Data Security and Reliability in Cloud Backup Systems with Deduplication

RAHUMED, Arthur

A Thesis Submitted in Partial Fulfilment

of the Requirements for the Degree of

Master of Philosophy

in

Computer Science and Engineering

The Chinese University of Hong Kong

September 2012

Abstract

Cloud storage is an emerging service model that enables individuals and enterprises to outsource the storage of data backups to remote cloud providers at a low cost. This thesis presents methods to ensure the data security and reliability of cloud backup systems.

In the first part of this thesis, we present FadeVersion, a secure cloud backup system that serves as a security layer on top of todays cloud storage services. FadeVersion follows the standard version-controlled backup design, which eliminates the storage of redundant data across different versions of backups. On top of this, FadeVersion applies cryptographic protection to data backups. Specifically, it enables fine-grained assured deletion, that is, cloud clients can assuredly delete particular backup versions or files on the cloud and make them permanently inaccessible to anyone, while other versions that share the common data of the deleted versions or files will remain unaffected. We implement a proof-of-concept prototype of FadeVersion and conduct empirical evaluation atop Amazon S3. We show that FadeVersion only adds minimal performance overhead over a traditional cloud backup service that does not support assured deletion.

In the second part of this thesis, we present *CFTDedup*, a distributed proxy system designed for providing storage efficiency via deduplication in cloud storage, while ensuring crash fault tolerance among proxies. It synchronizes deduplication metadata among proxies to provide strong consistency. It also batches metadata updates to mitigate synchronization overhead. We implement a preliminary prototype of CFTDedup and evaluate via testbed experiments its runtime performance in deduplication storage for virtual machine images. We also discuss several open issues on how to provide reliable, high-performance deduplication storage. Our CFTDedup prototype provides a platform to explore such issues.

摘要

雲存儲是一個新興的服務模式，讓個人和企業的數據備份外包予較低成本的遠程雲服務提供商。 本論文提出的方法，以確保數據的安全性和雲備份系統的可靠性。

在本論文的第一部分，我們提出 FadeVersion， 安全的雲備份作為今天的雲存儲服務上的安全層服務的系統。 FadeVersion 實現標準的版本控制備份設計，從而消除跨不同版本備份的冗餘數據存儲。 此外，FadeVersion 在此設計上加入了加密技術以保護備份。 具體來説，它實現細粒度安全刪除，那就是， 雲客戶可以穩妥地在雲上刪除特定的備份版本或文件，使有關文件永久無法被解讀， 而其它共用被刪除數據的備份版本或文件將不受影響。 我們實現了試驗性原型的 FadeVersion 並在亞馬遜S3之上進行實證評價。 我們證明了，相對於不支援度安全刪除技術傳統的雲備份服務 FadeVersion 只增加小量額外開鎖。

在本論文的第二部分，提出 *CFTDedup* 一個分佈式代理系統， 利用通過重複數據刪除增加雲存儲的效率，而同時確保代理之間的崩潰容錯。 代理之間會進行同步以保持重複數據刪除元數據的一致性。 另外，它也分批更新元數據減輕同步帶來的開銷。 我們實現了初步的原型CFTDedup並通過試驗台試驗， 以存儲虛擬機映像評估其重複數據刪除的運行性能。我們還討論了幾個開放問題， 例如如何提供可靠、高性能的重複數據刪除的存儲。我們的CFTDedup原型提供了一個平台來探討這些問題。

Acknowledgments

Special thanks to Patrick P. C. Lee, John C.S. Lui and Henry C. H. Chen

Contents

List of Publications

- *Arthur Rahumed*, and Patrick P. C. Lee, A Proxy-Based Deduplication Storage System with Crash Fault Tolerance, Submitted to USENIX 8th Workshop on Hot Topics in System Dependability

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing is an emerging service model that provides computation and storage resources on the Internet. One attractive functionality that cloud computing can offer is *cloud storage*. Individuals and enterprises are often required to *remotely archive* their data to avoid any information loss in case there are any hardware/software failures or unforeseen disasters. Instead of purchasing the needed storage media to keep data backups, individuals and enterprises can simply *outsource* their data backup services to the cloud service providers, which provide the necessary storage resources to host the data backups.

This thesis focuses on the problem of assured deletion and crash fault tolerance of cloud backup systems. We first propose a system called FadeVersion to introduce assured deletion into cloud backup systems and then later explore adding crash fault tolerance by proposing another system CFTDedup. However, due to time constraints, we were not able to add assured deletion to CFTDedup.

1.1 Cloud Based Backup and Assured Deletion

While cloud storage is attractive, how to provide security guarantees for outsourced data becomes a rising concern. One major security challenge is to provide the property of *assured deletion*, i.e., data files are permanently inaccessible upon requests of deletion.

Keeping data backups permanently is undesirable, as sensitive information may be exposed in the future because of data breach or erroneous management of cloud operators. Thus, to avoid liabilities, enterprises and government agencies usually keep their backups for a finite number of years and request to delete (or destroy) the backups afterwards. For example, the US Congress is formulating the Internet Data Retention legislation in asking ISPs to retain data for two years [1], while in United Kingdom, companies are required to retain wages and salary records for six years [2].

Assured deletion aims to provide cloud clients an option of reliably destroying their data backups upon requests. On the other hand, cloud providers may replicate multiple copies of data over the cloud infrastructure for fault-tolerance reasons. Since cloud providers do not publicize their replication policies, cloud clients do not know how many copies of their data are on the cloud, or where these copies are located. It is unclear whether cloud providers can reliably remove all replicated copies when cloud clients issue requests of deletion for their outsourced data.

Therefore the design of a highly secure cloud backup system that enables assured deletion for outsourced data backups on the cloud, while addressing the important features for a typical backup application is desirable. One such feature is to enable *version control* for outsourced data backups, so that cloud clients can roll-back to extract data from earlier versions. Typically, each backup version is incrementally built from the previous version. If the same file appears in multiple versions, then it is natural to store only one copy of the file and have the different versions refer to the file copy. However, there are data dependencies across different versions, and deleting an old version may make the future versions unrecoverable.

In the first part of this thesis, we introduce *FadeVersion* a work in collaboration with Henry Chen Chuk Hin. *FadeVersion* secure cloud backup system that supports both *version control* and *assured deletion*. FadeVersion allows fine-grained assured deletion, such that cloud clients can specify particular versions or files on the cloud to be assuredly deleted, while other versions that share the common data of the deleted versions or files will remain unaffected. The main idea of FadeVersion is to use a *layered encryption* approach. Suppose that a file F appears in multiple versions. We first encrypt F with key k, and then encrypt key k independently with different keys associated with different versions. Thus, if we remove a key of one version, we can still recover key k and hence file F in another version.

The result was a proof-of-concept prototype of FadeVersion that is compatible with today's cloud storage services. We extend an open-source cloud backup system Cumulus[3] and include the assured deletion feature. Using Amazon S3 as the cloud storage backend, we empirically evaluated the performance of FadeVersion. We also conducted economical cost analysis for FadeVersion based on the cost plans of different cloud providers. We showed that the additional overhead of FadeVersion is justifiable compared to a traditional cloud backup service that does not possess the assured deletion functionality.

My main contribution to the project was modification of the Cumulus backup program to support assured deletion, while my partner Henry Chen was responsible modifying the Cumulus backup restore program to support for the new backup format generated by FadeVersion.

1.2 Crash Fault Tolerance for Backup Systems with Deduplication

FadeVersion implements versioning via the technique of deduplication. *Deduplication* [4] is a technique that has been widely used to achieve efficient storage. It operates by splitting data into blocks and computing cryptographic hashes on the content for each block. If two blocks have identical content, then they have the same hash and hence only one physical copy of the block needs to be stored.

From a client's perspective, enabling deduplication in cloud storage can be achieved via a *proxy-based* approach, in which a proxy (or gateway) serves as an interface between different clients' data sources and cloud storage sites. The proxy applies deduplication to clients' data before uploading the unique data blocks to the cloud, and also keeps deduplication metadata to determine if a data block can be deduplicated. Each client only needs to interface with the proxy on how to upload/download data, without needing to implement the deduplication logic. The proxy-based approach has been used and analyzed in many cloud storage systems in business (e.g., [5, 6, 7, 8]) and academia (e.g., [9, 10, 11, 12]).

Deploying a single proxy is clearly prone to the performance bottleneck and *single-point-of-failure* problems. If a proxy fails, the upload procedure will have to be restarted from the beginning with another proxy, extending the period where computer systems are taken offline to complete the backup. Furthermore, single proxy systems which perform deduplication will lose their deduplication metadata which has to be recovered; otherwise the system will lose the ability to deduplicate the data it is currently backing up with data previously backed up.

Thus, we consider a *distributed proxy system*, in which multiple independent proxies coordinate among themselves in the deduplication process. If a proxy crashes, clients can re-connect to the cloud via a different proxy, and it is desirable that any subsequently uploaded data blocks can still be deduplicated with the previously uploaded blocks before the crash so as to preserve storage efficiency. On the other hand, maintaining consistent deduplication metadata across multiple proxies is challenging. Our observation is that deduplication is performed on a per-block basis, and hence updating deduplication metadata on each data block across all proxies can introduce significant synchronization overhead. This motivates us to explore the performance trade-off in designing a fault-tolerant distributed proxy system for deduplication storage.

In the second part of this thesis, we propose *CFTDedup*, a distributed proxy system designed for deduplication storage with crash fault tolerance (CFT). Our current CFTDedup design considers one extreme of consistency known as *strong consistency*, such that any updates to the deduplication metadata are fully serialized, replicated, and synchronized across all proxies. All proxies share the same view of the deduplication metadata. Any identical data blocks that are uploaded through different proxies can still be deduplicated with one another. In particular, we propose to aggregate data blocks into segments, and *batch* the updates of deduplication metadata to mitigate synchronization overhead.

We implement a preliminary prototype of CFTDedup, and evaluate via testbed experiments its performance in storing virtual machine images. We show that CFTDedup provides storage efficiency via deduplication, while ensuring fault tolerance against proxy and client crashes. We believe that our CFTDedup prototype provides a platform for future research. In view of this, we conclude by discussing several open issues on how to provide reliable, high-performance deduplication storage.

1.3 Outline of Thesis

The remainder of the report proceeds as follows. Chapter 2 provides the necessary background on deduplication, fault tolerance, and issues in integrating assured deletion version control. Chapters 3, 4, 5, discusses the threat model, design, implementation details of FadeVersion, and also the effectiveness and performance overhead of FadeVersion respectively. Chapter 6 describes the design and implementation of CFTDedup, and chapter 7 discusses the performance of CFTDedup. Furthermore we discuss design tradeoffs in CFTDedup and future work in chapter 8. Finally, we conclude the thesis in chapter 9.

Chapter 2

Background and Related Work

In this chapter, I will describe previous work related to FadeVersion and CFTDedup.

2.1 Deduplication

Deduplication refers to the removal of duplicate data[13], which can be used to reduce storage space and data transmission. It is performed by hashing the data of interest, and then looking up a deduplication index to determine if the data has been encountered before, if not, the system will update the deduplication index to include the hash of the data. The deduplication index can be implemented using a hash table in memory [14] or a database on the hard drive[13]. One notable example of a deduplication system is DropBox[15], a web based file hosting service[16].

2.2 Assured Deletion

There are different ways of achieving assured deletion. One approach is by *secure overwriting* [17], in which new data is written over original data to make the original data unrecoverable. Secure overwriting has also been applied in versioning file systems [18]. However, this requires internal modifications of a file system and is not feasible for outsourced storage, since the storage backends are maintained by third parties, and it has no guarantee that replicated data will be over-written.

Another approach is achieved by *cryptographic protection*, which removes the cryptographic keys that are used to decrypt data blocks to make the encrypted blocks unrecoverable [19, 20, 21, 22, 23, 24]. The encrypted data blocks are stored in outsourced storage (e.g., clouds), while the cryptographic keys are kept independently by a trusted entity also known as a *key escrow system.* Conversely, to retrieve the data, the users can ask the key escrow for the key in order to decrypt data downloaded from the cloud. The key escrow system ensures that the key is only accessible to authorized persons but not to the cloud operator, and is assumed to securely erase the key when data is deleted.

2.3 Policy Based Assured Deletion

The previously mentioned approach can be extended to support *policy based deletion*, where each piece of data is assigned a policy and deleted when the policy is revoked. Under such a scheme, each piece of data is still encrypted with a uniquely generated key (known as data keys). However, instead of storing keys for each individual piece of data in the key escrow, the key escrow only stores a limited number of keys, each of it associated with a single policy (known as control keys), which is then used to encrypt the data keys. The encrypted data key is stored along with the encrypted data, and decrypted by the key escrow during data retrieval. When a policy is revoked, the associated control key is securely erased, causing all data keys encrypted by the control key, and by extension the data encrypted by those data keys irretrievable. The scheme was proposed by Ephemerizer[25] as a means to reduce the number of keys stored by the key escrow to simplify key management.

FADE[23] further extends the previous idea by allowing any combination of policies to be applied on a single piece of data. For example, we may want to keep data, even if one of many policies applied is revoked, or we may want to delete data if any one of many policies is revoked. In the former case, we can create multiple copies of the data key and encrypt each with different control keys such that revocation of a single policy only removes access to only one copy of the data key, while allowing access to other copies via different policies. For the latter case, we can encrypt a single copy of the data key with each of control keys, and the data key cannot be decrypted with revocation of any policies. By applying a combination of the above two, i.e. multiple copies of data keys each encrypted multiple times by different groups of control keys, FADE is able to handle any combination of policies to be applied on data.

We note that existing studies for secure erasure does not consider the issue of version control. In Chapter 3.2, we show that existing version control systems and assured deletion systems are incompatible with each other.

2.4 Convergent Encryption

Version control follows the notion of *deduplication* [4], which eliminates the storage of redundant data chunks that have the same content. In the security context, recent studies propose *convergent encryption* [26, 27], such that the key for encrypting/decrypting a data chunk is a function of the content of the data chunk, so that the encryptions of two redundant data chunks will still return the same content. However, in convergent encryption, if we want to assuredly delete a data chunk of a particular version, we cannot simply remove its associated key, since it may make the identical chunks in other versions unrecoverable.

2.5 Cloud Based Backup Systems

There are a few cloud backup systems in the market. Examples include commercial systems like Dropbox [15], Jungle Disk [28], and Nasuni [6], as well as the open-source Cumulus system [3], all of which provide version control and archive different versions of backups. Specifically, Cumulus considers a *thin cloud* interface, meaning that the cloud only provides basic functionalities for outsourced storage, such as `put`, `get`, `list`, and `delete`[1] . It splits a file into chunks, and only modified chunks will be uploaded to the cloud. New versions may refer to the identical chunks in older versions, so no redundant chunks across versions will be stored. Note that Cumulus does not provide assured deletion.

In March 2011, Nasuni announced that its system enables the new snapshot retention policy that allows assured deletion of backup snapshots [29]. On the other hand, there is no formal study about their implementation methodologies and performance evaluation. We address this issue in the first part of this thesis. We provide a comprehensive study that describes the design details of how to integrate assured deletion into a general version-controlled system with deduplication. We also provide extensive empirical evaluation and monetary cost analysis for our design.

2.6 Fault Tolerant Deduplication Systems

Deduplication was first proposed in Venti [4] as a means to eliminate the storage of redundant blocks in archival storage. Data Domain [30] and Foundation [31] are proposed to improve the archival throughput of Venti via new deduplication indexing techniques. Such systems are centralized. They mainly focus on improving the deduplication performance without considering fault

[1]The `delete` operation only requests the cloud to remove the physical copy of a file, but there is no guarantee that the file is assuredly deleted.

tolerance.

Our work in the second part of this thesis studies deduplication in a distributed setting, which is also considered in the literature. Farsite [32] and Pastiche [33] are distributed systems that apply file-level deduplication and block-level deduplication, respectively. Hydrastor [34] implements decentralized, fault-tolerant deduplication for archival storage. It distributes both deduplication metadata and archival data in a distributed hash table. DeDe [35] implements fault-tolerant deduplication for virtual machine images in a decentralized SAN environment where no centralized metadata server is required. Note that DeDe applies offline deduplication (i.e., deduplication after writes), while we apply inline deduplication when data is about to be uploaded to the cloud (see Chapter 6.3 for details). Extreme Binning [36] and MAD2 [37] propose various performance optimization techniques to achieve high deduplication throughput for distributed storage. Our work in the second part of this thesis has a different design space from the above work, as we consider a distributed proxy system for cloud storage. Our objective is to maintain deduplication metadata in a fault-tolerant proxy system, while relying on the cloud infrastructure to provide high-availability storage for data.

Proxy-based solutions for cloud storage have been proposed in commercial solutions (e.g., Amazon's AWS Storage Gateway [5], Nasuni [6], Panzura [7], StorSimple [8]) and academic projects (e.g., RACS [9], DepSky [10], BlueSky [12], and NCCloud [11]). Commercial solutions provide limited implementation details, so it is difficult to evaluate their performance in maintaining fault tolerance. For academic projects, NCCloud [11] and BlueSky [12] use a single-proxy design. RACS [9] considers a distributed proxy system and uses Zookeeper-based [38] distributed locks to synchronize the states among multiple proxies. DepSky [10] uses low-contention file locks to support multiple writes on the same file. Our work focuses on evaluating the performance-consistency trade-off in the deduplication context.

Chapter 3

Design of FadeVersion

In this chapter, we present the design of FadeVersion, a secure cloud backup system that works seamlessly with today's cloud storage services such as Amazon S3. It is a client-side system which integrates *both* version control and assured deletion.

3.1 Threat Model and Assumptions for Fade Version

We consider a *retrospective attack* threat model: an attacker wants to recover specific files that have been deleted. This type of attack may occur if there is a security breach in the cloud data center, or if a subpoena is issued to demand data and encryption keys. We assume that the attacker is omnipotent, i.e., it can obtain copies of any encrypted data, as well as keys on any machines.

Our security goal is to achieve *assured deletion* of files for a cloud backup system with version control. We adopt the cryptographic approach [19, 20, 21, 22, 23, 24], i.e., by removing the keys that are used to decrypt the data backups stored on the cloud. We make two assumptions for this approach. First, the encryption operation is secure, in the sense that it is computationally infeasible to revert the encrypted data into the original data without the decryption key. Second, we assume that the decryption keys are maintained by a key escrow system that is totally independent of the cloud and can be fully controlled by cloud clients. If a file is requested to be assuredly deleted, then we require the associated key be securely erased [17], which we believe is feasible given that the size of a key is much smaller compared to a backup file. In chapter 3.7, we discuss in more detail the design of the key escrow system.

3.2 Motivation

We argue that existing version-controlled cloud backup systems (e.g., Cumulus [3]) and assured deletion systems (e.g., Vanish [21] and FADE [23]) are *incompatible.* To elaborate the issue, we consider a scenario in which we archive data backups using two independent systems, i.e., a version control system and an assured deletion system, and explain how they break certain functionalities.

There are two approaches of deployment. In the first approach, we first pass data backups through the version control system, followed by the assured deletion system, as shown in Figure 3.1(a). Suppose that Version V_1 is first generated, followed by Version V_2. In this case, if there are some identical file copies in both versions, then Version V_2 can keep references to point to the identical file copies in Version V_1 instead of storing redundant file copies. In other words, Version V_2 may *depend* on some files in Version V_1. Then we pass the versions through the assured deletion system, which we assume is based on cryptographic protection as described in chapter 2. Now, if we want to assuredly delete Version V_1, then we can remove the cryptographic key that encrypts Version V_1. However, since Version V_2 shares some files in Version V_1, some files in Version V_2 also become inaccessible. In short, assuredly deleting one version may also affect future versions.

In the second approach, we first pass data backups through the assured deletion system, followed by the version control system, as shown in Figure 3.1(b). First, each backup file is encrypted with different cryptographic keys by the assured deletion system. If two identical files are encrypted with different keys, then their encrypted copies will have different format. Thus, if we pass these encrypted files through the version control system, then the version control system cannot discover any commonality between the encrypted copies and cannot share identical files across versions.

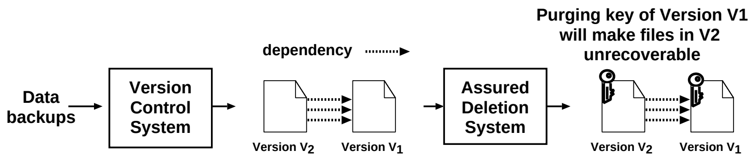
3.3 Main Idea

Our goal is to make both version control and assured deletion compatible with each other in a single design. The main idea of FadeVersion is as follows. We first start with the design of a version-controlled cloud backup system that has similar ideas as in Cumulus [3], in which we create different data objects that are to be archived on the cloud. On top of the version control design, we add a *layered* approach of cryptographic protection, in which data is encrypted with the first layer of keys called the *data keys*, and the data keys are further encrypted with another layer of keys called the *control keys*. The control keys are defined by *fine-grained policies* that specify how each file is accessed. If a policy is revoked, then its associated control key is deleted. If the data object is associated *solely* with the revoked policy, then it will be assured deleted; if the data object is associated with both the revoked policy and another active policy, then we still allow the data object to be accessed through the active policy. We elaborate how this idea is designed and implemented in the following chapters.

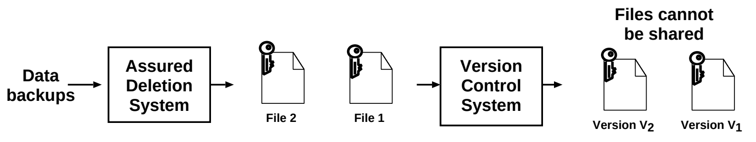
3.4 Version Control

In FadeVersion, each backup version (or *snapshot*) arranges data files into *file objects*. Each file object is of variable size with a configurable maximum-size threshold (e.g., currently set as 1 MB). If a file has size less than the threshold,

Figure 3.1 Illustration of why existing version control systems and assured deletion systems are incompatible.



(a) First version control, followed by assured deletion



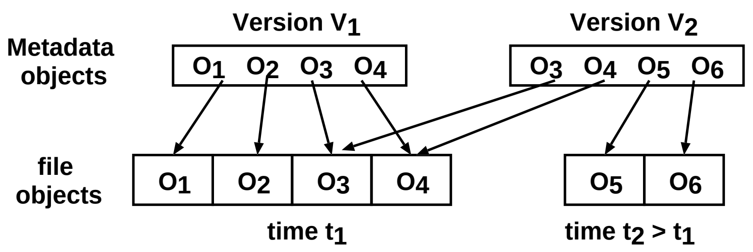
(b) First assured deletion, followed by version control

then it can be represented by a single object; otherwise, we split the file into multiple objects. Thus, if there is any modification to a large file, then we only need to upload the modified objects, rather than the whole file, to the cloud so as to save the upload and storage costs. To further reduce the upload cost, we can group multiple objects into a segment, and each transfer request is done on a per-segment basis [3].

In many cases, the same file (or object) may appear in multiple backup versions, or different files (or objects) may have the same content in the same or different versions. We employ *deduplication* [4] to further reduce storage. Specifically, if two objects have the same content, then we only need to store *one* object on the cloud and create smaller-size pointers to reference the stored object. To determine if two objects have the same content, we apply a cryptographic hash function (e.g., SHA-1) to the content of each object and check if both objects return the same hash value.

We may further look for the identical content that can be deduplicated within an object using a more fine-grained technique like Rabin Fingerprints [39]. However, we note that it does not always significantly improve the storage efficiency, such as using the datasets in our experiments (see Chapter 5). Therefore in this thesis, we assume that an object is the smallest unit of data

Figure 3.2 Illustration of how version control works.



backups.

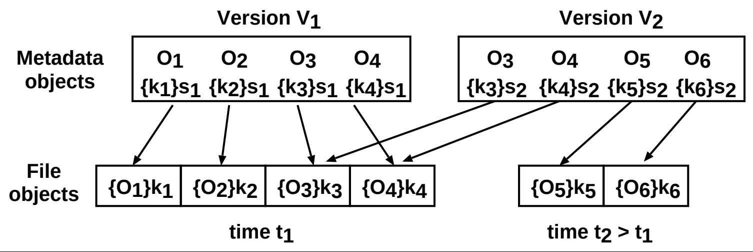
FadeVersion allows users to archive backup files at different time instants, and organizes backups into different versions (snapshots). For each version, there is a *metadata object* that describes the file objects. Figure 3.2 illustrates how we upload different backup versions. Suppose that at time t_1, we want to upload a version V_1 of four file objects: (O_1, O_2, O_3, O_4). Suppose later at time $t_2 > t_1$, we do not include O_1 and O_2, but add new file objects O_5 and O_6. Thus, the new version V_2 will upload the physical copies of O_5 and O_6, and its metadata object has pointers to refer to the physical copies of O_3 and O_4 in version V_1. Finally, all the metadata objects and file objects are stored on the cloud.

3.5 Assured Deletion

We now incorporate assured deletion into the version control design discussed in the previous sections. To simplify our discussion, we focus on the case where we want to assuredly delete a particular backup version.

FadeVersion employs *two-layer encryption* to achieve assured deletion. Figure 3.3 illustrates the idea. Denote $\{.\}k$ as the symmetric-key encryption (e.g., AES [40]) with key k. For each object O_i, we generate a *data key* k_i, and encrypt O_i with k_i via symmetric-key encryption (i.e., compute $\{O_i\}k_i$). For each version V_i, we generate a *control key* s_i, and encrypt all data keys of the objects associated with version V_i using s_i via symmetric-key encryption (i.e.,

Figure 3.3 Illustration of layered encryption, by extending the example shown in Figure 3.2.



compute $\{k_i\}s_i$). The encrypted data keys are stored in the metadata object of version V_i, and will be later uploaded to the cloud. The control keys are kept by a key escrow system (see Chapter 3.7). To recover a file object of a version, we need to get the corresponding control key of the version from the key escrow system, and decrypt the corresponding data key and hence the encrypted file object.

The deduplication feature is still maintained. For example, in Figure 3.3, both the encrypted copies O_3 and O_4 are still shared by both versions V_1 and V_2. Their respective data keys k_3 and k_4 are separately encrypted with s_1 and s_2. To recover O_3 and O_4, we can use either s_1 or s_2 to decrypt their data keys.

We now explain how FadeVersion enables assured deletion of a particular version. Suppose that we request to assuredly delete a particular version V_1. Then FadeVersion will *purge* the control key s_1 from the key escrow system. Since s_1 is purged, we cannot decrypt the encrypted data keys associated with snapshot V_1, even if there are many replicated copies on the cloud. Note that file objects O_1 and O_2 only appear in the assuredly deleted version V_1, but not in other active versions. Thus, both of them will become permanently inaccessible.

Note that the assured deletion of one version does not affect other active versions, even if different versions have *data dependency.* When we purge the control key s_1, we can still retrieve version V_2 that is protected by a different control key s_2, and hence recover the file objects O_3 and O_4. The layered encryption approach in essence *decouples* the data dependency across versions.

3.6 Assured Deletion for Multiple Policies

We can generalize the idea of assured deletion for multiple policies, each of which specifies the access privilege of a file object. Each file object can be simultaneously associated with multiple policies. If any one of the policies is revoked, then the file object will be assuredly deleted. This enables us to perform *fine-grained* assured deletion on data backups that are stored on the cloud.

To formalize, we now revise our notation associating a file object with multiple policies as follows. Let k_{id} be the data key for file object with a unique identifier *id*. Let P denote the policy that describes the access right for a file object, and s_P be the control key associated with policy P. Let $\{m\}k$ denotes the symmetric-key encryption of message m with key k. Thus, to protect a file object O with identifier *id* with policies P_1, P_2, $\cdots$, and P_n, we apply layered encryption as follows:

$$\{O\}k_{id} \text{ and } \{\{\{k_{id}\}s_{P_1}\}s_{P_2}...\}s_{P_n}.$$

If any control key s_{P_i} ($1 \leq i \leq n$) is purged, then k_{id} becomes inaccessible, so does file object O.

We illustrate how fine-grained assured deletion is achieved. Suppose that we archive the data files of Alice on a regular basis. Then we can associate each file object for file F with three policies: (i) *user-based policy* (e.g., "accessible by Alice only"), (ii) *file-based policy* (e.g., "accessible via file F only"), and (iii) *version-based policy* (e.g., "accessible via backup version V_i only"). Then we can support three different operations of assured deletion, respectively: (i) assuredly deleting all files of Alice across all backup versions by revoking the user-based policy, (ii) assuredly deleting file F across all backup versions by revoking the file-based policy, (iii) assuredly deleting a particular backup version by revoking the version-based policy. We point out that we can readily generalize the assured deletion scheme for other combinations of policies.

3.7 Key Management

The control keys are maintained by a key escrow system, which we assume can securely remove the control keys associated with revoked policies to achieve assured deletion (see Chapter 3.1). On the other hand, it is still important to maintain the robustness of the existing control keys that are associated with active policies. Here, we discuss two possible approaches to address the robustness of key management.

One approach is by encrypting all control keys with a single *master key*, while this master key is stored in secure hardware (e.g., trusted platform module [41]). The justification is that protecting the robustness of a single key is easier than protecting the robustness of multiple keys. However, if the hardware that stores the master key is failed, then all control keys will be lost.

Another approach is by using a quorum scheme based on threshold secret sharing [42]. Each control key is split into N key shares and are distributed to N independent key servers, such that we need at least $K < N$ of the key shares to recover the original control key. The justifications of applying the quorum scheme are two-fold. First, even if one key server is failed, we can still obtain the key shares from the remaining $N - 1$ key servers. This ensures the fault-tolerance of key management. Second, an attacker needs to compromise at least K key managers in order to obtain the control key for decrypting the data on the cloud. This increases the attack resources required by the attacker. On the other hand, the challenge is that it increases the management overhead of maintaining multiple key servers.

Chapter 4

Implementation of FadeVersion

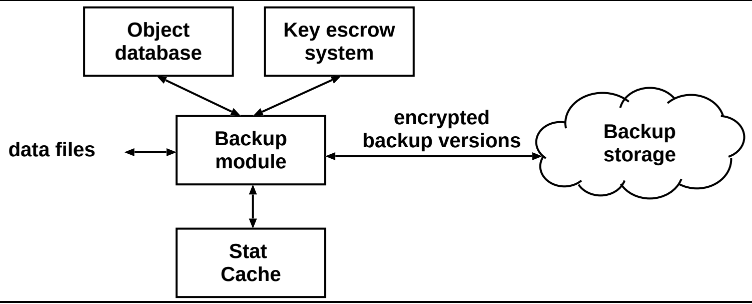
We now present how FadeVersion is implemented to support both version control and assured deletion. FadeVersion is an extension of Cumulus [3], upon which we add new cryptographic implementation for assured deletion. The cryptographic operations are implemented with OpenSSL [43].

4.1 System Entities

FadeVersion is built on several system entities, as illustrated in Figure 4.1. Their functionalities are described as follows.

Backup storage. It is the target destination where data backups are stored. The current implementation of FadeVersion uses Amazon S3 [44] as the storage backend. This can be easily extended to other third-party cloud storage

Figure 4.1 Architecture of FadeVersion.



services that offer generic file access semantics such as `put`, `get`, `list`, and `delete`.

Backup module. This is to (i) create backup versions from data files and upload them to the cloud, and (ii) retrieve backup versions from the cloud and recover the original data files. It acts as an interface for other entities. It queries the object database for deduplication optimization, and communicates with the key escrow system to obtain the keys for encryption/decryption.

Object database. It maintains the identifiers and hash values of all file objects that are stored in the backup storage. It also stores the data key for each file object. During the backup operation, the backup module queries the object database to check by hash values whether an identical file object is created in the previous backup version, so as to perform deduplication if possible. If an identical file object is found, then the corresponding data key will be retrieved, encrypted with the corresponding control keys, and included in the new backup version. The backup module also records new file objects in the database. We currently deploy the object database locally with the backup module. We also use SHA-1 as the hashing algorithm, but this can be easily configurable.

Key escrow system. It creates and manages control keys associated with policies (see Chapter 3.6). It creates mappings between each policy (defined by a unique identifier) and the corresponding control key. Currently, the key escrow system is implemented as a single key server process, which is deployed locally with the backup module. However, it can be extended for a higher degree of fault tolerance (see Chapter 3.7).

Stat Cache [3]. It keeps metadata locally generated by the system from completed backups (see Chapter 4.2) to improve backup performance. During the backup process, it can use the stored metadata to check if a file has been modified by comparing the last modification time of the file[1]on the filesystem and in the metadata. If the file has not been modified, then the backup module will directly reuse the information from the stat cache to construct the metadata of the unmodified file for the current backup version, without needing to process the unmodified file again.

4.2 Metadata Format in FadeVersion

We use the metadata object to keep the information of all archived file objects in a backup version (see Chapter 3.4). FadeVersion extends the metadata format in Cumulus [3] to include the policy information and the encrypted cryptographic keys, both of which are used for assured deletion.

Figure 4.2 shows the metadata formats for a single file in Cumulus and FadeVersion, assuming that the file contains three file objects (i.e., `A/1`, `A/2`, `A/3`). In FadeVersion, we add an additional field named `key`, which stores the data key of each associated data object. The data key is encrypted with the control keys of the corresponding policies, and the control keys are kept by the key escrow system. In our prototype, each file object is associated with three policies (see Chapter 3.6): (i) user-based policy, which is described by the `user` field, (ii) file-based policy, which is described by the `name` field, and (iii) version-based policy, which is described by the version in which the file resides. Based on the information, FadeVersion can know how to restore a file, i.e., by using the correct control keys from the key escrow system to decrypt the data keys, and how to revoke a policy and its associated files.

We use AES [40] as the encryption algorithm to encrypt file objects and their corresponding data keys. AES is a block-cipher encryption scheme with block size 128 bits, so the size of the encrypted data key remains the same even it is encrypted multiple times with different policies. In our implementation, we use the 128-bit key size for both data keys and control keys, so the size

[1] The modification time of a file is obtained using the `stat()` system call.

Figure 4.2 Metadata format for a single file: Cumulus (left) and FadeVersion (right).

```
name: fileA
checksum: sha1=...
ctime: 1300000000
data: A/1
    A/2
    A/3
group: 1 (root)
inode: ...
mode: 0755
mtime: 1300000000
size: 3000000
type: f
user: 1 (root)
```

```
name: fileA
checksum: sha1=...
ctime: 1300000000
data: A/1
    A/2
    A/3
group: 1 (root)
inode: ...
key: ENCRPYTED KEY FOR A/1
    ENCRYPTED KEY FOR A/2
    ENCRYPTED KEY FOR A/3
mode: 0755
mtime: 1300000000
size: 3000000
type: f
user: 1 (root)
```

of the encrypted data key is fixed to be 128 bits (16 bytes). If a file object has a large size, then the storage overhead for its encrypted data key will be insignificant.

Chapter 5

Evaluation of FadeVersion

In this chapter, we conduct an empirical study on the prototype of FadeVersion. We compare FadeVersion with Cumulus [3]. Our goal is to evaluate the performance overhead of adding assured deletion on top of a version-controlled cloud backup system. We explore the overhead from three perspectives: (i) backup/restore time, (ii) storage space, and (iii) monetary cost, and we show that the overhead of adding assured deletion is reasonable or minimal in all three perspectives.

5.1 Setup

Our experiments use Amazon S3 Singapore as our cloud storage backend. We deploy both Cumulus and FadeVersion on a Linux machine that resides in Hong Kong. The Linux machine is configured with Intel Quad-Core 2.4GHz CPU, 8GB RAM, and Seagate ST3250310NS hard drive.

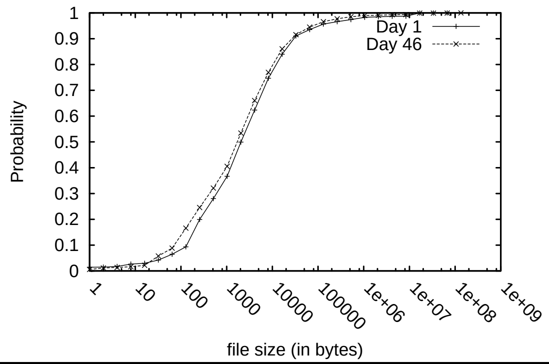
We drive our experiments with real-life workload. We conduct nightly backups for the file server of our research group. The dataset that we use consists of 46 days of snapshots of the home directory of one of the users. Table 5.1 summarizes the statistics of the dataset, including the summaries of the full snapshots on the first day (i.e., Day 1) and last day (i.e., Day 46).

Figure 5.1 shows the cumulative distribution functions of file sizes of the

Table 5.1 Statistics of our dataset.

	Day 1	Day 46
Number of files	5590	11946
Median	2054 B	1731 B
Average	172 KB	158 KB
Maximum	56.7 MB	100 MB
Total	940 MB	1.85 GB

Figure 5.1 File size statistics for Day 1 and Day 46.



full snapshots on Day 1 and Day 46, respectively, and Figure 5.2 shows the size of data changes per day reported by `rdiff-backup` [45]. Since the size of data changes is less significant compared to the size of the entire home directory, we expect that the distributions of file sizes across different days of data backups remain fairly stable throughout the entire backup period.

Figure 5.2 Size of data changes reported by `rdiff-backup` per day.

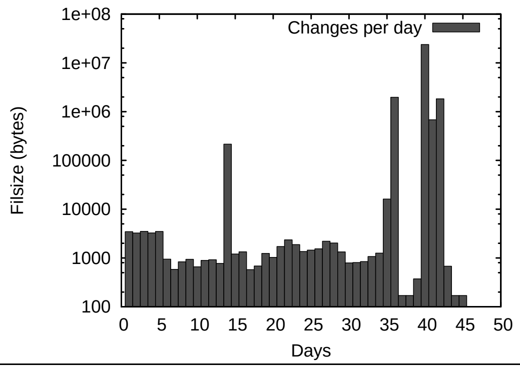


Figure 5.3 Backup time for each incremental backup.

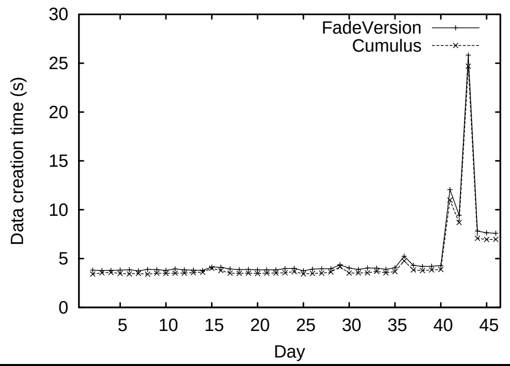
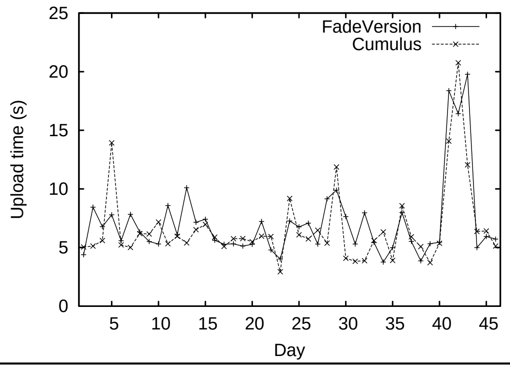


Figure 5.4 Upload time for each incremental backup.

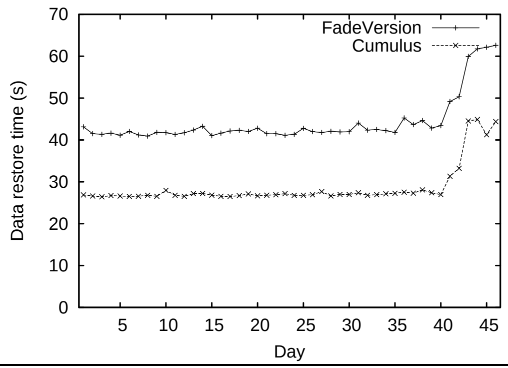


5.2 Backup/Restore Time

We first evaluate the backup operation. On Day 1, both Cumulus and FadeVersion start the *initial backup*, which uploads the full snapshot of the home directory to the cloud; from Day 2 onwards, both systems will conduct the *incremental backups*, which store the backup versions that are incrementally built from the previous backup versions.

The backup times for performing a full snapshot on the first day for Cumulus and FadeVersion are 43.18s and 44.55s, respectively (i.e., FadeVersion uses 3.2% more time). The additional overhead of FadeVersion is mainly due to the key management and cryptographic operations, but such overhead is minimal compared to Cumulus.

Figure 5.5 Restore time for all 46 days of snapshots from local storage.



The backup time of storing each incremental backup on the cloud is composed of two parts: (i) the time for creating a backup version based on the previous backup versions, and (ii) the time for uploading the created backup version to the cloud (i.e., Amazon S3 Singapore). Our measurements are averaged over three times.

Figure 5.3 shows the time for creating incremental backups for Cumulus and FadeVersion. FadeVersion introduces higher creation time. On average, FadeVersion uses 9.8% more time than Cumulus in creating incremental backups.

Figure 5.4 shows the time for uploading incremental backup versions to the cloud. We only measure the time to upload the incremental backups but not for the initial backup, as the latter takes much longer time than the incremental backups that follow. We observe that both Cumulus and FadeVersion have very similar values of upload time, and the average values are 6.624 s and 7.106 s, respectively.

We also evaluate the time for restoring a backup. The restore operation includes: (i) downloading the necessary file objects from the cloud and (ii) restoring the original view of the entire home directory. We note that the former part takes the dominant portion of time, and the overhead added by our restore module becomes insignificant. For instance, we try restoring the snapshot for Day 46 from S3, and the time taken (averaged over 10 trials each) by Cumulus and FadeVersion are 26.47 minutes and 26.13 minutes respectively, in which 25.11 minutes and 24.47 minutes are used in downloading files. Thus, both systems have very similar restore time, and the overhead of FadeVersion is easily masked by the downloading time. In order to minimize the effect of network fluctuations in restore time, we try restoring from local storage. Figure 5.5 shows the results of restoring snapshots from all 46 days in sequence from the local storage. On average, FadeVersion uses 55.1% more time than Cumulus in restoring backups. The overhead of FadeVersion is mainly due to the cryptographic operations of decrypting *all* encrypted file objects, and this accounts for 97.25% of the overhead on average. Note that the increase in restore time around day 40-45 for both systems is due to the increase of the size and number of files of the dataset on those days, both systems will have to process and copy more data from the file objects in order to recover the filesystem when compared to the previous snapshots.

5.3 Storage Space

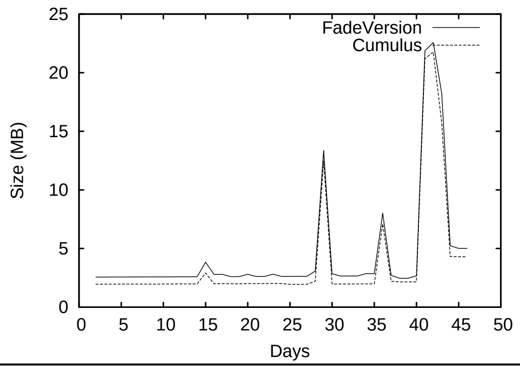
FadeVersion includes encrypted copies of data keys in data backups for assured deletion (see Chapter 4.2), and this introduces storage space overhead. Here, we evaluate the space overhead of FadeVersion due to the storage of keys. Table 5.2 summarizes the storage space of both systems. Note that the actual storage space on the cloud is less than the full snapshot sizes as shown in Table 5.1, mainly because both Cumulus and FadeVersion exploit deduplication to reduce the storage of redundant data (see Chapter 3.4). On average, FadeVersion introduces 19.4% more space increment per month compared to Cumulus.

We now focus on incremental backups. Figure 5.6 illustrates the storage space of both Cumulus and FadeVersion in the incremental backups on different days. We observe that FadeVersion introduces fairly similar storage space

Table 5.2 Summary of storage space on the cloud using Cumulus and FadeVersion.

	Initial Storage on Day 1	Total Storage on Day 46	Increment per month
Cumulus	597.03 MB	755.73 MB	105.67 MB
FadeVersion	597.51 MB	786.88 MB	126.24 MB

Figure 5.6 Size of incremental uploads for Cumulus and FadeVersion.



overhead on each day.

5.4 Monetary Cost

We estimate the monetary cost overhead of FadeVersion after adding assured deletion. Here, we focus on the backup operation. We consider the monetary costs due to two components: (i) the storage cost of storing 46 days of backup for a month and (ii) the bandwidth cost of uploading 46 days of incremental backups to the cloud since the initial backup. We consider the pricing plans of various cloud providers in addition to Amazon S3.

Table 5.3 shows the costs of Cumulus and FadeVersion. We observe that when compared to Cumulus, the additional storage cost of FadeVersion is within \$0.008 per month, and its additional bandwidth cost is within \$0.003. The monetary cost overhead of FadeVersion is minimal in general.

Table 5.3 Storage costs per month and overall bandwidth cost of Cumulus and FadeVersion for 46 days of backup with different cloud providers.

Providers	Storage $/GB/month	Cumulus	Fade-Version	Bandwidth for updates $/GB	Cumulus	Fade-Version
S3 (Singapore)	0.14	$0.103	$0.108	0.10	$0.0154	$0.0185
Rackspace	0.15	$0.111	$0.115	0.08	$0.0124	$0.0148
Nirvanix SDN	0.25	$0.184	$0.192	0.10	$0.0154	$0.0185
Windows Azure	0.15	$0.111	$0.115	0.10	$0.0154	$0.0185
Google Storage	0.17	$0.125	$0.131	0.10	$0.0154	$0.0185

5.5 Conclusions

We present the design and implementation of FadeVersion, a system that provides secure and cost effective backup services on the cloud. FadeVersion is designed for providing assured deletion for remote cloud backup applications, while allowing version control of data backups. We use a layered encryption approach to integrate both version control and assured deletion into one design. Through system prototyping and extensive experiments, we justify the performance overhead of FadeVersion in terms of time performance, storage space, and monetary cost.

We note that the main performance overhead of FadeVersion is the additional storage of cryptographic keys in data backups. In future work, we explore possible approaches of minimizing the number of keys to be stored and managed.

Chapter 6

CFTDedup Design

This chapter presents the design and implementation details of CFTDedup. The main use of CFTDedup is to provide deduplication for archival applications in cloud storage via a fault-tolerant distributed proxy design.

6.1 Failure Model

CFTDedup is deployed as a distributed proxy system in which each client connects to one of multiple proxies. In this work, we assume a *crash* (or *fail-stop*) failure model, in which both a proxy and a client may stop operating and lose all states maintained by them.

The goal of CFTDedup is to ensure *crash fault tolerance (CFT)* against proxy/client crashes. In case a proxy crashes, it loses all deduplication metadata that identifies whether an uploaded block can be deduplicated. We require that CFTDedup enable a client to connect to a different proxy that keeps the same deduplication metadata when the failed proxy crashes. CFTDedup achieves this by enforcing a *strong consistency* model, in which the deduplication metadata is fully synchronized across all proxies. On the other hand, in case a client crashes while uploading data, its current upload session aborts. We require that CFTDedup roll back the upload session of the failed client and restore the deduplication metadata to the state right before the upload session starts.

Here, we do not consider Byzantine faults, in which faulty nodes may behave arbitrarily [46]. We also assume that the cloud storage site provides fault-tolerant storage and is always available.

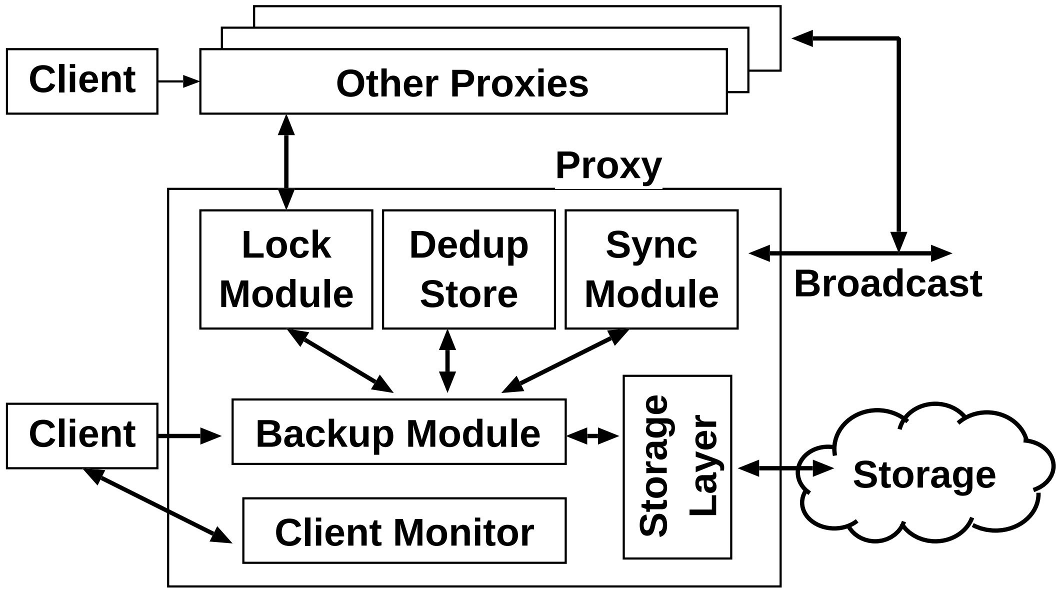
6.2 System Overview

Figure 6.1 illustrates the architecture of CFTDedup. Each Client uploads/-downloads data to/from the cloud via a proxy. Each proxy comprises several building blocks. The Backup Module executes the deduplication algorithm, and coordinates other modules to perform deduplication. The Dedup Store keeps the deduplication metadata, which will be replicated across all proxies. The Sync Module broadcasts changes of deduplication metadata to all proxies. The Lock Module implements distributed locking that ensures that only one proxy can modify deduplication metadata at any time. The Client Monitor monitors whether a client is alive, and rolls back the client's upload process if the client fails. The Storage Layer is an abstraction layer between a proxy and the cloud storage site.

CFTDedup is designed with modularity in mind. This enables us to easily add new functionalities into each module and address different open issues (see Chapter 8.1).

CFTDedup contains multiple proxies that we assume are deployed as independent servers. Since the proxies need to maintain a synchronized view, we assume that they are interconnected via a high-speed local area network so as to exchange view updates with low latency.

Figure 6.1 CFTDedup architecture.



6.3 Distributed Deduplication

We now describe how a CFTDedup proxy operates on a client's data stream (called a *snapshot*) being uploaded to the cloud and synchronizes the deduplication operation among several proxies.

Deduplication design. The Backup Module divides a client's snapshot into *blocks*, and applies cryptographic hashing to the block content. Multiple blocks with identical content all have the same hash, and only the blocks with unique content will be uploaded to the cloud. We assume that the probability that two distinct blocks have the same hash is negligible [4]. Our current implementation assumes *fixed-size* blocks of size 4KB each, but we can also apply deduplication on variable-size blocks (e.g., using Rabin fingerprinting [39]). Also, our current cryptographic hashing scheme is based on SHA-256.

Our deduplication approach is *inline*, meaning that deduplication is applied on the write path (i.e., when the data is about to be uploaded to the cloud). Inline deduplication not only improves storage efficiency, but also eliminates the transmission overhead of uploading redundant data to the cloud.

Each proxy keeps the *deduplication metadata*, which is used to identify if an uploaded block can be deduplicated with any identical blocks of the currently and previously uploaded snapshots. It holds the meta records of *all* unique blocks that have been uploaded. Each block record has the hash value, the block address, and a reference count that specifies the number of uploaded blocks sharing the same hash. The deduplication metadata is stored in Dedup Store.

Locking and synchronization. Our current design enforces strong consistency, such that only one proxy can handle data upload and modify the deduplication metadata at any time. Our goal is to have all proxies share the same view of the deduplication metadata. To achieve this, the Backup Module must first acquire a *distributed lock* from the Lock Module whenever accessing and modifying the deduplication metadata. The distributed lock can be held by one process at any time. Also, we use an *atomic broadcast model* [47] to synchronize a deduplication metadata update among all proxies. By atomic, we mean that when the Sync Module broadcasts an update, all surviving proxies must reliably receive the update.

Batching. If we update the deduplication metadata for each uploaded block, the overheads due to locking and synchronized broadcast will significantly increase. Thus, we apply the *batching* concept as follows. After the Backup Module locks the deduplication metadata, it keeps collecting uploaded blocks from the Client. It checks if each block can be deduplicated, and meanwhile updates its own copy of the deduplication metadata. It aggregates the unique blocks into a *segment*. If the segment size exceeds a pre-defined threshold or when the snapshot reaches the end, then it uploads the segment as a data object to the cloud and broadcasts the batched update of the deduplication metadata to other proxies. Finally it releases the lock. With batching, we perform locking and synchronization operations on a *per-segment basis* rather than on a per-block basis, thereby reducing the overheads incurred.

Putting it all together: upload/download. A client performs the upload/download operations as follows. In upload, a client provides a snapshot of data to a proxy, which applies deduplication and broadcasts the deduplication updates as described above. The proxy also constructs a *snapshot metadata*, which contains the block addresses describing how the snapshot can later be downloaded and reconstructed. The snapshot metadata generally has a much smaller size than the actual snapshot data, and will also be stored on the cloud.

In download, the client first downloads, via one of the proxies, the snapshot metadata, followed by the segments that contain the blocks of the snapshot. The proxy can cache the downloaded blocks locally, so any subsequent blocks with identical content need not be downloaded again. Here, the unit of download is a segment. Since a segment may contain blocks referenced by other snapshots, a large segment size can increase the likelihood of downloading unnecessary data blocks.

6.4 Crash Fault Tolerance

We discuss how CFTDedup recovers from crash failures.

Proxy failure. Suppose that the proxy that currently handles the upload operation fails before broadcasting the deduplication metadata updates. Then the client re-connects to another surviving proxy, and resumes uploading the segment currently handled by the failed proxy. Note that the failures of other proxies do not affect the current upload session.

Client failure. If the client fails during upload, then all proxies decrement all the reference counts of the blocks that have been uploaded, implying that the upload session of the client is rolled back. The blocks that have already been uploaded to the cloud may be reclaimed later via garbage collection.

6.5 Implementation

We implement a preliminary prototype of CFTDedup in Java. CFTDedup requires two specific services: distributed locking and atomic broadcast, which we currently implement by Zookeeper [38] and Spread [48], respectively. We also leverage Zookeeper, a distributed coordination service, to monitor all clients and proxies. Any client/proxy that has no response after a pre-configured timeout is considered to be failed.

Note that our CFTDedup prototype is still in its early development stage and hence only provides baseline performance. Its performance could be improved via careful optimizations.

Chapter 7

Evaluation of CFTDedup

We conduct testbed experiments to evaluate different operations of our CFTDedup prototype, and to understand the performance overhead of CFTDedup in maintaining strong consistency among multiple proxies.

We show that by increasing the segment size, we are able to reduce the overhead of synchronization to a reasonable percentage with a small penalty on restore time, and the time required by clients to switch proxies to resume backup when proxies fail is small when compared with restarting backup from the beginning.

7.1 Setup

Dataset. We consider a dataset of 21 virtual machine (VM) images that are used by students for their programming projects in a university undergraduate course. Each VM is initially installed with Ubuntu 10.04 and allocated with 10GB disk space. It is also configured to download and install any latest patches from the Internet when it is online. Students develop their programs and install applications on their assigned VMs during a 3-month semester. At the semester end, we collect a snapshot of the 21 VM images for our experiments.

Testbed. Our testbed comprises two client machines and three proxy machines. We also configure a FTP server that mimics a cloud storage site. Each machine is equipped with 2.66GHz CPU and 4GB RAM. All machines are interconnected with a Gigabit Ethernet switch.

Metrics. Our experiments mainly focus on two metrics: storage space usage and runtime performance of different operations. In runtime measurements, we obtain average results over three runs.

7.2 Experiment 1 (Archival)

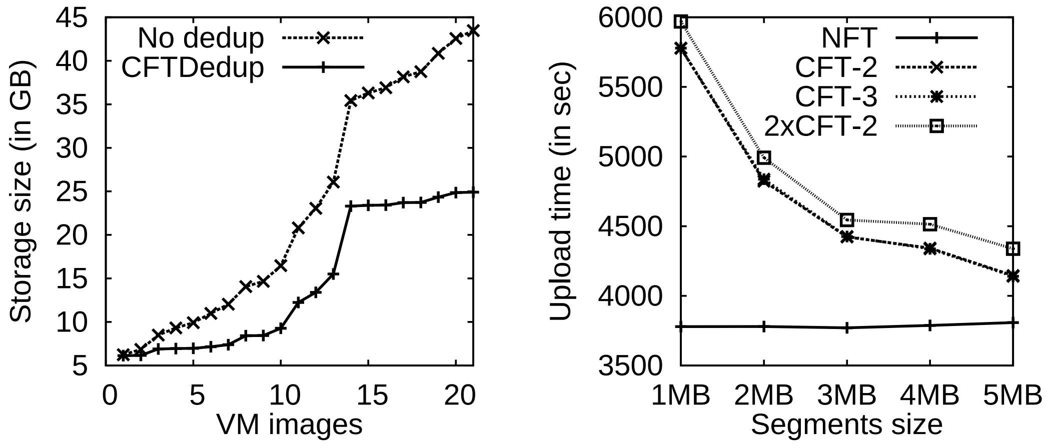
We evaluate the archival performance of uploading all VM images in our dataset, in terms of storage space used and upload runtime. Here, we upload all VM images as a single snapshot.

Figure 7.1(a) shows the cumulative space usage for storing all VM images using deduplication with 4KB block size. Compared to without deduplication (in which we exclude zero-filled blocks), we reduce the storage space by around 40%. Note that the snapshot metadata in our CFTDedup implementation only introduces 0.323% of storage space overhead to the snapshot of VM images (not shown in the figure).

We also measure the runtime performance of uploading all VM images under different settings: a single proxy without CFT (denoted by NFT), two proxies with CFT (denoted by CFT-2), and three proxies with CFT (denoted by CFT-3). All the above settings use only one client. Also, we run two clients connecting to two different proxies with CFT (denoted by 2xCFT-2). We split the dataset into two subsets of roughly the same size and have both clients upload the subsets concurrently. We then measure the time it takes for both clients to complete the upload.

Figure 7.1(b) shows the upload time versus the segment size for each of the settings. Compared to NFT, the CFT settings have higher upload times due to the locking and synchronization overheads. However, such overheads can

Figure 7.1 Experiment 1 (Archival performance): (a) Cumulative space usage with and without deduplication; (b) Upload time versus segment size.



(a) (b)

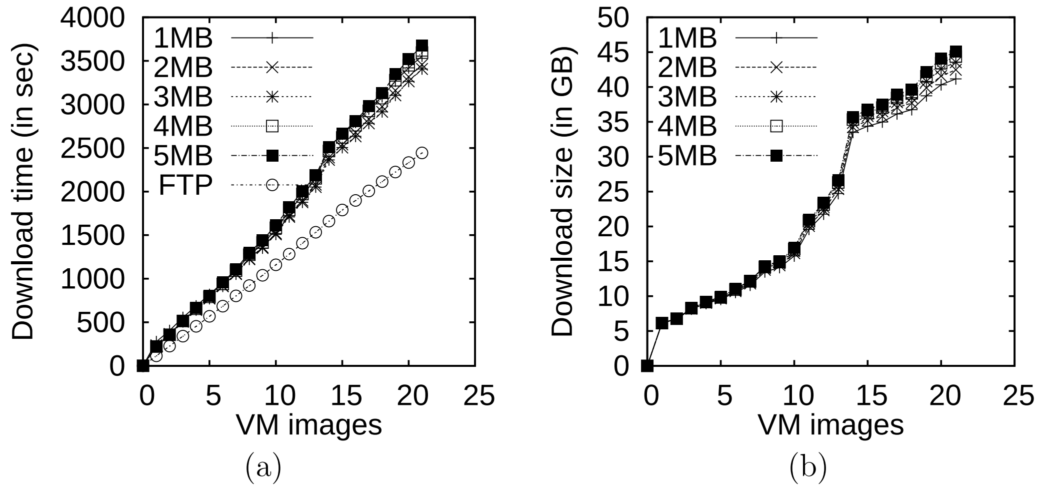
be reduced as the segment size increases. For example, comparing CFT-2 and CFT-3 (both of which have very similar performance) to NFT, the increase in upload time drops from 52.9% to 8.7% when the segment size increases from 1MB to 5MB. In addition, 2xCFT-2 has slightly higher upload time than CFT-2 by 3.3-4.6%, since the proxies compete for the distributed lock during the upload.

7.3 Experiment 2 (Restore)

We now evaluate the performance of restoring VM images on the client. We download each VM image individually that was uploaded in Experiment 1, in which we vary the segment size. The disk and memory cache on the proxy are cleared after each VM image is downloaded.

Figure 7.2(a) shows the times needed to download each VM image for different segment sizes. It also shows the time of downloading each VM image directly from the FTP server when no deduplication is applied to the VM image storage. Compared to without deduplication, CFTDedup incurs an overhead of 29-50%. The reason is that blocks of a VM image can be deduplicated with

Figure 7.2 Experiment 2 (Restore performance): (a) Download time versus segment size; (b) Amount of downloaded data versus segment size.



those of other VM images, and there are additional seeks for reconstructing a VM image from different segments. This problem is called fragmentation [31] and is inherent in deduplication. Also, the download time increases with the segment size. For example, the 5MB case incurs 3.4% more download time than the 1MB case.

To see why the download time increases with the segment size, Figure 7.2(b) shows the amount of downloaded data during restore. As the segment size increases, more data is downloaded. The reason is that data is downloaded on a per-segment basis (see Chapter 6.3). A segment that we download may contain blocks not being used by the current VM image file. A larger segment generally contains more such unused blocks. For example, the 5MB case has 9.55% more data than the 1MB case.

7.4 Experiment 3 (Recovery)

We now measure the recovery performance of CFTDedup when a proxy or a client fails. Here, we configure two proxies with CFT, and have a client upload all VM images as a single snapshot via one of the proxies using 1MB segment size. In the middle of the upload, we disable either the proxy that handles the upload or the client to resemble a failure. We leverage Zookeeper to detect failures using timeouts (see Chapter 6.5). In our experiment, we set the timeout to be 10s.

We first consider the recovery of a proxy failure. We disable the proxy that handles the upload at a certain time after the upload begins. The client will switch to another proxy and resume the upload session. We then measure the recovery time from when the failure happens until the upload session resumes. Figure 7.3(a) shows the recovery time required versus the proxy's lifetime. The recovery time generally takes 10-12s, and this is mainly determined by the 10s timeout in our Zookeeper configuration.

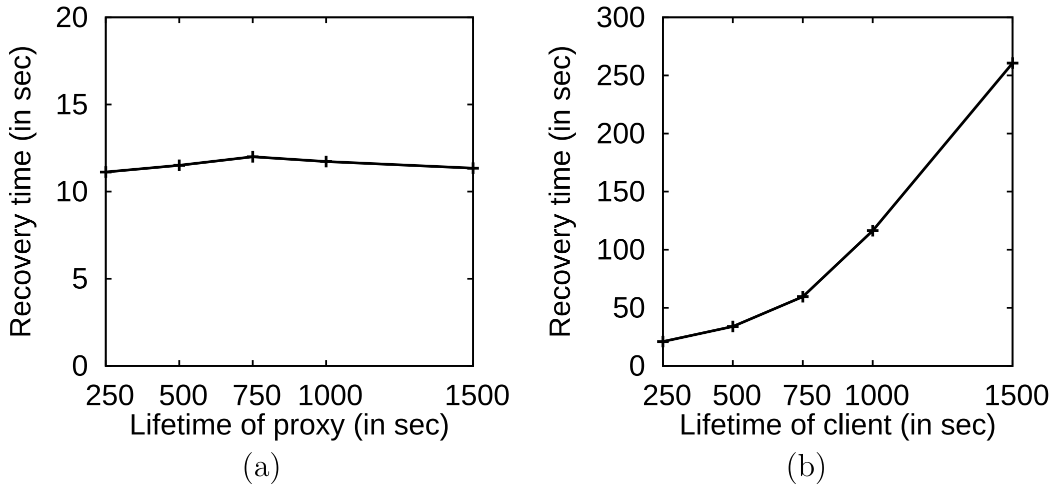
We next consider the recovery of a client failure, in which the proxy needs to roll back the client's upload session. We disable the client at a certain time after the upload begins. We then measure the recovery time from when the failure happens until the entire client's upload session is rolled back. Figure 7.3(b) shows the recovery time versus the client's lifetime. The recovery time increases with the client's lifetime as more data has already been uploaded and needs to be rolled back. Nevertheless, the recovery time is fairly small. For example, if the client fails 1500s (or 25 minutes) after the upload session starts, the recovery time is within 4.5 minutes.

7.5 Summary

We summarize our findings in the previous sections below to better illustrate the trade-offs of introducing fault tolerance:

- When a proxy fails during backup, the time for a client to switch proxies is near constant (around 10-12s).
- Overhead in upload time can be as small as 8.7% by increasing segment

Figure 7.3 Experiment 3 (Recovery performance): (a) Recovery time when a proxy fails; (b) Recovery time when the client fails.



size to 5MB, while incurring a 3.4% time overhead during restore operations.

Therefore, we can conclude that for less than 10% of backup time overhead, users are able to avoid restarting their backup tasks due to proxy failures, while only incurring a small penalty for restore operations.

Chapter 8

Future work and Conclusions of CFTDedup

8.1 Future Work

Our CFTDedup prototype provides a platform for future research. We now highlight several open issues, and we plan to extend CFTDedup to address them.

Consistency models. We currently consider one extreme of consistency called strong consistency on deduplication metadata management. To further mitigate synchronization overhead, we can use a weaker form of consistency and allow proxies to have inconsistent deduplication metadata. It is possible that two identical blocks that are simultaneously uploaded to different proxies cannot be deduplicated with each other, since the updates of the deduplication metadata are not yet reflected among the proxies. Understanding the trade-offs between synchronization overhead and storage efficiency for different forms of consistency models is our future work.

Upload throughput. The current CFTDedup design fully serializes the data upload and deduplication operations, such that only one proxy can perform such operations at any time. However, with a weaker consistency model, we may parallelize the operations with multiple proxies and achieve higher upload throughput. In addition, load balancing among multiple proxies is possible to further improve the upload performance.

Deduplication design. We currently maintain a synchronized copy of the deduplication metadata that centrally coordinates the deduplication process among all proxies. To improve the deduplication performance, we may explore a decentralized approach similar to DeDe [35], in which each proxy locally applies deduplication and synchronizes the global deduplication updates later. Also, for some data workloads such as archival data, we can exploit data locality and design specific deduplication approaches [30]. Integrating more elegant deduplication designs into CFTDedup is an ongoing work.

Other issues. Other future directions include: (i) extending CFTDedup to support Byzantine fault tolerance, (ii) evaluating CFTDedup in larger-scale deployment, (iii) integrating CFTDedup with multiple cloud vendors as in [9, 10, 11], etc.

8.2 Conclusions

This thesis proposes the design and implementation of CFTDedup, a distributed proxy system which improves storage efficiency via deduplication in cloud storage, while ensuring crash fault tolerance among proxies. We implement a preliminary prototype of CFTDedup, and show via benchmarks that the tradeoffs of adding fault tolerance can be small. We plan to use it as a baseline to explore different consistency models, throughput enhancement techniques, and deduplication designs. Our preliminary CFTDedup prototype is currently available at: **http://ansrlab.cse.cuhk.edu.hk/software/cftdedup**.

Chapter 9

Conclusion

This thesis has presented methods to ensure the data security and reliability of cloud backup systems.

In the first part of this thesis (chapters 3 - 5), we presented FadeVersion, a secure cloud backup system that serves as a security layer on top of todays cloud storage services. It achieves fine-grained assured deletion on top of deduplication by employing two-layer encryption. Cloud clients can assuredly delete particular backup versions or files on the cloud while making them permanently inaccessible to anyone, while other versions that share common data of the deleted versions or files will remain unaffected. We implemented a proof-of-concept prototype of FadeVersion and conduct empirical evaluation atop Amazon S3, and showed that FadeVersion only adds minimal performance overhead over a traditional cloud backup service that does not support assured deletion.

In the second part of this thesis (chapters 6 - 8), we presented *CFTDedup*, a distributed proxy system designed for providing storage efficiency via deduplication in cloud storage, while ensuring crash fault tolerance among proxies. It synchronizes deduplication metadata among proxies to provide strong consistency, and batches metadata updates to mitigate synchronization overhead. We implemented a preliminary prototype of CFTDedup and evaluated via testbed experiments its runtime performance in deduplication storage for virtual machine images.

Both systems provide a platform for further research, especially CFTDedup with open issues described in chapter 8.1. With the increasing popularity of cloud storage, we anticipate that the techniques described in this thesis will be essential to ensuring security and reliability when using cloud storage systems for backup.

Bibliography

[1] D. McCullagh, Fbi, politicos renew push for isp data retention laws, http://news.cnet.com/8301-13578_3-9926803-38.html, 2008.

[2] Watson Hall Ltd, Uk data retention requirements, https://www.watsonhall.com/resources/downloads/paper-uk-data-retention-requirements.pdf, 2009.

[3] M. Vrable, S. Savage, and G. Voelker, Cumulus: Filesystem backup to the cloud, in *Proc. of USENIX FAST*, 2009.

[4] S. Quinlan and S. Dorward, Venti: a new approach to archival storage, in *Proc. USENIX FAST*, 2002.

[5] A. S. Gateway, http://aws.amazon.com/storagegateway/.

[6] Nasuni, http://www.nasuni.com.

[7] Panzura, http://www.panzura.com.

[8] StorSimple, http://www.storsimple.com.

[9] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, RACS: A Case for Cloud Storage Diversity, in *Proc. of ACM SoCC*, 2010.

[10] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, DEPSKY: Dependable and Secure Storage in a Cloud-of-Clouds, in *Proc. of ACM EuroSys*, 2011.

[11] Y. Hu, H. Chen, P. Lee, and Y. Tang, NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds, in *Proc. of USENIX FAST*, 2012.

[12] M. Vrable, S. Savage, and G. M. Voelker, BlueSky: A Cloud-Backed File System for the Enterprise, in *Proc. of USENIX FAST*, 2012.

[13] S. Quinlan and S. Dorward, Venti: a new approach to archival storage, in *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, volume 4, 2002.

[14] B. Zhu, K. Li, and H. Patterson, Avoiding the disk bottleneck in the data domain deduplication file system, in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, page 18, USENIX Association, 2008.

[15] Dropbox, `http://www.dropbox.com`, 2010.

[16] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds, in *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, ACM, 2009.

[17] P. Gutmann, Secure deletion of data from magnetic and solid-state memory, in *Proc. of USENIX Security Symposium*, 1996.

[18] Z. N. J. Peterson, R. Burns, J. Herring, A. Stubblefield, and A. D. Rubin, Secure Deletion for a Versioning File System, in *Proc. of USENIX FAST*, 2005.

[19] D. Boneh and R. Lipton, A Revocable Backup System, in *Proc. of USENIX Security Symposium*, 1996.

[20] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy, Keypad: An Auditing File System for Theft-Prone Devices, in *Proc. of ACM EuroSys*, 2011.

[21] R. Geambasu, T. Kohno, A. Levy, and H. Levy, Vanish: Increasing data privacy with self-destructing data, in *Proc. of USENIX Security Symposium*, 2009.

[22] R. Perlman, File System Design with Assured Delete, in *ISOC NDSS*, 2007.

[23] Y. Tang, P. Lee, J. Lui, and R. Perlman, FADE: Secure Overlay Cloud Storage with File Assured Deletion, in *Proc. of SecureComm*, 2010.

[24] S. Yu, C. Wang, K. Ren, and W. Lou, Attribute Based Data Sharing with Attribute Revocation, in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.

[25] R. Perlman, (2005).

[26] P. Anderson and L. Zhang, Fast and Secure Laptop Backups with Encrypted De-duplication, in *Proc. of USENIX LISA*, 2010.

[27] M. W. Storer, K. Greenan, D. D. E. Long, and E. L. Miller, Secure Data Deduplication, in *Proc. of StorageSS*, 2008.

[28] JungleDisk, `http://www.jungledisk.com/`, 2010.

[29] Nasuni, Nasuni Announces New Snapshot Retention Functionality in Nasuni Filer; Enables Fail-Safe File Deletion in the Cloud, 2011, http://www.nasuni.com/news/press-releases/nasuni-announces-new-snapshot-retention-functionality-in-nasuni-filer-enables-fail-safe-file-deletion-in-the-cloud/.

[30] B. Zhu, K. Li, and H. Patterson, Avoiding the Disk Bottleneck in the Data Domain Deduplication File System, in *Proc. of USENIX FAST*, 2008.

[31] S. Rhea, R. Cox, and A. Pesterev, Fast, Inexpensive Content-Addressed Storage in Foundation, in *Proc. of USENIX ATC*, 2008.

[32] A. Adya et al., FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment, in *Proc. of USENIX OSDI*, 2002.

[33] L. Cox, C. Murray, and B. Noble, Pastiche: Making backup cheap and easy, in *Proc. of USENIX OSDI*, 2002.

[34] C. Dubnicki et al., Hydrastor: a Scalable Secondary Storage, in *Proc. of USENIX FAST*, 2009.

[35] A. Clements, I. Ahmad, M. Vilayannur, and J. Li, Decentralized Deduplication in SAN Cluster File Systems, in *Proc. of USENIX ATC*, 2009.

[36] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup, in *Proc. IEEE MASCOTS*, pages 1–9, IEEE, 2009.

[37] J. Wei, H. Jiang, K. Zhou, and D. Feng, MAD2: A Scalable High-Throughput Exact Deduplication Approach for Network Backup Services, in *Proc. of IEEE MSST*, 2010.

[38] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, ZooKeeper: Wait-Free Coordination for Internet-Scale Systems, in *Proc. of USENIX ATC*, 2010.

[39] M. O. Rabin, Fingerprinting by random polynomials, Technical Report Tech. Report TR-CSE-03-01, Center for Research in Computing Technology, Harvard University, 1981.

[40] NIST, Advanced Encryption Standard, `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`, 2001, FIPS PUB 197.

[41] Trusted Computing Group, `http://www.trustedcomputinggroup.org/`.

[42] A. Shamir, CACM **22**, 612 (1979).

[43] OpenSSL, `http://www.openssl.org/`, 2010.

[44] Amazon S3, `http://aws.amazon.com/s3/`.

[45] rdiff-backup, `http://www.nongnu.org/rdiff-backup/`.

[46] L. Lamport, R. Shostak, and M. Pease, ACM Trans. on Programming Languages and Systems **4**, 382 (1982).

[47] X. Défago, A. Schiper, and P. Urbán, ACM Computing Surveys **36**, 372 (2004).

[48] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton, The Spread Toolkit: Architecture and Performance, Technical report, TR CNDS-2004-1, Johns Hopkins University, 2004.