2011-09-10T03:27:31Z. 9th July 2020. URL: https://github.com/madler/zlib (visited on 09/07/2020).

[4] Adobe Developers Association. *Adobe Developers Association*. visited: 17/06-2019. 2019. URL: https://www.adobe.io/content/dam/udp/en/open/standards/tiff/TIFF6.pdf.

[5] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer and R. P. Wattenhofer. "Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment". In: *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*. Vol. 36. USENIX, Dec. 2002, pp. 1–14.

[6] A. Ahmad, K. Kim, M. I. Sarfaraz and B. Lee. "OBLIVIATE: A Data Oblivious Filesystem for Intel SGX". In: *25th Annual Network and Distributed System Security Symposium (NDSS 2018)*. San Diego, California, USA: The Internet Society, Feb. 2018.

[7] A. Alba, G. Alatorre, C. Bolik, A. Corrao, T. Clark, S. Gopisetty, R. Haas, R. I. Kat, B. S. Langston, N. S. Mandagere, D. Noll, S. Padbidri, R. Routray, Y. Song, C. H. Tan and A. Traeger. "Efficient and agile storage management in software defined environments". In: *IBM Journal of Research and Development* 58.2 (2014), 5:1–5:12. DOI: 10.1147/JRD.2014.2302381.

[8] R. Anderson. "The Eternity Service". In: *1st International Conference on the Theory and Applications of Cryptography (PRAGOCRYPT)*. Vol. 96. 1996, pp. 242–252.

[9] A. Appleby Austin. *aappleby/smhasher*. GitHub. URL: https://github.com/aappleby/smhasher (visited on 12/07/2020).

[10] *Archive.org Information – Internet Archive Help Center*. URL: https://help.archive.org/hc/en-us/articles/360014755952-Archive-org-Information.

[11] ARM Limited. *Building a Secure System using TrustZone Technology*. Tech. rep. 2009. URL: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[12] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica and M. Zaharia. "A View of Cloud Computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58. DOI: 10.1145/1721654.1721672.

[13] J. Aspnes, J. Feigenbaum, A. Yampolskiy and S. Zhong. "Towards a theory of data entanglement". In: *Theoretical Computer Science* 389.1 (Dec. 2007), pp. 26–43. DOI: 10.1016/j.tcs.2007.07.021.

[14] M. Augier. "Trustworthy Cloud Storage". PhD dissertation. Lausanne, Switzerland: EPFL, 2016.

[15] *AWS IoT Analytics*. visited: 19/09-2019. 2019. URL: https://aws.amazon.com/iot-analytics/.

[16] R. Barbi. "Erasure coding for distributed storage systems". PhD dissertation. Neuchâtel, Switzerland: University of Neuchâtel, 2019.

[17] R. Barbi, D. Burihabwa, P. Felber, H. Mercier and V. Schiavoni. "RECAST: Random Entanglement for Censorship-Resistant Archival STorage". In: *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018*. Luxembourg City, Luxembourg, June 2018, pp. 171–182. DOI: 10.1109/DSN.2018.00029.

[18] *bedup Btrfs deduplication*. visited: 13/09-2019. 2010. URL: https://github.com/g2p/bedup.

[19] J. B. Bell and D. Mcilroy. "Data Compression Using Long Common Strings". In: *Proceedings of the Data Compression Conference, (DCC 1999)*. Snowbird, Utah, USA: IEEE Computer Society, Mar. 1999, pp. 287–295. DOI: 10.1109/DCC.1999.755678.

[20] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte and M. Wingate. "PLFS: A Checkpoint Filesystem for Parallel Applications". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. Portland, Oregon, USA: ACM, Nov. 2009, 21:1–21:12. DOI: 10.1145/1654059.1654081.

[21] A. Bessani, M. Correia, B. Quaresma, F. André and P. Sousa. "DepSky: Dependable and Secure Storage in a Cloud-of-Clouds". In: *ACM Transactions on Storage (TOS)* 9.4 (Nov. 2013). DOI: 10.1145/2535929.

[22] A. Bessani, R. Mendes, T. Oliveira, N. F. Neves, M. Correia, M. Pasin and P. Veríssimo. "SCFS: A Shared Cloud-backed File System". In: *2014 USENIX Annual Technical Conference*. Philadelphia, PA, USA, 2014, pp. 169–180.

[23] D. Bhagwat, K. Eshghi, D. D. Long and M. Lillibridge. "Extreme binning: Scalable, parallel deduplication for chunk-based file backup". In: *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*. London, UK: IEEE, Sept. 2009, pp. 1–9. DOI: 10.1109/MASCOT.2009.5366623.

[24] A. Bijlani and U. Ramachandran. "Extension Framework for File Systems in User space". In: *2019 USENIX Annual Technical Conference*. Renton, WA, USA: USENIX Association, July 2019, pp. 121–134.

[25] M. Blaze. "A Cryptographic File System for UNIX". In: *Proceedings of the 1st ACM Conference on Computer and Communications Security*. Fairfax, Virginia, USA: ACM, Nov. 1993, pp. 9–16. DOI: 10.1145/168588.168590.

[26] M. Boßlet. *Breaking Murmur: Hash-flooding DoS reloaded*. Dec. 2012. URL: http://emboss.github.io/blog/2012/12/14/breaking-murmur-hash-flooding-dos-reloaded/ (visited on 13/07/2020).

[27] Bottle. *Bottle: Python Web Framework*. URL: http://www.bottlepy.org/.

[28] K. D. Bowers, A. Juels and A. Oprea. "HAIL: A High-availability and Integrity Layer for Cloud Storage". In: *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*. Chicago, Illinois, USA: ACM, Nov. 2009, pp. 187–198. DOI: 10.1145/1653662.1653686.

[29] J. Bowling. "Opendedup: open-source deduplication put to the test". In: *Linux Journal* 2013.228 (2013), p. 2.

[30] A. Broder. "On the resemblance and containment of documents". In: *Proceedings. Compression and Complexity of SEQUENCES 1997*. Salerno, Italy: IEEE, 1998, pp. 21–29. DOI: 10.1109/SEQUEN.1997.666900.

[31] D. Burihabwa. "PlayCloud: A Platform to Experiment with Coding Techniques for Storage in the Cloud". In: *35th IEEE Symposium on Reliable Distributed Systems, SRDS 2016*. Budapest, Hungary, Oct. 2016, pp. 215–216. DOI: 10.1109/SRDS.2016.037.

[32] D. Burihabwa, P. Felber, H. Mercier and V. Schiavoni. "A Performance Evaluation of Erasure Coding Libraries for Cloud-Based Data Stores - (Practical Experience Report)". In: *Distributed Applications and Interoperable Systems - 16th IFIP WG 6.1 International Conference, DAIS 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016*. Heraklion, Crete, Greece, June 2016, pp. 160–173. DOI: 10.1007/978-3-319-39577-7_13.

[33] D. Burihabwa, P. Felber, H. Mercier and V. Schiavoni. "SGX-FS: Hardening a File System in User-Space with Intel SGX". In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Nicosia, Cyprus, Dec. 2018, pp. 67–72. DOI: 10.1109/CloudCom2018.2018.00027.

[34] D. Burihabwa, R. Pontes, P. Felber, F. Maia, H. Mercier, R. Oliveira, J. Paulo and V. Schiavoni. "On the Cost of Safe Storage for Public Clouds: An Experimental Evaluation". In: *35th IEEE Symposium on Reliable Distributed Systems, SRDS 2016*. Budapest, Hungary, Oct. 2016, pp. 157–166. DOI: 10.1109/SRDS.2016.028.

[35] *bzip2*. URL: http://www.bzip.org/ (visited on 09/07/2020).

[36] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan and L. Rigas. "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP 11)*. Cascais, Portugal: ACM, 2011, pp. 143–157. DOI: 10.1145/2043556.2043571.

[37] M. Carlson, A. Yoder, L. Schoeb, D. Deel, C. Pratt, C. Lionetti and D. Voigt. *Software Defined Storage*. Tech. rep. Jan. 2015. URL: https://www.snia.org/sites/default/files/SNIA_Software_Defined_Storage_%20White_Paper_v1.pdf.

[38] G. Cattaneo, L. Catuogno, A. D. Sorbo and P. Persiano. "The Design and Implementation of a Transparent Cryptographic File System for UNIX". In: *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. Boston, Massachusetts, USA: USENIX, 2001, pp. 199–212. URL: https://www.usenix.org/legacy/publications/library/proceedings/usenix01/freenix01/full_papers/cattaneo/cattaneo.pdf.

[39] F. Chen, T. Luo and X. Zhang. "CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives." In: *9th USENIX Conference on File and Storage Technologies (FAST' 11)*. San Jose, CA, USA: USENIX, Feb. 2011, pp. 77–90.

[40] H. C. Chen, Y. Hu, P. P. Lee and Y. Tang. "NCCloud: a network-coding-based storage system in a cloud-of-clouds". In: *Transactions on Computers* 63.1 (Jan. 2014), pp. 31–44. DOI: 10.1109/TC.2013.167.

[41] J. Y. Chung, C. Joe-Wong, S. Ha, J. W.-K. Hong and M. Chiang. "CYRUS: Towards Client-defined Cloud Storage". In: *Proceedings of the 10th European Conference on Computer Systems (Eurosys 2015)*. Bordeaux, France: ACM, Apr. 2015, 17:1–17:16. DOI: 10.1145/2741948.2741951.

[42] A. Cidon, R. Escriva, S. Katti, M. Rosenblum and E. G. Sirer. "Tiered Replication: A Cost-effective Alternative to Full Cluster Geo-replication". In: *2015 USENIX Annual Technical Conference*. Santa Clara, CA, USA, 2015, pp. 31–43.

[43] A. Cidon, S. M. Rumble, R. Stutsman, S. Katti, J. K. Ousterhout and M. Rosenblum. "Copysets: Reducing the Frequency of Data Loss in Cloud Storage". In: *2013 USENIX Annual Technical Conference*. San Jose, CA, USA, 2013, pp. 37–48.

[44] I. Clarke, O. Sandberg, M. Toseland and V. Verendel. *Private Communication Through a Network of Trusted Connections: The Dark Freenet*. 2010.

[45] I. Clarke, O. Sandberg, B. Wiley and T. W. Hong. "Freenet: A Distributed Anonymous Information Storage and Retrieval System". In: *Designing Privacy Enhancing Technologies, International Workshop on Design Issues in Anonymity and Unobservability*. Vol. 2009. Berkeley, CA, USA: Springer, July 2000, pp. 46–66. DOI: 10.1007/3-540-44702-4_4.

[46] A. T. Clements, I. Ahmad, M. Vilayannur, J. Li et al. "Decentralized Deduplication in SAN Cluster File Systems". In: *USENIX Annual Technical Conference*. 2009, pp. 101–114.

[47] R. S. Code. *RECAST Source Code*. URL: https://github.com/safecloud-project/recast.

[48] T. kernel development community. *FUSE — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[49] *Compression in VDO*. Red Hat Customer Portal. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html-single/deduplicating_and_compressing_storage/index#compression-in-vdo_enabling-or-disabling-compression-in-vdo (visited on 13/07/2020).

[50] S. Contiu, E. Leblond and L. Réveillère. "Benchmarking Cryptographic Schemes for Securing Public Cloud Storages". In: *Distributed Applications and Interoperable Systems (DAIS 2017)*. Vol. 10320. Lecture notes in Computer Science. Neuchâtel, Switzerland: Springer, June 2017, pp. 163–176. DOI: 10.1007/978-3-319-59665-5_12.

[51] J. Conway. "The game of life". In: *Scientific American* 223.4 (1970), p. 4.

[52] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan and R. Sears. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud Computing (SoCC 2010)*. Indianapolis, Indiana, USA: ACM, June 2010, pp. 143–154. DOI: 10.1145/1807128.1807152.

[53] V. Costan and S. Devadas. "Intel SGX Explained". In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86.

[54] CryFS. *CryFS*. URL: https://www.cryfs.org/.

[55] DM-Crypt. *DM-Crypt*. URL: http://www.saout.de/misc/dm-crypt/.

[56] Cryptography. *pyca/cryptography*. URL: https://cryptography.io.

[57] Damien DeVille. *Dropbox and FUSE*. URL: https://blogs.dropbox.com/tech/2016/05/going-deeper-with-project-infinite.

[58] P. Damme, D. Habich, J. Hildebrandt and W. Lehner. "Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses)". In: *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017*. Venice, Italy, Mar. 2017, pp. 72–83. DOI: 10.5441/002/edbt.2017.08.

[59] H. Dang, E. Purwanto and E.-C. Chang. "Proofs of Data Residency: Checking whether Your Cloud Files Have Been Relocated". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2nd Apr. 2017, pp. 408–422. DOI: 10.1145/3052973.3053016.

[60] Q. H. Dang. *Secure Hash Standard (SHS)*. Tech. rep. Information Technology Laboratory, National Institute of Standards and Technology, 2015. DOI: 10.6028/NIST.FIPS.180-4.

[61] D.-T. Dang-Nguyen, C. Pasquini, V. Conotter and G. Boato. "RAISE: A Raw Images Dataset for Digital Image Forensics". In: *Proceedings of the 6th ACM Multimedia Systems Conference*. MMSys '15. Portland, Oregon: ACM, 2015, pp. 219–224. DOI: 10.1145/2713168.2713194.

[62] *Deduplicating and compressing storage - Red Hat Enterprise Linux 8*. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html-single/deduplicating_and_compressing_storage (visited on 13/07/2020).

[63] R. Denning and D. Elizabeth. *Cryptography and Data Security*. Boston, MA, USA: Addison-Wesley, 1982.

[64] L. P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951. RFC Editor, May 1996. URL: https://tools.ietf.org/rfc/rfc1951.

[65] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright and K. Ramchandran. "Network Coding for Distributed Storage Systems". In: *IEEE Transactions on Information Theory* 56.9 (2010), pp. 4539–4551.

[66] R. Dingledine, M. J. Freedman and D. Molnar. "The Free Haven Project: Distributed Anonymous Storage Service". In: *Designing Privacy Enhancing Technologies*. Springer. 2001, pp. 67–95.

[67] D. Dobre, P. Viotti and M. Vukolić. "Hybris: Robust Hybrid Cloud Storage". In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. ACM, Nov. 2014, 12:1–12:14. DOI: 10.1145/2670979.2670991.

[68] Docker. *Docker*. URL: https://www.docker.com/.

[69] Docker. *Docker and AUFS in practice*. URL: https://docs.docker.com/engine/userguide/storagedriver/aufs-driver.

[70] M. Docs. *How NTFS Works: Local File Systems*. 2009. URL: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc781134(v=ws.10)?redirectedfrom=MSDN.

[71] F. Douglis and A. Iyengar. "Application-specific Delta-encoding via Resemblance Detection". In: *USENIX Annual Technical Conference*. 2003, pp. 113–126.

[72] Dropbox. *Dropbox*. URL: https://www.dropbox.com.

[73] Dropbox. *If Dropbox receives legal requests for information*. URL: https://www.dropbox.com/help/security/legal-requests.

[74] DStat. *DStat*. URL: http://linux.die.net/man/1/dstat.

[75] A. J. Duncan, S. Creese and M. Goldsmith. "Insider Attacks in Cloud Computing". In: *IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. 2012.

[76] M. J. Dworkin. *SHA-3 standard: Permutation-based hash and extendable-output functions*. Tech. rep. 2015. DOI: 10.6028/NIST.FIPS.202.

[77] D. Eastlake and P. Jones. *US secure hash algorithm 1 (SHA1)*. RFC 3174. RFC Editor, Sept. 2001. URL: https://tools.ietf.org/rfc/rfc3174.

[78] I. Ecrypt. "ECRYPT II yearly report on algorithms and keysizes". In: *Available on http://www. ecrypt. eu. org* (2012), p. 5.

[79] eCryptfs. *eCryptfs*. URL: http://ecryptfs.org/.

[80] S. El Rouayheb, S. Goparaju, H. M. Kiah and O. Milenkovic. "Synchronizing edits in distributed storage networks". In: *IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2015, pp. 1472–1476.

[81] T. ElGamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". In: *Advances in cryptology*. Springer. 1984, pp. 10–18.

[82] *EncFS*. URL: https://github.com/vgough/encfs.

[83] T. Esteves, R. Macedo, A. Faria, B. Portela, J. Paulo, J. Pereira and D. Harnik. "TrustFS: An SGX-Enabled Stackable File System Framework". In: *2019 38th International Symposium on Reliable Distributed Systems Workshops (SRDSW)*. 2019 38th International Symposium on Reliable Distributed Systems Workshops (SRDSW). Oct. 2019, pp. 25–30. DOI: 10.1109/SRDSW49218.2019.00012.

[84] *European Commission Data Protection*. 2015. URL: http://ec.europa.eu/public_opinion/archives/ebs/ebs_431_en.pdf.

[85] facebook. *ZStandard source code*. original-date: 2015-01-24T00:22:38Z. URL: https://github.com/facebook/zstd (visited on 09/07/2020).

[86] *FileBench*. URL: https://github.com/filebench/filebench.

[87] M. Finck. *Blockchains and Data Protection in the European Union*. ID 3080322. Rochester, NY: Social Science Research Network, 30th Nov. 2017. DOI: 10.2139/ssrn.3080322.

[88] O. S. Foundation. *OpenSSL: Cryptography and SSL/TLS Toolkit*. visited: 28/09-2019. URL: https://www.openssl.org/.

[89] J. C. Frank, S. M. Frank, L. A. Thurlow, T. M. Kroeger, E. L. Miller and D. D. Long. "Percival: A searchable secret-split datastore". In: *31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE. 2015, pp. 1–12.

[90] S. Frankel, R. Glenn and S. Kelly. *The AES-CBC Cipher Algorithm and Its Use with IPsec*. RFC 3602. RFC Editor, Sept. 2003. URL: https://tools.ietf.org/rfc/rfc3602.

[91] Freenet. *Freenet*. URL: https://freenetproject.org.

[92] B. Furht. "A Survey of Multimedia Compression Techniques and Standards. Part I: JPEG Standard". In: *Real-Time Imaging* 1.1 (Apr. 1995), pp. 49–67. DOI: 10.1006/rtim.1995.1005.

[93] B. Furht. "A Survey of Multimedia Compression Techniques and Standards. Part II: Video Compression". In: *Real-Time Imaging* 1.5 (Nov. 1995), pp. 319–337. DOI: 10.1006/rtim.1995.1033.

[94] *FUSE*. URL: https://sourceforge.net/projects/fuse/.

[95] J.-l. Gailly and M. Adler. *Jean-loup Gailly and Mark Adler*. visited: 06/09-2019. URL: http://www.gzip.org/.

[96] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy and H. M. Levy. "Comet: An active distributed key-value store." In: *OSDI*. 2010, pp. 323–336.

[97] Gluster. *GlusterFS*. URL: https://github.com/gluster/glusterfs.

[98] Google Drive. *Google Drive*. URL: https://www.google.com/drive.

[99] *google/snappy*. original-date: 2014-03-03T21:58:09Z. URL: https://github.com/google/snappy (visited on 09/07/2020).

[100] P. Gopalan, C. Huang, H. Simitci and S. Yekhanin. "On the Locality of Codeword Symbols". In: *IEEE Transactions on Information Theory* 58.11 (Nov. 2012), pp. 6925–6934. DOI: 10.1109/TIT.2012.2208937.

[101] R. Gracia-Tinedo, P. García-López, M. Sánchez-Artigas, J. Sampé, Y. Moatti, E. Rom, D. Naor, R. Nou, T. Cortés, W. Oppermann and P. Michiardi. "IOStack: Software-Defined Object Storage". In: *IEEE Internet Computing* 20.3 (2016), pp. 10–18.

[102] T. Granlund, D. MacKenzie, P. Eggert and J. Meyering. *Torbjorn Granlund and David MacKenzie and Paul Eggert and Jim Meyering*. visited: 15/01-2020. URL: http://www.man7.org/linux/man-pages/man1/du.1.html.

[103] K. M. Greenan, X. Li and J. J. Wylie. "Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs". In: *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE. 2010.

[104] B. Gregg. *eBPF: One Small Step*. 15th May 2015. URL: http://www.brendangregg.com/blog/2015-05-15/ebpf-one-small-step.html (visited on 31/03/2020).

[105] V. Guruswami and M. Sudan. "Improved decoding of Reed-Solomon and algebraic-geometry codes". In: *IEEE Transactions on Information Theory* 45.6 (1999), pp. 1757–1767. DOI: 10.1109/18.782097.

[106] R. Halalai, P. Felber, A.-M. Kermarrec and F. Taïani. "Agar: A Caching System for Erasure-Coded Data". In: *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*. IEEE Computer Society, 2017, pp. 23–33. DOI: 10.1109/ICDCS.2017.97.

[107] M. A. Halcrow. "eCryptfs: An enterprise-class encrypted filesystem for linux". In: *Proceedings of the 2005 Linux Symposium*. Vol. 1. 2005, pp. 201–218.

[108] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. Anderson and D. Wetherall. "MetaSync: File Synchronization Across Multiple Untrusted Storage Services". In: *2015 USENIX Annual Technical Conference*. Santa Clara, CA, USA, 2015, pp. 83–95.

[109] S. Hand and T. Roscoe. "Mnemosyne: Peer-to-Peer Steganographic Storage". In: *1st International Workshop on Peer-to-Peer Systems (IPTPS)*. Springer. 2002, pp. 130–140.

[110] *Hash Tree Directories - ext4 Data Structures and Algorithms*. URL: `https://www.kernel.org/doc/html/latest/filesystems/ext4/dynamic.html#hash-tree-directories` (visited on 01/11/2020).

[111] J. S. Heidemann and G. J. Popek. "File-system Development with Stackable Layers". In: *ACM Trans. Comput. Syst.* 12.1 (Feb. 1994), pp. 58–89. DOI: 10.1145/174613.174616.

[112] S. Heron. "Advanced Encryption Standard (AES)". In: *Network Security* 2009.12 (2009), pp. 8–12.

[113] A. van Hoff. "Generic Diff Format Specification". In: *W3C* (1997).

[114] M. Honan. *How Apple and Amazon Security Flaws Led to My Epic Hacking*. 7th Aug. 2012. URL: `https://www.wired.com/2012/08/apple-amazon-mat-honan-hacking/`.

[115] S. M. Hosseini, D. Pratas and A. J. Pinho. "A Survey on Data Compression Methods for Biological Sequences". In: *Information* 7.4 (2016), p. 56. DOI: 10.3390/info7040056.

[116] C. Huang, M. Chen and J. Li. "Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems". In: *ACM Transactions on Storage (TOS)* 9.1 (Mar. 2013), 3:1–3:28. DOI: 10.1145/2435204.2435207.

[117] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li and S. Yekhanin. "Erasure coding in Windows Azure Storage". In: *2012 USENIX Annual Technical Conference*. Boston, MA, USA, 2012, pp. 15–26.

[118] D. A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (Sept. 1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[119] L.-D. Ibáñez, K. O'Hara and E. Simperl. "On Blockchains and the General Data Protection Regulation". In: (2018), p. 13.

[120] Intel SGX Sealing. *Intel SGX Sealing*. 2016. URL: `https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing`.

[121] I. ISA-L. *Intel ISA-L*. URL: `https://github.com/01org/isa-l`.

[122] S. Ishiguro, J. Murakami, Y. Oyama and O. Tatebe. "Optimizing Local File Accesses for FUSE-Based Distributed Storage". In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. Salt Lake City, UT, USA, 2012, pp. 760–765. DOI: 10.1109/SC.Companion.2012.104.

[123] A. Jain and K. I. Lakhtaria. "Comparative Study of Dictionary based Compression Algorithms on Text Data". In: *International Journal of Computer Science and Network Security (IJCSNS)* 16.2 (2016), p. 88.

[124] S. Jeong. *Anti-ISIS Hacktivists Are Attacking the Internet Archive*. 16th June 2016. URL: https://www.vice.com/en_us/article/3davzn/anti-isis-hacktivists-are-attacking-the-internet-archive.

[125] Jetico. *BestCRYPT*. URL: http://www.jetico.com/products/personal-privacy/bestcrypt-container-encryption.

[126] Joshua MacDonald. *Joshua MacDonald*. visited: 22/04-2020. 2019. URL: http://xdelta.org/.

[127] Joshua MacDonald. *Joshua MacDonald*. visited: 22/04-2020. 2019. URL: https://github.com/jmacd/xdelta.

[128] S. Kamara and K. Lauter. "Cryptographic Cloud Storage". In: *Proceedings of the 14th International Conference on Financial Cryptograpy and Data Security*. Springer-Verlag, 2010.

[129] D. Kaplan, J. Powell and T. Woller. *AMD Memory Encryption*. White Paper. Advanced Micro Devices, 2016. URL: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[130] R. Kaur, I. Chana and J. Bhattacharya. "Data deduplication techniques for efficient cloud storage management: a systematic review". In: *The Journal of Supercomputing* 74.5 (2018), pp. 2035–2085.

[131] O. Khan, R. C. Burns, J. S. Plank, W. Pierce and C. Huang. "Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads." In: *Proceedings of the 7th Conference on File and Storage Technologies*. FAST'12. USENIX Association, p. 20.

[132] O. Khan, R. C. Burns, J. S. Plank and C. Huang. "In Search of I/O-Optimal Recovery from Disk Failures." In: *HotStorage*. 2011.

[133] D. Kim, S. Song and B.-Y. Choi. "Existing Deduplication Techniques". In: *Data Deduplication for Data Optimization for Storage and Network Systems*. Cham: Springer International Publishing, 2017, pp. 23–76. DOI: 10.1007/978-3-319-42280-0_2.

[134] Z. A. Kissel. "Obfuscation of the Standard XOR Encryption Algorithm". In: *Crossroads* 11.3 (May 2005), pp. 6–6.

[135] O. Kolosov, G. Yadgar, M. Liram, I. Tamo and A. Barg. "On Fault Tolerance, Locality, and Optimality in Locally Repairable Codes". In: *2018 USENIX Annual Technical Conference*. Boston, MA, USA, 2018, pp. 865–877.

[136] D. Korn, J. MacDonald, J. Mogul and K. Vo. *The VCDIFF Generic Differencing and Compression Data Format*. RFC 3284. RFC Editor, June 2002. URL: https://tools.ietf.org/rfc/rfc3284.

[137] D. G. Korn and K.-P. Vo. "Engineering a Differencing and Compression Data Format." In: *USENIX annual technical conference, general track*. 2002, pp. 219–228.

[138] R. Kotla, L. Alvisi and M. Dahlin. "SafeStore: A Durable and Practical Storage System". In: *Proceedings of the 2007 USENIX Annual Technical Conference*. Santa Clara, CA, USA, 2007, pp. 129–142.

[139] H. Krawczyk. "Secret Sharing Made Short". In: *Advances in Cryptology — CRYPTO' 93*. Lecture Notes in Computer Science. Springer, 1994, pp. 136–146. DOI: 10.1007/3-540-48329-2_12.

[140] P. Kulkarni, F. Douglis, J. D. LaVoie and J. M. Tracey. "Redundancy Elimination Within Large Collections of Files". In: *USENIX Annual Technical Conference, General Track*. 2004, pp. 59–72. URL: http://dl.acm.org/citation.cfm?id=1247415.1247420.

[141] A. L'Hutereau, D. Burihabwa, P. Felber, H. Mercier and V. Schiavoni. "Blockchain-Based Metadata Protection for Archival Systems". In: *IEEE 38th Symposium on Reliable Distributed Systems (SRDS)*. Lyon, France, Oct. 2019, pp. 315–323. DOI: 10.1109/SRDS47363.2019.00044.

[142] L. Lamport, R. E. Shostak and M. C. Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (1982), pp. 382–401. DOI: 10.1145/357172.357176.

[143] G. G. Langdon. "Arithmetic coding". In: *IBM J. Res. Develop* 23 (1979), pp. 149–162.

[144] D. A. Lelewer and D. S. Hirschberg. "Data compression". In: *ACM Computing Surveys* 19.3 (Sept. 1987), pp. 261–296. DOI: 10.1145/45072.45074.

[145] *LessFS*. URL: http://www.lessfs.com/wordpress/.

[146] *LevelDB*. URL: http://leveldb.org/.

[147] J. Li, M. Krohn, D. Mazieres and D. Shasha. "SUNDR: Secure untrusted data repository". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2004.

[148] C. N. I. et Libertés (CNIL). *La Blockchain: quelles solutions pour un usage responsable en présence de données personnelles ?* Sept. 2018, p. 11.

[149] *LibFUSE*. URL: https://github.com/libfuse/libfuse.

[150] S. Lin and D. J. Costello. *Error Control Coding*. Second. Paerson Prentice Hall, 2004.

[151] *Linux Crypto API*. URL: https://kernel.org/doc/html/latest/crypto/.

[152] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom and M. Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[153] C. Lomont. *Chris Lomont*. visited: 25/09-2019. 2010. URL: https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions/.

[154] P. Lougher and R. Lougher. *SquashFS*. 2008.

[155] M. Luby and D. Zuckermank. *An xor-based erasure-resilient coding scheme*. Tech. rep. Tech Report, 1995.

[156] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman and V. Stemann. "Practical loss-resilient codes". In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM. 1997, pp. 150–159.

[157] *LZ4 source code*. original-date: 2014-03-25T15:52:21Z. URL: https://github.com/lz4/lz4 (visited on 09/07/2020).

[158] *LZMA SDK (Software Development Kit)*. URL: https://www.7-zip.org/sdk.html (visited on 09/07/2020).

[159] M. D. MacLaren. "The art of computer programming. Volume 2: Seminumerical algorithms (Donald E. Knuth)". In: *SIAM Review* 12.2 (1970), pp. 306–308.

[160] F. J. MacWilliams and N. J. A. Sloane. *The theory of error-correcting codes*. Elsevier, 1977.

[161] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin and M. Walfish. "Depot: Cloud Storage with Minimal Trust". In: *ACM Transactions on Computer Systems (TOCS)* (2011).

[162] J. Malhotra and J. Bakal. "A survey and comparative study of data deduplication techniques". In: *2015 International Conference on Pervasive Computing (ICPC)*. Jan. 2015, pp. 1–5. DOI: 10.1109/PERVASIVE.2015.7087116.

[163] N. Mandagere, P. Zhou, M. A. Smith and S. Uttamchandani. "Demystifying data deduplication". In: *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*. Companion '08. Leuven, Belgium: ACM, Dec. 2008, pp. 12–17. DOI: 10.1145/1462735.1462739.

[164] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. Rosenthal and M. Baker. "The LOCKSS peer-to-peer digital preservation system". In: *ACM Transactions on Computer Systems (TOCS)* 23 (2005), pp. 2–50.

[165] A. D. R. Marc Waldman and L. F. Cranor. "Publius: A robust, tamper-evident, censorship-resistant, web publishing system". In: *9th USENIX Security Symposium*. 2000, pp. 59–72.

[166] M. Matuska. *Martin Matuska*. visited: 15/01-2020. URL: https://www.freebsd.org/cgi/man.cgi?zfs(8).

[167] M. Matuska. *Martin Matuska*. visited: 15/01-2020. URL: https://www.freebsd.org/cgi/man.cgi?query=zpool&apropos=0&sektion=8&manpath=FreeBSD+12.1-RELEASE+and+Ports&arch=default&format=html.

[168] A. D. McDonald and M. G. Kuhn. "StegFS: A Steganographic File System for Linux". In: *Information Hiding: Third International Workshop, IH'99, Dresden, Germany, September 29 - October 1, 1999 Proceedings*. 2000, pp. 463–477.

[169] M. K. McKusick, W. N. Joy, S. J. Leffler and R. S. Fabry. "A fast file system for UNIX". In: *ACM Transactions on Computer Systems (TOCS)* 2.3 (1984), pp. 181–197.

[170] D. Meister, A. Brinkmann and T. Süß. "File recipe compression in data deduplication systems". In: *11th USENIX Conference on File and Storage Technologies (FAST13)*. 2013, pp. 175–182.

[171] R. Mendes, T. Oliveira, V. V. Cogo, N. F. Neves and A. N. Bessani. "CHARON: A Secure Cloud-of-Clouds System for Storing and Sharing Big Data". In: *IEEE Transactions on Cloud Computing* (May 2019). DOI: 10.1109/TCC.2019.2916856.

[172] H. Mercier, M. Augier and A. K. Lenstra. "STEP-archival: Storage Integrity and Anti-Tampering using Data Entanglement". In: *Proceedings of the 2015 International Symposium on Information Theory*. Extended version submitted to the IEEE Transactions on Information Theory. Hong Kong, 2015, pp. 1590–1594.

[173] H. Mercier, M. Augier and A. K. Lenstra. "STeP-archival: Storage integrity and anti-tampering using data entanglement". In: *IEEE Transactions on Information Theory* (2018). doi:10.1109/TIT.2018.2825981. Preliminary version in the proceedings of the IEEE International Symposium on Information Theory (ISIT), 2015, pp. 1590–1594.

[174] J. Metcalf. *GitHub Archive Program: the journey of the world's open source code to the Arctic*. 16th July 2020. URL: https://github.blog/2020-07-16-github-archive-program-the-journey-of-the-worlds-open-source-code-to-the-arctic/.

[175] *MetFS*. URL: http://www.enderunix.org/metfs/.

[176] D. T. Meyer and W. J. Bolosky. "A study of practical deduplication". In: *ACM Transactions Storage* 7 (2012), pp. 1–20. DOI: 10.1145/2078861.2078864.

[177] Microsoft. *EFS*. URL: https://technet.microsoft.com/en-us/library/bb457065.aspx.

[178] Minio. *Minio*. URL: https://minio.io.

[179] J. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff and D. Hellerstein. "Delta encoding in HTTP". In: 3229 (Jan. 2002), pp. 1–49. URL: https://tools.ietf.org/rfc/rfc3229.

[180] *Mountable HDFS*. URL: https://wiki.apache.org/hadoop/MountableHDFS.

[181] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss and F. Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*. 2020.

[182] C.-H. Ng, M. Ma, T.-Y. Wong, P. P. Lee and J. Lui. "Live deduplication storage of virtual machine images in an open-source cloud". In: *Proceedings of the 12th International Middleware Conference*. International Federation for Information Processing. 2011, pp. 80–99.

[183] L. Nielsen, D. Burihabwa, P. Felber, D. E. Lucani and V. Schiavoni. "MinervaFS: A User-Space File System for Generalized Deduplication". In: *Submitted to ACM/IFIP Middleware 2020*. June 2020.

[184] U. Niesen. "An Information-Theoretic Analysis of Deduplication". In: *IEEE ISIT*. 2017, pp. 1738–1742. DOI: 10.1109/ISIT.2017.8006827.

[185] NumPy. *NumPy*. URL: https://numpy.org.

[186] T. E. B. Observatory and Forum. *Blockchain and the GDPR: a thematic report prepared by the european union blockchain observatory and forum*. 16th Oct. 2018, p. 36. URL: https://www.eublockchainforum.eu/sites/default/files/reports/20181016_report_gdpr.pdf (visited on 03/03/2020).

[187] OceanStore. *OceanStore*. URL: http://www.oceanstore.net.

[188] F. E. Oggier and A. Datta. "Self-repairing codes - Local repairability for cheap and fast maintenance of erasure coded data". In: *Computing* 97.2 (2015), pp. 171–201.

[189] T. Okamoto and J. Stern. "Almost uniform density of power residues and the provable security of ESIGN". In: *Advances in Cryptology (ASIACRYPT)*. Springer, 2003.

[190] OneDrive. *OneDrive*. URL: https://onedrive.live.com.

[191] OpenStack. *Erasure Code API library written in C with pluggable Erasure Code backends*. URL: https://github.com/openstack/liberasurecode.

[192] OpenZFS. *OpenZFS*. visited: 19/06-2019. 2019. URL: http://open-zfs.org/wiki/Performance%5C_tuning%5C#Deduplication.

[193] OpenZFS. *OpenZFS*. visited: 19/06-2019. 2019. URL: http://open-zfs.org/wiki/Performance%5C_tuning%5C#Dataset%5C_recordsize.

[194] Oracle. *Oracle*. visited: 31/01-2019. 2019. URL: https://docs.oracle.com/cd/E23823%5C_01/html/819-5461/zfsover-2.html%5C#gayou.

[195] OSboxes. *OSboxes*. visited: 09/01-2020. URL: https://www.osboxes.org/vmware-images/.

[196] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. 1st ed. Springer-Verlag, 2010. DOI: 10.1007/978-3-642-04101-3.

[197] C. Padwick, M. Deskevich, F. Pacifici and S. Smallwood. "WorldView-2 pan-sharpening". In: *Proceedings of the ASPRS 2010 Annual Conference, San Diego, CA, USA*. Vol. 2630. 2010, pp. 1–14.

[198] D. S. Papailiopoulos and A. G. Dimakis. "Locally Repairable Codes". In: *IEEE Transactions on Information Theory* 60.10 (Oct. 2014), pp. 5843–5855. DOI: 10.1109/TIT.2014.2325570.

[199] J. Paulo and J. Pereira. "A survey and Classification of Storage Deduplication Systems". In: *ACM Computing Surveys* 47 (June 2014), p. 11. DOI: 10.1145/2611778.

[200] D. Phillips. "A Directory Index for Ext2". In: *Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5*. ALS '01. 2001, pp. 20–20. URL: http://dl.acm.org/citation.cfm?id=1268488.1268508.

[201] Planet Labs Inc. *Planet Labs Inc*. visited: 17/06-2019. 2019. URL: https://info.planet.com/download-free-high-resolution-skysat-image-samples/.

[202] J. S. Plank. "The raid-6 liber8tion code". In: *International Journal of High Performance Computing Applications* (2009).

[203] J. S. Plank, J. Luo, C. D. Schuman, L. Xu and Z. Wilcox-O'Hearn. "A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage." In: *7th USENIX Conference on File and Storage Technologies (FAST09)*. Berkeley, CA, USA, 2009, pp. 253–265.

[204] J. S. Plank, S. Simmerman and C. D. Schuman. *Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2*. Tech. rep. Technical Report CS-08-627, University of Tennessee, 2008.

[205] J. S. Plank and L. Xu. "Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications". In: *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on*. IEEE. 2006, pp. 173–180.

[206] J. S. Plank, K. M. Greenan and E. L. Miller. "Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions". In: *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*. San Jose, CA, 2013.

[207] R. Pontes, D. Burihabwa, F. Maia, J. Paulo, V. Schiavoni, P. Felber, H. Mercier and R. Oliveira. "SafeFS: a modular architecture for secure user-space file systems: one FUSE to rule them all". In: *Proceedings of the 10th ACM International Systems and Storage Conference*. Haifa, Israel: ACM, May 2017, p. 9. DOI: 10.1145/3078468.3078480.

[208] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang and L. Zhuang. "Enabling Security in Cloud Storage SLAs with CloudProof". In: *2011 USENIX Annual Technical Conference*. Portland, OR, USA, 2011, pp. 355–368.

[209] *PPDD*. URL: http://linux01.gwdg.de/~alatham/ppdd.html.

[210] *Proof of Authority Chains. The Parity Ethereum Documentation.* URL: https://github.com/paritytech/wiki/blob/master/Proof-of-Authority-Chains.md (visited on 28/02/2020).

[211] Publius. *Publius.* URL: http://www.cs.nyu.edu/~waldman/publius.

[212] PyECLib. *PyECLib.* URL: https://pypi.python.org/pypi/PyECLib.

[213] Python. *Python.* URL: https://www.python.org/.

[214] Qinlu He, Zhanhuai Li and Xiao Zhang. "Data deduplication techniques". In: *2010 International Conference on Future Information Technology and Management Engineering*. Vol. 1. Oct. 2010, pp. 430–433. DOI: 10.1109/FITME.2010.5656539.

[215] S. Quinlan and S. Dorward. "Venti: A New Approach to Archival Storage." In: *USENIX Conference on File and Storage Technologies (FAST02)*. 2002, pp. 89–102.

[216] M. O. Rabin. "Efficient dispersal of information for security, load balancing, and fault tolerance". In: *Journal of the ACM (JACM)* 36 (1989), pp. 335–348.

[217] M. O. Rabin. "Fingerprinting by random polynomials". In: *Technical report* (1981).

[218] A. Rajgarhia and A. Gehani. "Performance and extension of user space file systems". In: *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*. Sierre, Switzerland, 2010, pp. 206–213. DOI: 10.1145/1774088.1774130.

[219] Y. Rathore, M. K. Ahirwar and R. Pandey. "A Brief Study of Data Compression Algorithms". In: *International Journal of Computer Science and Information Security* 11 (Sept. 2013). URL: https://www.academia.edu/11814232/A_Brief_Study_of_Data_Compression_Algorithms (visited on 09/07/2020).

[220] M. A. Razzaque, C. Bleakley and S. Dobson. "Compression in wireless sensor networks: A survey and comparative evaluation". In: *ACM Transactions on Sensor Networks* 10.1 (Dec. 2013), 5:1–5:44. DOI: 10.1145/2528948.

[221] Redis. *Redis.* URL: http://redis.io.

[222] I. S. Reed and G. Solomon. "Polynomial codes over certain finite fields". In: *Journal of the society for industrial and applied mathematics* 8.2 (1960), pp. 300–304.

[223] D. Reinsel, J. Gantz and J. Rydning. *The Digitization of the World from Edge to Core*. Tech. rep. Nov. 2018, p. 28. URL: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

[224] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao and J. Kubiatowicz. "Pond: The OceanStore Prototype". In: *2nd USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, 2003, pp. 1–14.

[225] R. L. Rivest. *The MD5 Algorithm*. RFC 1321. RFC Editor, Apr. 1992, pp. 1–21. URL: http://www.rfc-editor.org/rfc/rfc1321.

[226] *RocksDB.* URL: https://rocksdb.org/.

[227] O. Rodeh and A. Teperman. "zFS - A Scalable Distributed File System Using Object Disks". In: *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, MSS 2003*. San Diego, CA, USA, 2003, pp. 207–218. DOI: 10.1109/MASS.2003.1194858.

[228] *S3FS.* URL: https://github.com/s3fs-fuse/s3fs-fuse.

[229] *S3QL.* URL: http://www.rath.org/s3ql-docs/.

[230] C. Sanchez-Avila and R. Sanchez-Reillol. "The Rijndael block cipher (AES proposal) : a comparison with DES". In: *IEEE 35th International Carnahan Conference on Security Technology*. 2001.

[231] M. T. Sandikkaya and A. E. Harmanci. "Security Problems of Platform-as-a-Service (PaaS) Clouds and Practical Solutions to the Problems". In: *IEEE 31st Symposium on Reliable Distributed Systems (SRDS)*. 2012.

[232] J. Sankaran. *Reed Solomon Decoder: TMS320C64x Implementation*. Tech. rep. Dec. 2000.

[233] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen and D. Borthakur. "XORing elephants: novel erasure codes for big data". In: *Proceedings of the 39th international conference on Very Large Data Bases*. 2013, pp. 325–336.

[234] S. van Schaik, A. Kwong, D. Genkin and Y. Yarom. *van Schaik, Stephan and Kwong, Andrew and Genkin, Daniel and Yarom, Yuval*. 2020. URL: `https://sgaxeattack.com/`.

[235] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos and C. Giuffrida. "RIDL: Rogue In-Flight Data Load". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP). May 2019, pp. 88–105. DOI: `10.1109/SP.2019.00087`.

[236] S. van Schaik, M. Minkin, A. Kwong, D. Genkin and Y. Yarom. *van Schaik, Stephan and Minkin, Marina and Kwong, Andrew and Genkin, Daniel and Yarom, Yuval*. 2020. URL: `https://cacheoutattack.com/`.

[237] B. Schneier. "Description of a new variable-length key, 64-bit block cipher (Blowfish)". In: *International Workshop on Fast Software Encryption*. Springer. 1993, pp. 191–204.

[238] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher and D. Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *CCS*. 2019.

[239] M. Schwarz, S. Weiser, D. Gruss, C. Maurice and S. Mangard. "Malware guard extension: Using SGX to conceal cache attacks". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2017, pp. 3–24.

[240] A. Shamir. "How to Share a Secret". In: *Communincations of ACM* 22.11 (1979), pp. 612–613.

[241] S. Shanmugasundaram and R. Lourdusamy. "A comparative study of text compression algorithms". In: *International Journal of Wisdom Based Computing* 1.3 (2011), pp. 68–76.

[242] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li and S. Sengupta. "Primary Data Deduplication—Large Scale Study and System Design". In: *USENIX Annual Tech Conference*. 2012, pp. 285–296.

[243] Y. Shin, D. Koo and J. Hur. "A Survey of Secure Data Deduplication Schemes for Cloud Storage Systems". In: *ACM Computing Surveys* 49.4 (Jan. 2017), 74:1–74:38. DOI: `10.1145/3017428`.

[244] K. Shvachko, H. Kuang, S. Radia and R. Chansler. "The Hadoop Distributed File System". In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010, pp. 1–10.

[245] R. Singel. *Dropbox Lied to Users About Data Security, Complaint to FTC Alleges*. 13th May 2011. URL: https://www.wired.com/2011/05/dropbox-ftc/.

[246] *Software Updates: Courgette - The Chromium Projects*. URL: https://dev.chromium.org/developers/design-documents/software-updates-courgette (visited on 10/07/2020).

[247] *SshFS*. URL: https://github.com/libfuse/sshfs.

[248] N. Standard. "Data Encryption Standard (DES)". In: *Federal Information Processing Standards Publication* (1999).

[249] I. Stefanovici, B. Schroeder, G. O'Shea and E. Thereska. "sRoute: Treating the Storage Stack Like a Network". In: *Proceedings of the 14th USENIX Conference on File and Storage Technologies*. 2016, pp. 197–212.

[250] M. W. Storer, K. M. Greenan, E. L. Miller and K. Voruganti. "POTSHARDS: Secure Long-Term Storage Without Encryption". In: *Proceedings of the 2007 USENIX Annual Technical Conference*. Santa Clara, CA, USA, 2007, pp. 142–156.

[251] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek and R. Morris. "Flexible, Wide-area Storage for Distributed Systems with WheelFS". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. 2009, pp. 43–58.

[252] A. Stubblefield and D. S. Wallach. *Dagster: Censorship resistant publishing without replication*. Tech. rep. TR01-380. Rice University, July 2001.

[253] S. Sundararaman, L. Visampalli, A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. "Refuse to crash with Re-FUSE". In: *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011*. Salzburg, Austria, 2011, pp. 77–90. DOI: 10.1145/1966445.1966453.

[254] Swisscom. *Accidental Data Deletion in myCloud*. URL: https://www.swisscom.ch/en/about/news/2019/07/faktencheck-mycloud.html.

[255] N. Szabo. *Smart Contracts*. URL: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html (visited on 28/02/2020).

[256] I. Tamo and A. Barg. "A Family of Optimal Locally Recoverable Codes". In: *IEEE Transactions on Information Theory* 60.8 (Aug. 2014), pp. 4661–4676. ISSN: 0018-9448. DOI: 10.1109/TIT.2014.2321280.

[257] H. Tang, F. Liu, G. Shen, Y. Jin and C. Guo. "UniDrive: Synergize Multiple Consumer Cloud Storage Services". In: *Proceedings of the 16th Annual Middleware Conference (Middleware)*. 2015.

[258] *Tar - GNU Project - Free Software Foundation*. URL: https://www.gnu.org/software/tar/ (visited on 09/07/2020).

[259] V. Tarasov, A. Gupta, K. Sourav, S. Trehan and E. Zadok. "Terra Incognita: On the Practicality of User-Space File Systems". In: *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*. 2015.

[260] V. Tarasov, E. Zadok and S. Shepler. *Vasily Tarasov and Erez Zadok and Spencer Shepler*. visited: 13/09-2019. 2010. URL: https://github.com/filebench/filebench.

[261] V. Tarasov, E. Zadok and S. Shepler. "Filebench: A flexible framework for file system benchmarking". In: *login: The USENIX Magazine* 41.1 (2016), pp. 6–12.

[262] The Apache Software Foundation. *ab - Apache HTTP server benchmarking tool*. URL: `https://httpd.apache.org/docs/2.4/programs/ab.html`.

[263] The Apache Software Foundation. *Apache Zookeeper*. URL: `https://zookeeper.apache.org`.

[264] The Internet Archive. *Internet Archive: Digital Library of Free & Borrowable Books, Movies, Music & Wayback Machine*. URL: `https://archive.org`.

[265] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black and T. Zhu. "IOFlow: A Software-defined Storage Architecture". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 182–196.

[266] *Tuning KVM*. URL: `http://www.linux-kvm.org/page/Tuning_KVM`.

[267] Tuxera. *Open Source: NTFS-3G driver software and community*. URL: `https://www.tuxera.com/community/open-source-ntfs-3g/`.

[268] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki and A. Bohra. "HydraFS: A High-throughput File System for the HYDRAstor Content-addressable Storage System". In: *Proceedings of the 8th USENIX Conference on File and Storage Technologies*. 2010, pp. 17–17.

[269] uWSGI. *uWSGI*. URL: `https://uwsgi-docs.readthedocs.io/en/latest/`.

[270] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom and R. Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, 991–1008.

[271] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss and F. Piessens. "LVI: Hijacking transient execution through microarchitectural load value injection". In: *41th IEEE Symposium on Security and Privacy (S&P'20)*. 2020, pp. 1399–1417.

[272] B. K. R. Vangoor, V. Tarasov and E. Zadok. "To FUSE or Not to FUSE: Performance of User-Space File Systems". In: *Proceedings of the 15th USENIX Conference on File and Storage Technologies*. 2017, pp. 59–72.

[273] R. Vestergaard, D. E. Lucani and Q. Zhang. "Generalized Deduplication: Lossless Compression for Large Amounts of Small IoT Data". In: *European Wireless Conf.* Aarhus, Denmark, May 2019.

[274] R. Vestergaard, Q. Zhang and D. E. Lucani. "Lossless Compression of Time Series Data with Generalized Deduplication". In: *IEEE GLOBECOM*. Waikoloa, USA, Dec. 2019.

[275] R. Vestergaard, Q. Zhang and D. E. Lucani. "Generalized Deduplication: Bounds, Convergence, and Asymptotic Properties". In: *IEEE GLOBECOM*. Waikoloa, USA, Dec. 2019. arXiv: `1901.02720`.

[276] W. Vogels. "Beyond Server Consolidation". In: *ACM Queue* 6 (2008), pp. 20–26.

[277] P. Voigt and A. Von dem Bussche. "The EU General Data Protection Regulation (GDPR)". In: *A Practical Guide, 1st Ed., Cham: Springer International Publishing* (2017).

[278] M. Vrable, S. Savage and G. M. Voelker. "BlueSky: A cloud-backed file system for the enterprise". In: *Proceedings of the 10th USENIX Conference on File and Storage Technologies*. 2012, pp. 19–19.

[279] M. Waldman and D. Mazières. "Tangler: A Censorship-resistant Publishing System Based on Document Entanglements". In: *Proceedings of the 8th ACM Conference on Computer and Communications Security (CCS)*. Philadelphia, PA, USA, 2001.

[280] S. R. Walli. "The POSIX Family of Standards". In: *StandardView* 3.1 (1995), pp. 11–17.

[281] H. Weatherspoon and J. D. Kubiatowicz. "Erasure Coding Vs. Replication: A Quantitative Comparison". In: *Peer-to-Peer Systems*. Lecture Notes in Computer Science. Springer, 2002, pp. 328–337. DOI: 10.1007/3-540-45748-8_31.

[282] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long and C. Maltzahn. "Ceph: A Scalable, High-Performance Distributed File System". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pp. 307–320.

[283] T. A. Welch. "A technique for high-performance data compression". In: *Computer* 6 (1984). Publisher: IEEE, pp. 8–19.

[284] R. Wesley. "Error detecting and error correcting codes". In: *The Bell System Technical Journal* 29.2 (Apr. 1950), pp. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[285] S. B. Wicker and V. K. Bhargava. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[286] Wikipedia Dump. *Wikipedia Dump*. visited: 14/01-2020. URL: https://dumps.wikimedia.org/other/static_html_dumps/current/scn/.

[287] C. K. Wong and S. S. Lam. "Digital signatures for flows and multicasts". In: *IEEE/ACM Transactions on Networking* 7 (Aug. 1999), pp. 502–513.

[288] G. Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum Project Yellow Paper* 151 (2014). URL: http://gavwood.com/paper.pdf (visited on 21/03/2017).

[289] C. P. Wright, M. C. Martino and E. Zadok. "NCryptfs: A Secure and Convenient Cryptographic File System". In: *Proceedings of the General Track: 2003 USENIX Annual Technical Conference*. 2003, pp. 197–210.

[290] WSGI. *WSGI*. URL: http://www.wsgi.org.

[291] M. Xia, M. Saxena, M. Blaum and D. A. Pease. "A Tale of Two Erasure Codes in HDFS". In: *13th USENIX Conference on File and Storage Technologies (FAST'15)*. 2015, pp. 213–226.

[292] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang and Y. Zhou. "A Comprehensive Study of the Past, Present, and Future of Data Deduplication". In: *Proceedings of the IEEE* 104.9 (Sept. 2016), pp. 1681–1710. DOI: 10.1109/JPROC.2016.2571298.

[293] W. Xu, L. Huang, A. Fox, D. Patterson and M. I. Jordan. "Detecting Large-scale System Problems by Mining Console Logs". In: *ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP09)*. ACM, 2009, pp. 117–132. DOI: 10.1145/1629575.1629587.

[294] W. Yan, J. Yao and Q. Cao. "Defuse: Decoupling Metadata and Data Processing in FUSE Framework for Performance Improvement". In: *IEEE Access* 7 (2019), pp. 138473–138484. DOI: 10.1109/ACCESS.2019.2942954.

[295] L. L. You, K. T. Pollack and D. D. Long. "Deep Store: an archival storage system architecture". In: *21st International Conference On Data Engineering (ICDE)*. IEEE. 2005, pp. 804–815.

[296] L. L. You, K. T. Pollack, D. D. Long and K. Gopinath. "PRESIDIO: A Framework for Efficient Archival Data Storage". In: *ACM Transactions on Storage (TOS)* 7 (2011), 6:1–6:60.

[297] E. Zadok, J. M. Andersen, I. Badulescu and J. Nieh. "Fast Indexing: Support for Size-Changing Algorithms in Stackable File Systems". In: *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA, 2001, pp. 289–304.

[298] E. Zadok, I. Badulescu and A. Shender. *CryptFS: A stackable vnode level encryption file system*. Tech. rep. Technical Report CUCS-021-98, Computer Science Department, Columbia University, 1998.

[299] E. Zadok, I. Badulescu and A. Shender. "Extending File Systems Using Stackable Templates". In: *Proceedings of the 1999 USENIX Annual Technical Conference*. Monterey, CA, USA, 1999, pp. 57–70.

[300] E. Zadok and J. Nieh. "FiST: A Language for Stackable File Systems". In: *Proceedings of the General Track: 2000 USENIX Annual Technical Conference*. San Diego, CA, USA, 2000, pp. 55–70.

[301] D. Zhang, Q. Liu, Y. Wu, Y. Li and L. Xiao. "Compression and Indexing Based on BWT: A Survey". In: *10th Web Information System and Application Conference*. Nov. 2013, pp. 61–64. DOI: 10.1109/WISA.2013.20.

[302] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. "End-to-end Data Integrity for File Systems: A ZFS Case Study." In: *8th USENIX Conference on File and Storage Technologies (FAST'10)*. San Jose, CA, USA, Feb. 2010, pp. 29–42.

[303] Z. Zhang, X. Li and X. Li. "Study on lossless data compression based on embedded system". In: *2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA)*. Changsha, China, Sept. 2010, pp. 1225–1230. DOI: 10.1109/BICTA.2010.5645148.

[304] G. Zhao, C. Rong, J. Li, F. Zhang and Y. Tang. "Trusted Data Sharing over Untrusted Cloud Storage Providers". In: *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CLOUDCOM)*. Indianapolis, IN, USA, Nov. 2010, pp. 97–103. DOI: 10.1109/CloudCom.2010.36.

[305] R. Zhao, C. Yue, B. Tak and C. Tang. "SafeSky: A Secure Cloud Storage Middleware for End-User Applications". In: *IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. Montreal, QC, Canada, Sept. 2015. DOI: 10.1109/SRDS.2015.23.

[306] B. Zhu, K. Li and R. H. Patterson. "Avoiding the Disk Bottleneck in the Data Domain Deduplication File System". In: *6th USENIX Conference on File and Storage Technologies (FAST'08)*. San Jose, CA, USA, Feb. 2008, pp. 1–14.

[307] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng and M. R. Lyu. "Tools and Benchmarks for Automated Log Parsing". In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 10)*. May 2019, pp. 121–130. DOI: 10.1109/ICSE-SEIP.2019.00021.

[308] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 23.3 (May 1977), pp. 337–343. DOI: 10.1109/TIT.1977.1055714.

[309] J. Ziv and A. Lempel. "Compression of individual sequences via variable-rate coding". In: *IEEE Transactions on Information Theory* 24.5 (Sept. 1978), pp. 530–536. DOI: 10.1109/TIT.1978.1055934.