A Secure Cloud Backup System with Assured Deletion and Version Control

Arthur Rahumed, Henry C. H. Chen, Yang Tang, Patrick P. C. Lee, and John C. S. Lui
The Chinese University of Hong Kong, Hong Kong
{arahumed,chchen,tangyang,pclee,cslui}@cse.cuhk.edu.hk

***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. However, cloud clients must enforce security guarantees of their outsourced data backups. We present *FadeVersion*, a secure cloud backup system that serves as a security layer on top of today's 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.**

I. 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.

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 [9], while in United Kingdom, companies are required to retain wages and salary records for six years [24].

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.

Thus, we are interested in 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. 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 other 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. This is one challenge we aim to overcome.

In this paper, we present *FadeVersion*, a 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.

We implement 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 [23] and include the assured deletion feature. Using Amazon S3 as the cloud storage backend, we empirically evaluate the performance of FadeVersion. We also conduct monetary cost analysis for FadeVersion based on the cost plans of different cloud providers. We show that the additional overhead of FadeVersion is justifiable compared to Cumulus, which does not possess the assured deletion functionality.

The remainder of the paper proceeds as follows. In Section II, we provide the necessary background on cloud storage

systems and the related technical issues. In Section III, we present the threat model and assumptions we make in our design. In Sections IV and V, we discuss the design and implementation details of FadeVersion, respectively. In Section VI, we evaluate the cost effectiveness and performance overhead of our system. Section VII concludes and presents future work.

II. Background and Related Work

There are different ways of achieving assured deletion. One approach is by *secure overwriting* [7], 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 [15]. 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 [3], [5], [6], [14], [21], [25]. The encrypted data blocks are stored in outsourced storage (e.g., clouds), while the cryptographic keys are kept independently by a key escrow system. For instance, FADE [21] supports policy-based assured deletion, in which data can be assuredly deleted according to revoked policies. However, existing studies do not consider version control for this approach. As shown in Section IV-A, existing version control systems and assured deletion systems are incompatible with each other.

Version control follows the notion of *deduplication* [16], which eliminates the storage of redundant data chunks that have the same content. In the security context, recent studies propose *convergent encryption* [2], [20], 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.

There are a few cloud backup systems in the market. Examples include commercial systems like Dropbox [4], Jungle Disk [8], and Nasuni [10], as well as the open-source Cumulus system [23], 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.

[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.

In March 2011, Nasuni announced that its system enables the new snapshot retention policy that allows assured deletion of backup snapshots [11]. On the other hand, there is no formal study about their implementation methodologies and performance evaluation. We address this issue in this paper. 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.

III. Threat Model and Assumptions

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.

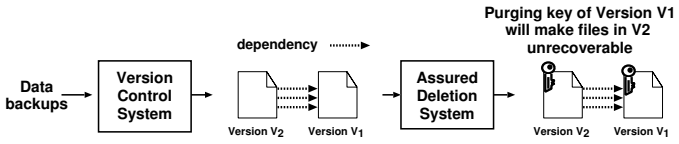
Our security goal is to achieve *assured deletion* of files for a cloud backup system with version control. We adopt the cryptographic approach [3], [5], [6], [14], [21], [25], i.e., by removing the keys that are used to decrypt the data backups stored on the cloud. We make two assumptions for this approach. 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 [7], which we believe is feasible given that the size of a key is much smaller compared to a backup file. In Section IV-F, we discuss in more detail the design of the key escrow system.

IV. Design of FadeVersion

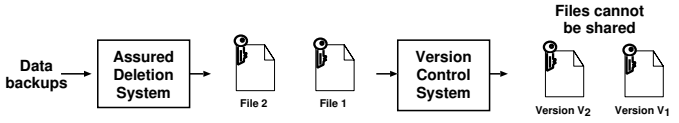
A. Motivation

We argue that existing version-controlled cloud backup systems (e.g., Cumulus [23]) and assured deletion systems (e.g., Vanish [6] and FADE [21]) 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 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



(a) First version control, followed by assured deletion



(b) First assured deletion, followed by version control

Fig. 1. Illustration of why existing version control systems and assured deletion systems are incompatible.

described in Section II. 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 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.

B. 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 [23], 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 subsections.

C. Version Control

In FadeVersion, each backup version (or *snapshot*) arranges data files into *file objects*. Each file object is of variable size

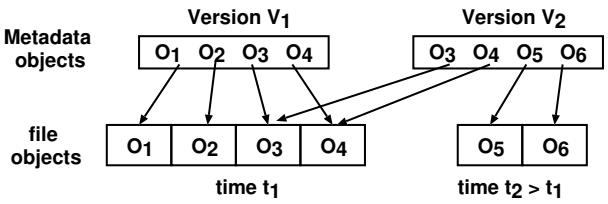


Fig. 2. Illustration of how version control works.

with a configurable maximum-size threshold (e.g., currently set as 1 MB). If a file has size less than the threshold, 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 [23].

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* [16] 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 [17]. However, we note that it does not always significantly improve the storage efficiency, such as using the datasets in our experiments (see Section VI). Therefore in this paper, we assume that an object is the smallest unit of data 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 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.

D. Assured Deletion

We now incorporate assured deletion into the version control design discussed in the previous subsections. 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 illustrates the idea. Denote $\{.\}k$ as the symmetric-key encryption (e.g., AES [12]) with key k.

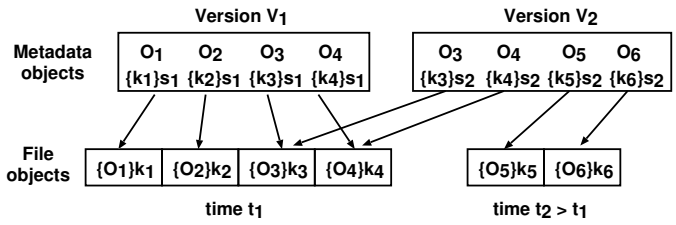


Fig. 3. Illustration of layered encryption, by extending the example shown in Figure 2.

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., 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 Section IV-F). 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, 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.

E. 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.

F. 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 Section III). 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 [22]). 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 [19]. 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.

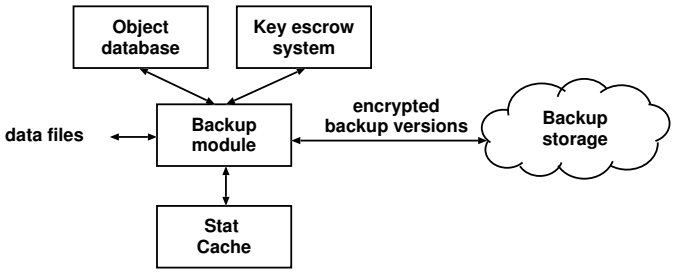


Fig. 4. Architecture of FadeVersion.

V. Implementation Details

We now present how FadeVersion is implemented to support both version control and assured deletion. FadeVersion is an extension of Cumulus [23], upon which we add new cryptographic implementation for assured deletion. The cryptographic operations are implemented with OpenSSL [13].

A. System Entities

FadeVersion is built on several system entities, as illustrated in Figure 4. 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 [1] as the storage backend. This can be easily extended to other third-party cloud storage 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 Section IV-E). 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 Section IV-F).

```
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)
```

Fig. 5. Metadata format for a single file: Cumulus (left) and FadeVersion (right).

Stat Cache [23]. It keeps metadata locally and is used to check if a file has been modified based on the modification time returned from the `stat()` system call. 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, instead of reading the objects of the unmodified file. The use of the stat cache can further improve the backup performance.

B. Metadata Format in FadeVersion

We use the metadata object to keep the information of all archived file objects in a backup version (see Section IV-C). FadeVersion extends the metadata format in Cumulus [23] to include the policy information and the encrypted cryptographic keys, both of which are used for assured deletion.

Figure 5 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 Section IV-E): (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 [12] 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 of the encrypted data key is fixed to be 128

TABLE I
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

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

VI. EXPERIMENTS

In this section, we conduct an empirical study on the prototype of FadeVersion. We compare FadeVersion with Cumulus [23]. 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.

A. 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 co-authors of this paper. Table I 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 6 shows the cumulative distribution functions of file sizes of the full snapshots on Day 1 and Day 46, respectively, and Figure 7 shows the size of data changes per day reported by `rdiff-backup` [18]. 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.

B. 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.

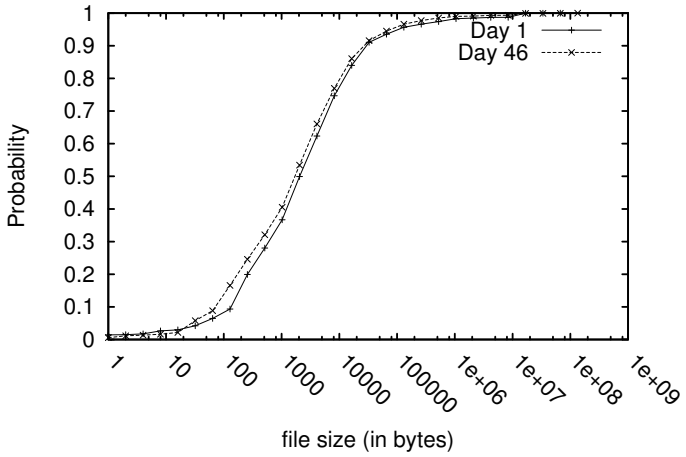


Fig. 6. File size statistics for Day 1 and Day 46.

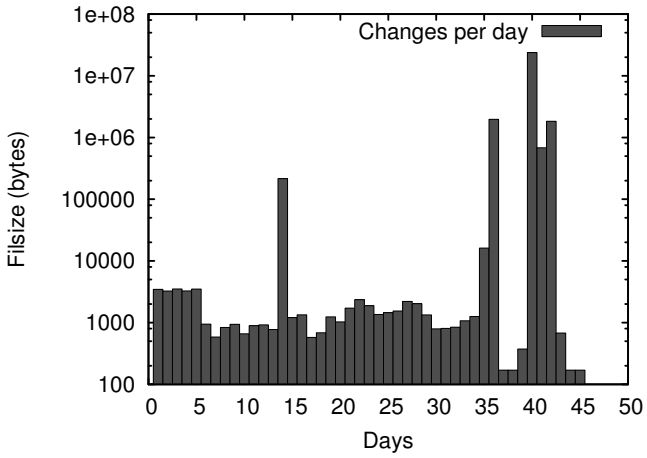


Fig. 7. Size of data changes reported by `rdiff-backup` per day.

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 8 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 9 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

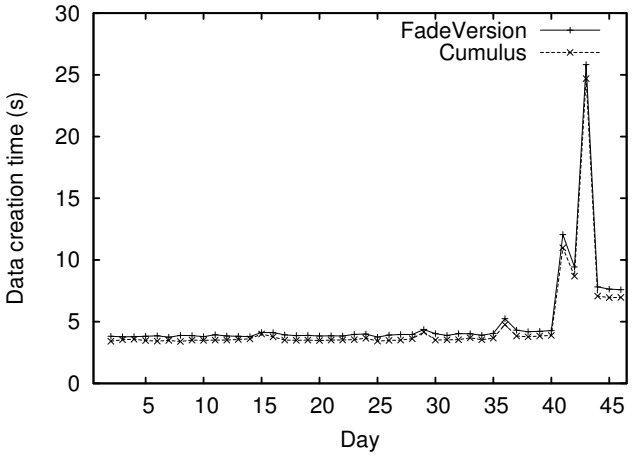


Fig. 8. Backup time for each incremental backup.

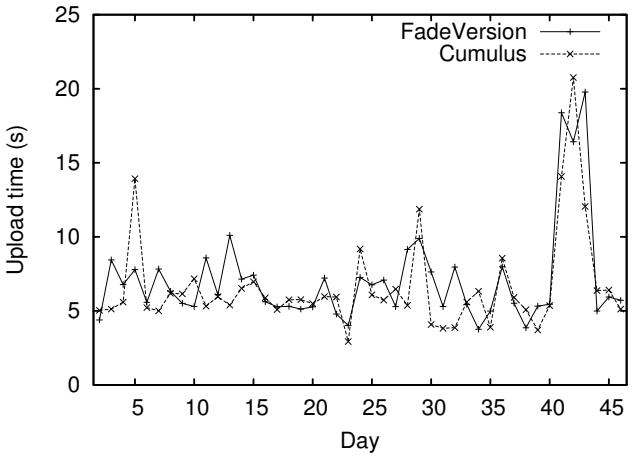


Fig. 9. Upload time for each incremental backup.

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 10 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.

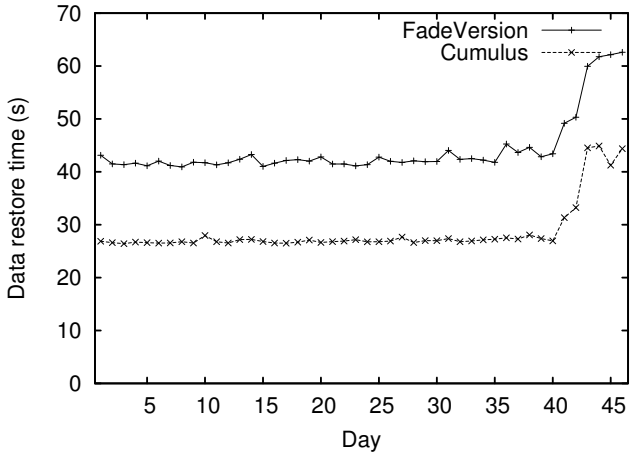


Fig. 10. Restore time for all 46 days of snapshots from local storage.

TABLE II
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

C. Storage Space

FadeVersion includes encrypted copies of data keys in data backups for assured deletion (see Section V-B), and this introduces storage space overhead. Here, we evaluate the space overhead of FadeVersion due to the storage of keys. Table II 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 I, mainly because both Cumulus and FadeVersion exploit deduplication to reduce the storage of redundant data (see Section IV-C). On average, FadeVersion introduces 19.4% more space increment per month compared to Cumulus.

We now focus on incremental backups. Figure 11 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 overhead on each day.

D. 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 III 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

TABLE III
STORAGE COSTS PER MONTH AND OVERALL BANDWIDTH COST OF CUMULUS AND FADEVERSION FOR 46 DAYS OF BACKUP WITH DIFFERENT CLOUD PROVIDERS.

Providers	Storage Cost $/GB/month	Cumulus	FadeVersion	Bandwidth Cost for updates$/GB	Cumulus	FadeVersion
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

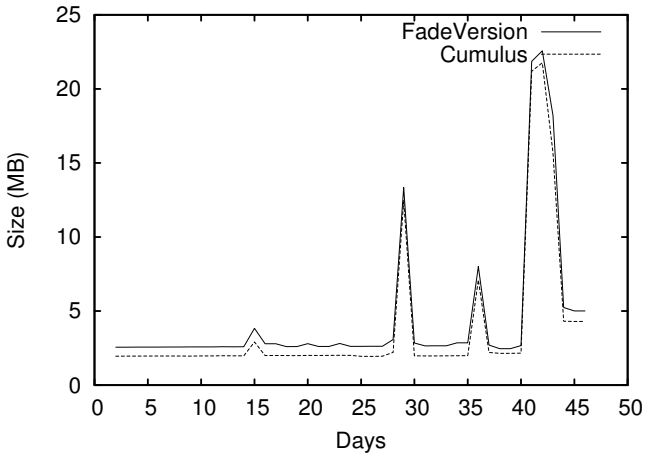


Fig. 11. Size of incremental uploads for Cumulus and FadeVersion.

its additional bandwidth cost is within $0.003. The monetary cost overhead of FadeVersion is minimal in general.

VII. CONCLUSIONS AND FUTURE WORK

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.

ACKNOWLEDGMENT

This research is supported by project #MMT-p1-10 of the Shun Hing Institute of Advanced Engineering, The Chinese University of Hong Kong. Arthur Rahumed and Patrick P. C. Lee are affiliated with SHIAE in this research.

REFERENCES

[1] Amazon S3. http://aws.amazon.com/s3/.
[2] P. Anderson and L. Zhang. Fast and Secure Laptop Backups with Encrypted De-duplication. In *Proc. of USENIX LISA*, 2010.
[3] D. Boneh and R. Lipton. A Revocable Backup System. In *Proc. of USENIX Security Symposium*, 1996.
[4] Dropbox. http://www.dropbox.com, 2010.
[5] 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.
[6] R. Geambasu, T. Kohno, A. Levy, and H. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proc. of USENIX Security Symposium*, 2009.
[7] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proc. of USENIX Security Symposium*, 1996.
[8] JungleDisk. http://www.jungledisk.com/, 2010.
[9] D. McCullagh. Fbi, politicos renew push for isp data retention laws. http://news.cnet.com/8301-13578_3-9926803-38.html, Apr 2008.
[10] Nasuni. http://www.nasuni.com.
[11] Nasuni. Nasuni Announces New Snapshot Retention Functionality in Nasuni Filer; Enables Fail-Safe File Deletion in the Cloud, Mar 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/.
[12] NIST. Advanced Encryption Standard. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, Nov 2001. FIPS PUB 197.
[13] OpenSSL. http://www.openssl.org/, 2010.
[14] R. Perlman. File System Design with Assured Delete. In *ISOC NDSS*, 2007.
[15] 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.
[16] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proc. USENIX FAST*, 2002.
[17] M. O. Rabin. Fingerprinting by random polynomials. Technical Report Tech. Report TR-CSE-03-01, Center for Research in Computing Technology, Harvard University, 1981.
[18] rdiff-backup. http://www.nongnu.org/rdiff-backup/.
[19] A. Shamir. How to Share a Secret. *CACM*, 22(11):612–613, Nov 1979.
[20] M. W. Storer, K. Greenan, D. D. E. Long, and E. L. Miller. Secure Data Deduplication. In *Proc. of StorageSS*, 2008.
[21] Y. Tang, P. Lee, J. Lui, and R. Perlman. FADE: Secure Overlay Cloud Storage with File Assured Deletion. In *Proc. of SecureComm*, 2010.
[22] Trusted Computing Group. http://www.trustedcomputinggroup.org/.
[23] M. Vrable, S. Savage, and G. Voelker. Cumulus: Filesystem backup to the cloud. In *Proc. of USENIX FAST*, 2009.
[24] Watson Hall Ltd. Uk data retention requirements. https://www.watsonhall.com/resources/downloads/paper-uk-data-retention-requirements.pdf, 2009.
[25] 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)*, Apr 2010.