Secure Overlay Cloud Storage with Access Control and Assured Deletion

Yang Tang, Patrick P. C. Lee, John C. S. Lui, Radia Perlman

Abstract—We can now outsource data backups off-site to third-party cloud storage services so as to reduce data management costs. However, we must provide security guarantees for the outsourced data, which is now maintained by third parties. We design and implement *FADE*, a secure overlay cloud storage system that achieves fine-grained, policy-based access control and file assured deletion. It associates outsourced files with file access policies, and assuredly deletes files to make them unrecoverable to anyone upon revocations of file access policies. To achieve such security goals, FADE is built upon a set of cryptographic key operations that are self-maintained by a quorum of key managers that are independent of third-party clouds. In particular, FADE acts as an overlay system that works seamlessly atop today's cloud storage services. We implement a proof-of-concept prototype of FADE atop Amazon S3, one of today's cloud storage services. We conduct extensive empirical studies, and demonstrate that FADE provides security protection for outsourced data, while introducing only minimal performance and monetary cost overhead. Our work provides insights of how to incorporate value-added security features into today's cloud storage services.

Keywords—access control, assured deletion, backup/recovery, cloud storage

1 INTRODUCTION

Cloud storage is a new business solution for remote backup outsourcing, as it offers an abstraction of infinite storage space for clients to host data backups in a pay-as-you-go manner [5]. It helps enterprises and government agencies significantly reduce their financial overhead of data management, since they can now archive their data backups remotely to third-party cloud storage providers rather than maintain data centers on their own. For example, SmugMug [30], a photo sharing website, chose to host terabytes of photos on Amazon S3 in 2006 and saved thousands of dollars on maintaining storage devices [3]. More case studies of using cloud storage for remote backup can be found in [2]. Apart from enterprises and government agencies, individuals can also archive their personal data to the cloud using tools like Dropbox [10]. In particular, with the advent of smartphones, we expect that more people will use Dropbox-like tools to move audio/video files from their smartphones to the cloud, given that smartphones typically have limited storage resources.

However, security concerns become relevant as we now outsource the storage of possibly sensitive data to third parities. In this paper, we are particularly interested in two security issues. First, we need to provide guarantees of *access control*, in which we must ensure that only authorized parties can access the outsourced data on the cloud. In particular, we must prohibit third-party cloud storage providers from mining any sensitive information of their clients' data for their own marketing purposes. Second, it is important to provide guarantees of *assured deletion*, meaning that outsourced data is permanently inaccessible to anybody (including the data owner) upon requests of deletion of data. Keeping data permanently is undesirable, as data may be unexpectedly disclosed in the future due to malicious attacks on the cloud or careless management of cloud operators. The challenge of achieving assured deletion is that we have to trust cloud storage providers to actually delete data, but they may be reluctant in doing so [28]. Also, cloud storage providers typically keep multiple backup copies of data for fault-tolerance reasons. It is uncertain, from cloud clients' perspectives, whether cloud providers reliably remove all backup copies upon requests of deletion.

The security concerns motivate us, as cloud clients, to have a system that can enforce access control and assured deletion of outsourced data on the cloud *in a fine-grained manner*. However, building such a system is a difficult task, especially when it involves protocol or hardware changes in cloud storage infrastructures that are externally owned and managed by third-party cloud providers. Thus, it is necessary to design a secure *overlay* cloud storage system that can be overlaid and work seamlessly atop existing cloud storage services.

In this paper, we present FADE, a secure overlay cloud storage system that provides fine-grained access control and assured deletion for outsourced data on the cloud, while working seamlessly atop today's cloud storage services. In FADE, active data files that remain on the cloud are associated with a set of user-defined *file access policies* (e.g., time expiration, read/write permissions of authorized users), such that data files are accessible only to users who satisfy the file access policies. In addition, FADE generalizes time-based file assured deletion [12], [23] (i.e.,

- *Y. Tang, P. P. C. Lee, and J. C. S. Lui are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong (emails: {tangyang,pclee,cslui}@cse.cuhk.edu.hk).*
- *Radia Perlman is with Intel Labs (email: radiaperlman@gmail.com).*
- *An earlier conference version appeared in [33].*

data files are assuredly deleted upon time expiration) into a more fine-grained approach called *policy-based file assured deletion*, in which data files are assuredly deleted when the associated file access policies are revoked and become obsolete. The design intuition of FADE is to decouple the management of encrypted data and cryptographic keys, such that encrypted data remains on third-party (untrusted) cloud storage providers, while cryptographic keys are independently maintained and operated by a quorum of key managers that altogether form trustworthiness. To provide guarantees of access control and assured deletion, FADE leverages off-the-shelf cryptographic schemes including threshold secret sharing [29] and attribute-based encryption [7], [13], [25], [27], and performs various cryptographic key operations that provide security protection for basic file upload/download operations. We implement a proof-of-concept prototype of FADE to justify its feasibility, and export a set of library APIs that can be used, as a value-added security service, to enhance the security properties of general data outsourcing applications.

We point out that the design of FADE is based on the *thin-cloud* interface [35], meaning that it only requires the cloud to support the basic data access operations such as `put` and `get`. Thus, FADE is applicable for general types of storage backends, as long as such backends provide the interface for uploading and downloading data. On the other hand, in the context of cloud storage, we need to specifically consider the performance metrics that are inherent to cloud storage, including data transmission performance (given that a cloud is deployed over the Internet) and monetary cost overhead (given that a cloud charges clients for data outsourcing). Thus, we empirically evaluate FADE according to the specific features of cloud storage, so as to justify that FADE is actually a feasible solution for secure cloud storage.

Our contributions are summarized as follows.

- We propose a new *policy-based file assured deletion* scheme that reliably deletes files with regard to revoked file access policies. In this context, we design the key management schemes for various file manipulation operations, with the emphasis on fine-grained security protection.
- On top of policy-based file assured deletion, we design and implement two new features: (i) fine-grained access control based on attribute-based encryption and (ii) fault-tolerant key management with a quorum of key managers based on threshold secret sharing.
- We implement a working prototype of FADE atop Amazon S3. Our implementation of FADE exports a set of APIs that can be adapted into different data outsourcing applications.
- We empirically evaluate the performance overhead of FADE atop Amazon S3. Using experiments in a realistic network environment, we show the feasibility of FADE in improving the security protection of data storage on the cloud in practice. We also analyze the monetary cost overhead of FADE under a practical cloud backup scenario.

In summary, our work seeks to address the access control and assured deletion problems from a practical perspective. Our FADE implementation characterizes and evaluates the performance and monetary cost implications of applying access control and assured deletion in a real-life cloud storage environment.

The remainder of the paper proceeds as follows. In Section 2, we describe and motivate the concept of policy-based file assured deletion, a major building block of FADE. In Section 3, we overview the design of FADE, and define our design goals. In Section 4, we explain the design of FADE in achieving access control and assured deletion. In Section 5, we describe the implementation details of FADE. In Section 6, we evaluate FADE atop Amazon S3. Section 7 reviews related work on securing outsourced data storage. Finally, Section 8 concludes.

2 Policy-based File Assured Deletion

FADE seeks to achieve both access control and assured deletion for outsourced data. The design of FADE is centered around the concept of *policy-based file assured deletion*. We first review time-based file assured deletion proposed in earlier work. We then explain the more general concept policy-based file assured deletion and motivate why it is important in certain scenarios.

2.1 Background

Time-based file assured deletion, which is first introduced in [23], means that files can be securely deleted and remain permanently inaccessible after a pre-defined duration. The main idea is that a file is encrypted with a *data key* by the owner of the file, and this data key is further encrypted with a *control key* by a separate key manager (known as *Ephemerizer* [23]). The key manager is a server that is responsible for cryptographic key management. In [23], the control key is *time-based*, meaning that it will be completely removed by the key manager when an expiration time is reached, where the expiration time is specified when the file is first declared. Without the control key, the data key and hence the data file remain encrypted and are deemed to be inaccessible. Thus, the main security property of file assured deletion is that even if a cloud provider does not remove expired file copies from its storage, those files remain encrypted and unrecoverable.

An open issue in the work [23] is that it is uncertain that whether time-based file assured deletion is feasible in practice, as there is no empirical evaluation. Later, the idea of time-based file assured deletion is prototyped in Vanish [12]. Vanish divides a data key into multiple key shares, which are then stored in different nodes of a public Peer-to-Peer Distributed Hash Table (P2P DHT) system. Nodes remove the key shares that reside in their caches for a fixed time period. If a file needs to remain accessible after the time period, then the file owner needs to update the key shares in node caches. Since Vanish is built on the cache-aging mechanism in the P2P DHT, it is difficult to generalize the idea from time-based deletion to a fine-grained control of assured deletion with respect to different file access policies. We elaborate this issue in the following section.

2.2 Policy-based Deletion

We now generalize time-based deletion to policy-based deletion as follows. We associate each file with a single atomic *file access policy* (or *policy* for short), or more generally, a Boolean combination of atomic policies. Each (atomic) policy is associated with a control key, and all the control keys are maintained by the key manager. Suppose now that a file is associated with a single policy. Then similar to time-based deletion, the file content is encrypted with a data key, and the data key is further encrypted with the control key corresponding to the policy. When the policy is revoked, the corresponding control key will be removed from the key manager. Thus, when the policy associated with a file is revoked and no longer holds, the data key and hence the encrypted content of the file cannot be recovered with the control key of the policy. In this case, we say the file is assuredly deleted. The main idea of policy-based deletion is to delete files that are associated with revoked policies.

The definition of a policy varies across applications. In fact, time-based deletion is a special case under our framework. In general, policies with other access rights can be defined. To motivate the use of policy-based deletion, let us consider a scenario where a company outsources its data to the cloud. We consider four practical cases where policy-based deletion will be useful.

Storing files for tenured employees. For each employee (e.g., Alice), we can define a *user-based* policy "*P: Alice is an employee*", and associate this policy with all files of Alice. If Alice quits her job, then the key manager will expunge the control key of policy P. Thus, nobody including Alice can access the files associated with P on the cloud, and those files are said to be deleted.

Storing files for contract-based employees. An employee may be affiliated with the company for only a fixed length of time. Then we can form a combination of the user-based and time-based policies for employees' files. For example, for a contract-based employee Bob whose contract expires on 2010-01-01, we have two policies "*P_1: Bob is an employee*" and "*P_2: valid before 2010-01-01*". Then all files of Bob are associated with the policy combination $P_1 \wedge P_2$. If either P_1 or P_2 is revoked, then Bob's files are deleted.

Storing files for a team of employees. The company may have different teams, each of which has more than one employee. As in above, we can assign each employee i a policy combination $P_{i1} \wedge P_{i2}$, where P_{i1} and P_{i2} denote the user-based and time-based policies, respectively. We then associate the team's files with the disjunctive combination $(P_{11} \wedge P_{12}) \vee (P_{21} \wedge P_{22}) \vee \cdots \vee (P_{N1} \wedge P_{N2})$ for employees $1, 2, \ldots, N$. Thus, the team's files can be accessed by any one of the employees, and will be deleted when the policies of all employees of the team are revoked.

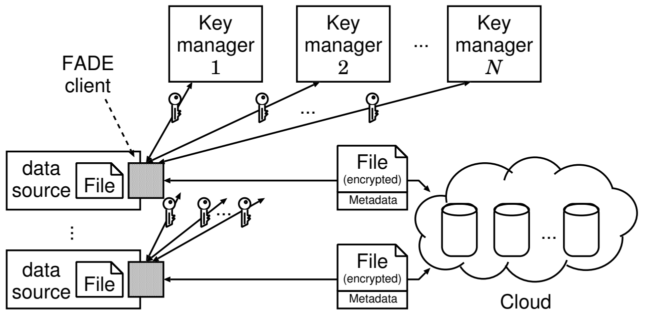


Fig. 1: The FADE system. Each client (deployed locally with its own data source) interacts with one or multiple key managers and uploads/downloads data files to/from the cloud.

Switching a cloud provider. The company can define a *customer-based* policy "*P: a customer of cloud provider X*", and all files that are stored on cloud X are tied with policy P. If the company switches to a new cloud provider, then it can revoke policy P. Thus, all files on cloud X will be deleted. We argue that switching cloud providers has its potential application, such as in avoiding vendor lock-ins [1].

3 FADE Overview

We now overview the design of FADE, a system that provides guarantees of access control and assured deletion for outsourced data in cloud storage. We present the necessary components of FADE, and state the design and security goals that it seeks to achieve.

Figure 1 illustrates an overview of the FADE system. The cloud hosts data files on behalf of a group of FADE users who want to outsource data files to the cloud based on their definitions of file access policies. FADE can be viewed as an overlay system atop the underlying cloud. It applies security protection to the outsourced data files before they are hosted on the cloud.

3.1 Entities

As shown in Figure 1, the FADE system is composed of two main entities:

- **FADE clients.** A *FADE client* (or *client* for short) is an interface that bridges the data source (e.g., filesystem) and the cloud. It applies encryption (decryption) to the outsourced data files uploaded to (downloaded from) the cloud. It also interacts with the key managers to perform the necessary cryptographic key operations.
- **Key managers**. FADE is built on a quorum of key managers [29], each of which is a stand-alone entity that maintains policy-based keys for access control and assured deletion (see Sections 3.3 and 3.4).

The cloud, maintained by a third-party provider, provides storage space for hosting data files on behalf of different FADE clients in a pay-as-you-go manner. Each of the data files is associated with a combination of file access policies. FADE is built on the thin-cloud interface [35], and assumes only the basic cloud operations for uploading and downloading data files. We emphasize that we do *not* require any protocol and implementation changes on the cloud to support FADE.

3.2 Deployment

In our current design, a FADE client is deployed locally with its corresponding data source as a local driver or daemon. Note that it is also possible to deploy the FADE client as a cloud storage proxy [1], so that it can interconnect multiple data sources. In proxy deployment, we can use standard TLS/SSL [9] to protect the communication between each data source and the proxy.

In FADE, the set of key managers is deployed as a centralized trusted service, whose trustworthiness is enforced through a quorum scheme. We assume that the key managers are centrally maintained, for example, by the system administrators of an enterprise that deploys FADE for its employees. We note that this centralized control is opposed to the core design of Vanish [12], which proposes to use decentralized key management on top of existing P2P DHT systems. However, as discussed in Section 2, there is no straightforward solution to develop fine-grained cryptographic key management operations over a decentralized P2P DHT system. Also, the Vanish implementation that was published in [12] is subject to Sybil attacks [38], which particularly target DHT systems. In view of this, we propose to deploy a centralized key management service, and use a quorum scheme to improve its robustness.

3.3 Cryptographic Keys

FADE defines three types of cryptographic keys to protect data files stored on the cloud:

- **Data key.** A data key is a random secret that is generated and maintained by a FADE client. It is used for encrypting or decrypting data files via symmetric-key encryption (e.g., AES).
- **Control key.** A control key is associated with a particular policy. It is represented by a public-private key pair, and the private control key is maintained by the quorum of key managers. It is used to encrypt/decrypt the data keys of the files protected with the same policy. The control key forms the basis of policy-based assured deletion.
- **Access key.** Similar to the control key, an access key is associated with a particular policy, and is represented by a public-private key pair. However, unlike the control key, the private access key is maintained by a FADE client that is authorized to access files of the associated policy. The access key is built on attribute-based encryption [7], and forms the basis of policy-based access control.

Intuitively, to successfully decrypt an encrypted file stored on the cloud, we require the correct data key, control key, and access key. Without any of these keys, it is computationally infeasible to recover an outsourced file being protected by FADE. The following explains how we manage such keys to achieve our security goals.

3.4 Security Goals

We formally state the security goals that FADE seeks to achieve in order to protect the outsourced data files.

Threat model. Here, we consider an adversary that seeks to compromise the privacy of two types of files that are outsourced and stored on the cloud: (i) *active files*, i.e., the data files that the adversary is unauthorized to access and (ii) *deleted files*, i.e., the data files that have been requested to be deleted by the authorized parties. Clearly, FADE needs to properly encrypt outsourced data files to ensure that their information is not disclosed to unauthorized parties. The underlying assumption is that the encryption mechanism is secure, such that it is computationally infeasible to recover the encrypted content without knowing the cryptographic key for decryption.

Security properties. Given our threat model, we focus on two specific security goals that FADE seeks to achieve for fine-grained security control:

- **Policy-based access control.** A FADE client is authorized to access only the files whose associated policies are active and are satisfied by the client.
- **Policy-based assured deletion.** A file is deleted (or permanently inaccessible) if its associated policies are revoked and become obsolete. That is, even if a file copy that is associated with revoked policies exists, it remains encrypted and we cannot retrieve the corresponding cryptographic keys to recover the file. Thus, the file copy becomes unrecoverable by anyone (including the owner of the file).

Assumptions. To achieve the above security goals, we make the following assumptions regarding the key management in FADE. FADE is built on a quorum of key managers based on Shamir's (M, N) threshold secret sharing [29], in which we create N key shares for a key, such that any $M \leq N$ of the key shares can be used to recover the key. First, to access files associated with active policies, we require at least M out of N key managers keep the key shares of the required control keys and correctly perform the cryptographic key operations. Second, to assuredly delete files, at least $N-M+1$ out of N key managers must securely erase the corresponding control keys of the revoked policies, such as via secure overwriting [14], which we believe is feasible for smaller-size keys as opposed to larger-size data files.

The parameters M and N determine the trade-off between the fault tolerance assumptions of key managers when accessing and deleting files. If M is small (large), then we need fewer (more) key managers to be active in order to access a file, but we need more (fewer) key managers to purge the revoked control keys in order to delete a file. How to adjust the trade-off depends on different application needs.

We assume that a key manager is subject to only *fail-stop failures* (e.g., system crashes, data losses), so the quorum scheme enables our key management to be robust against fail-stop failures. On the other hand, we do not consider the case of arbitrary (or Byzantine) failures in key managers (e.g., tampering with control keys or policies), where the security threats are beyond the scope of this work. We pose the protection against arbitrary failures in future studies.

In addition, we require that each FADE client does not keep the raw copy of the data key that is used to protect a data file. Once it successfully encrypts or decrypts a data file, it must discard the raw copy of the corresponding data key; otherwise, the file may be recoverable should the raw copy of the data key be disclosed. Only the encrypted copy of the data key, together with the encrypted data files, will be kept and stored on the cloud.

In our threat model, we only focus on protecting the data files stored on the cloud. Therefore, we do not consider the case where the FADE client discloses the successfully decrypted data files that are retrieved from the cloud, as such files are outside our protection scope.

4 FADE DESIGN

In this section, we present the design of FADE. In particular, we propose several cryptographic key operations that enable FADE to achieve our security goals.

4.1 Basic Operations of FADE

We start with the basic design of FADE. To simplify our discussion, we make two assumptions. First, only a single key manager is used. Second, before accessing a file, a client needs to present authentication credentials (e.g., based on public key infrastructure certificates) to the key manager to show that it satisfies the proper policies associated with the files, so that the key manager will perform cryptographic key operations. We explain in Section 4.2 how to relax both of the assumptions through multiple key managers with threshold secret sharing and access control with attribute-based encryption.

4.1.1 File Upload/Download

We now introduce the basic operations of how a client uploads/downloads files to/from the cloud. We start with the case where each file is associated with a single policy, and then explain how a file is associated with multiple policies in Section 4.1.3.

Our design is based on *blinded RSA* [32] (or blinded decryption [23]), in which the client requests the key manager to decrypt a blinded version of the encrypted data key. If the associated policy is satisfied, then the key manager will decrypt and return the blinded version of the original data key. The client can then recover the data key. The motivation of using this blinded decryption approach is that the actual content of the data key remains confidential to the key manager as well as to any attacker that sniffs the communication between the client and the key manager.

Notation	Description
F	Data file generated by the client
K	Data key used to encrypt file F
P_i	Policy with index i
p_i, q_i	RSA prime numbers for policy P_i (kept secret by the key manager)
n_i	$n_i = p_i q_i$, known to the public
(e_i, d_i)	RSA public/private control key pair for policy P_i
S_i	Secret key corresponding to policy P_i
$\{.\}_{KEY}$	Symmetric-key encryption with key KEY
R	The random number used for blinded RSA

TABLE 1: Notation used in this paper.

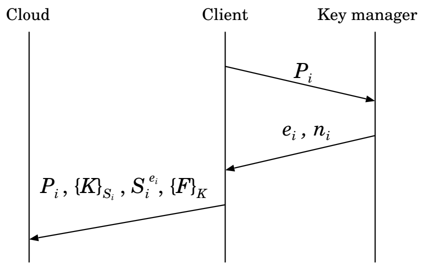


Fig. 2: File upload.

Table 1 summarizes the notation used in this paper. We first summarize the major notation used throughout the paper. For each policy i, the key manager generates two secret large RSA prime numbers p_i and q_i and computes the product $n_i = p_i q_i$[1]. The key manager then randomly chooses the RSA public-private control key pair (e_i, d_i). The parameters (n_i, e_i) will be publicized, while d_i is securely stored in the key manager. On the other hand, when the client encrypts a file F, it randomly generates a data key K, and a secret key S_i that corresponds to policy P_i. We let $\{.\}_{KEY}$ denote the symmetric-key encryption operation (e.g., AES) with key KEY. We let R be the blinded component when we use blinded RSA for the exchanges of cryptographic keys.

Suppose that F is associated with policy P_i. Our goal here is to ensure that K, and hence F, are accessible only when policy P_i is satisfied. Note that we only present the operations on cryptographic keys, while the implementation subtleties, such as the metadata that stores the policy information, will be discussed in Section 5. Also, when we raise some number to exponents e_i or d_i, it must be done over modulo n_i. For brevity, we drop "mod n_i" in our discussion.

File upload. Figure 2 shows the file upload operation.

1. We require that each policy i uses a distinct n_i to avoid the common modulus attack on RSA [19].

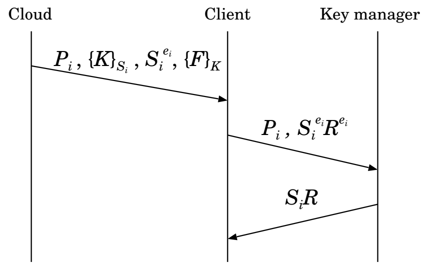

Cloud
Client
Key manager
P_i, $\{K\}_{S_i}$, $S_i^{e_i}$, $\{F\}_K$
P_i, $S_i^{e_i}R^{e_i}$
S_iR

Fig. 3: File download.

The client first requests the public control key (n_i, e_i) of policy P_i from the key manager, and caches (n_i, e_i) for subsequent uses if the same policy P_i is associated with other files. Then the client generates two random keys K and S_i, and sends $\{K\}_{S_i}$, $S_i^{e_i}$, and $\{F\}_K$ to the cloud[2]. Then the client must discard K and S_i (see Section 3.4 for justifications). To protect the integrity of a file, the client computes an HMAC signature on every encrypted file and stores the HMAC signature together with the encrypted file in the cloud. We assume that the client has a long-term private secret value for the HMAC computation.

File download. Figure 3 shows the file download operation. The client fetches $\{K\}_{S_i}$, $S_i^{e_i}$, and $\{F\}_K$ from the cloud. The client will first check whether the HMAC signature is valid before decrypting the file. Then the client generates a secret random number R, computes R^{e_i}, and sends $S_i^{e_i} \cdot R^{e_i} = (S_iR)^{e_i}$ to the key manager to request for decryption. The key manager then computes and returns $((S_iR)^{e_i})^{d_i} = S_iR$ to the client, which can now remove R and obtain S_i, and decrypt $\{K\}_{S_i}$ and hence $\{F\}_K$.

4.1.2 Policy Revocation for File Assured Deletion

If a policy P_i is revoked, then the key manager completely removes the private control key d_i and the secret prime numbers p_i and q_i. Thus, we cannot recover S_i from $S_i^{e_i}$, and hence cannot recover K and file F. We say that file F, which is tied to policy P_i, is assuredly deleted. Note that the policy revocation operations do not involve interactions with the cloud.

4.1.3 Multiple Policies

FADE supports a Boolean combination of multiple policies. We mainly focus on two kinds of logical connectives: (i) the conjunction (AND), which means the data is accessible only when every policy is satisfied; and (ii) the disjunction (OR), which means if any policy is satisfied, then the data is accessible. Our following operations on a Boolean combination of policies are similar to those in [24], while the focus of [24] is on digital rights management rather than file assured deletion.

- **Conjunctive Policies.** Suppose that F is associated with conjunctive policies $P_1 \wedge P_2 \wedge \cdots \wedge P_m$. To upload F to the cloud, the client first randomly generates a data key K, and different secret keys $S_1, S_2, \ldots, S_m$. It then sends the following to the cloud: $\{\{K\}_{S_1}\}_{S_2} \cdots_{S_m}$, $S_1^{e_1}$, $S_2^{e_2}$, …, $S_m^{e_m}$, and $\{F\}_K$. On the other hand, to recover F, the client generates a random number R and sends $(S_1R)^{e_1}$, $(S_2R)^{e_2}$, …, $(S_mR)^{e_m}$ to the key manager, which then returns $S_1R, S_2R, \ldots, S_mR$. The client can then recover $S_1, S_2, \ldots, S_m$, and hence K and F.
- **Disjunctive Policies.** Suppose that F is associated with disjunctive policies $P_{i_1} \vee P_{i_2} \vee \cdots \vee P_{i_m}$. To upload F to the cloud, the client will send the following: $\{K\}_{S_1}, \{K\}_{S_2}, \ldots, \{K\}_{S_m}, S_1^{e_1}, S_2^{e_2}, \ldots, S_m^{e_m}$, and $\{F\}_K$. Therefore, the client needs to compute m different encrypted copies of K. On the other hand, to recover F, we can use any one of the policies to decrypt the file, as in the above operations.

To delete a file associated with conjunctive policies, we simply revoke any of the policies (say, P_j). Thus, we cannot recover S_j and hence the data key K and file F. On the other hand, to delete a file associated with disjunctive policies, we need to revoke all policies, so that $S_j^{e_j}$ cannot be recovered for all j. Note that for any Boolean combination of policies, we can express it in canonical form, e.g., in the disjunction (OR) of conjunctive (AND) policies.

4.1.4 Policy Renewal

We conclude this section with the discussion of policy renewal. Policy renewal means to associate a file with a new policy (or combination of policies). For example, if a user wants to extend the expiration time of a file, then the user can update the old policy that specifies an earlier expiration time to the new policy that specifies a later expiration time.

In FADE, policy renewal merely operates on keys, *without retrieving the encrypted file from the cloud.* The procedures can be summarized as follows: (i) download all encrypted keys (including the data key for the file and the set of control keys for the associated Boolean combination of policies) from the cloud, (ii) send them to the key manager for decryption, (iii) recover the data key, (iv) re-encrypt the data key with the control keys of the new Boolean combination of policies, and finally (v) send the newly encrypted keys back to the cloud.

In some special cases, we can simplify the key operations of policy renewal. Suppose that the Boolean combination structure of policies remains unchanged, but one of the atomic policies P_i is changed to P_j. For example, when we extend the contract date of Bob (see Section 2.2), we may need to update the particular time-based policy of Bob without changing other policies. Then instead of decrypting and re-encrypting the data key with the control keys that correspond to the new Boolean combination of policies, we can simply update the control key that corresponds to the particular atomic policy. Figure 4 illustrates this special case of policy renewal. In this case, the client simply sends the blinded

2. We point out that the encrypted keys (i.e., $\{K\}_{S_i}$, $S_i^{e_i}$) can be stored in the cloud without creating risks of leaking confidential information.

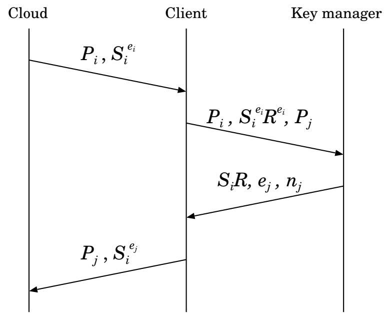


Fig. 4: A special case of policy renewal - when policy P_i is renewed to policy P_j.

version ${S_i}^{e_i}R^{e_i}$ to the key manager, which then returns S_iR. The client then recovers S_i. Now, the client re-encrypts S_i into $S_i^{e_j} \pmod{n_j}$, where (n_j, e_j) is the public control key of policy P_j, and sends it to the cloud. Note that the encrypted data key K remains intact.

4.2 Extensions of FADE

We now discuss two extensions to the basic design of FADE. The first extension is to use *attribute-based encryption* (ABE) [27], [7], [13], [25] in order to authenticate clients through policy-based access control. The second extension is to use threshold secret sharing [29] in order to achieve better reliability for key management.

By no means do we claim the protocol designs of ABE-based access control and threshold secret sharing are our contributions. Instead, our contribution here is to demonstrate a proof of applicability of existing security mechanisms in a real-life cloud storage environment, and characterize their performance overheads via our empirical experiments (see Section 6).

4.2.1 Access Control with ABE

To recover a file from the cloud, a client needs to request the key manager (assuming that only a single key manager is deployed) to decrypt the data key. The client needs to present authentication credentials to the key manager to show that it indeed satisfies the policies associated with the files. One implementation approach for this authentication process is based on the public-key infrastructure (PKI). However, this client-based authentication requires the key manager to have accesses to the association of *every* client and its satisfied policies. This limits the scalability and flexibility if we scale up the number of supported clients and their associations with policies.

To resolve the scalability issue, *attribute-based encryption (ABE)* [27], [7], [13], [25] turns out to be the most appropriate solution (see Section 2.2). In particular, our approach is based on *Ciphertext-Policy Attribute-Based Encryption (CP-ABE)* [7]. We summarize the essential ideas of ABE that are sufficient for our FADE design, while we refer readers to [7] for details. Each client first obtains, from the key issuing authority of the ABE system, an ABE-based private *access key* that corresponds to a set of attributes[3] the client satisfies. This can be done by having the client present authentication credentials to the key issuing authority, but we emphasize that this authentication is only a one-time bootstrap process. Later, when a client requests the key manager to decrypt the data key of a file on the cloud, the key manager will encrypt the response messages using the ABE-based public access key that corresponds to the combination of policies associated with the file. If the client indeed satisfies the policy combination, then it can use its ABE-based private access key to recover the data key. Note that the key manager does not have to know exactly each individual client who requests decryption of a data key.

FADE uses two independent keys for each policy. The first one is the private control key that is maintained by the key manager for assured deletion. If the control key is removed from the key manager, then the client cannot recover the files associated with the corresponding policy. Another one is the ABE-based access key that is used for access control. The ABE-based private access key is distributed to the clients who satisfy the corresponding policy, as in the ABE approach, while the key manager holds the ABE-based public access key and uses it to encrypt the response messages returned to the clients. The use of the two sets of keys for the same policy enables FADE to achieve both access control and assured deletion.

We now modify the FADE operations to include the ABE feature as follows. We assume that we operate on a file that is associated with a single policy.

File Upload. The file upload operation remains unchanged, since we only need the public parameters from the key manager for this operation, and hence we do not need to authenticate the client.

File Download. The file download operation requires authentication of the client. When the client requests the key manager to decrypt $S_i^{e_i}R^{e_i}$, the key manager encrypts its answer S_iR with ABE based on the policy of the file. Therefore, if the client satisfies the policy, then it can decrypt the response message and get S_iR.

Policy Renewal. Similar to above, the key manager encrypts S_iR with ABE when the client requests it to decrypt the old policy. For the re-encryption with the new policy, there is no need to enforce access control since we only need the public parameters.

Policy Revocation. Here we use a challenge-response mechanism in order for the key manager to authenticate the client. Figure 5 shows the revised policy revocation protocol. In the first round, the client tells the key manager that it wants to revoke policy P_i. The key manager then generates a random number r as a challenge, encrypts it with ABE that corresponds to policy P_i, and gives it to the client. Next, if the client is genuine, then it can decrypt r and send its hash to the key manager as

3. An attribute is equivalent to an atomic policy that we define for policy-based file assured deletion (see Section 2).

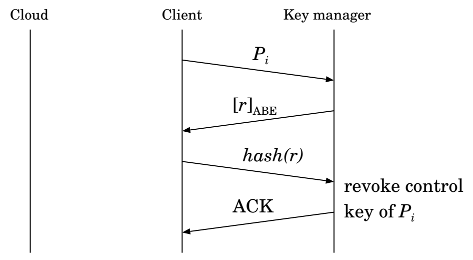


Fig. 5: Policy revocation with ABE.

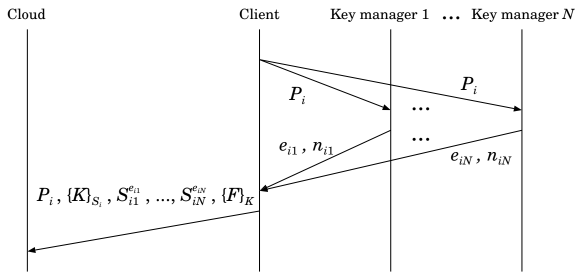

Cloud
Client
Key manager 1 … Key manager N
P_i
P_i
e_{i1}, n_{i1}
e_{iN}, n_{iN}
P_i, $\{K\}_{S_i}$, $S_{i1}^{e_{i1}}$, ..., $S_{iN}^{e_{iN}}$, $\{F\}_K$

Fig. 6: File upload with multiple key managers.

the response to that challenge. Finally, the key manager revokes the policy and acknowledges the client.

4.2.2 Multiple Key Managers

We point out that the use of a single key manager will lead to the single-point-of-failure problem. An untrustworthy key manager may either prematurely removes the keys before the client requests to revoke them, or fail to remove the keys when it is requested to. The former case may prevent the client from getting its data back, while the latter case may subvert assured deletion. Therefore, it is important to improve the robustness of the key management service to minimize its chance of being compromised. Here, we apply Shamir's (M, N) threshold secret sharing scheme [29], where $M \leq N$ (see Section 3.4). Using Shamir's scheme, we divide a secret into N shares and distribute them over N independent key managers, such that we must obtain the correct shares from at least M out of N key managers in order to reconstruct the original secret.

In FADE, we need to address the challenge of how to manage the control keys with $N > 1$ key managers. For each policy P_i, the jth key manager (where $1 \leq j \leq N$) will *independently* generate and maintain an RSA public/private control key pair (e_{ij}, d_{ij}) corresponding to a modulus n_{ij}. We point out that this key pair is independent of the key pairs generated by other key managers, although all such key pairs correspond to the same policy P_i. Also, each key manager keeps its own key pair and will not release it to other key managers.

Let us consider a file F that is associated with policy P_i. We now describe the file/policy operations of FADE using multiple key managers.

File Upload. Figure 6 shows the file upload operation with multiple key managers. Instead of storing $S_i^{e_i}$ on the cloud as in the case of using a single key manager, the client now splits S_i into N shares, $S_{i1}, S_{i2}, \ldots, S_{iN}$

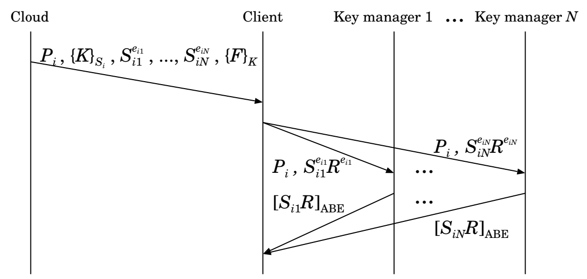

Cloud
Client
Key manager 1 … Key manager N
P_i, $\{K\}_{S_i}$, $S_{i1}^{e_{i1}}$, ..., $S_{iN}^{e_{iN}}$, $\{F\}_K$
P_i, $S_{i1}^{e_{i1}}R^{e_{i1}}$
P_i, $S_{iN}^{e_{iN}}R^{e_{iN}}$
$[S_{i1}R]_{\mathrm{ABE}}$
$[S_{iN}R]_{\mathrm{ABE}}$

Fig. 7: File download with multiple key managers and ABE.

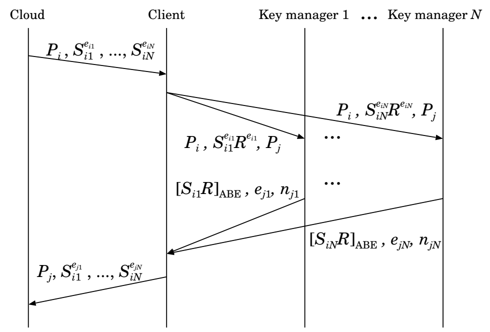

Cloud
Client
Key manager 1 … Key manager N
P_i, $S_{i1}^{e_{i1}}$, ..., $S_{iN}^{e_{iN}}$
P_i, $S_{iN}^{e_{iN}}R^{e_{iN}}$, P_j
P_i, $S_{i1}^{e_{i1}}R^{e_{i1}}$, P_j
$[S_{i1}R]_{\mathrm{ABE}}$, e_{j1}, n_{j1}
$[S_{iN}R]_{\mathrm{ABE}}$, e_{jN}, n_{jN}
P_j, $S_{i1}^{e_{i1}}$, ..., $S_{iN}^{e_{iN}}$

Fig. 8: A special case of policy renewal with multiple key managers and ABE - when policy P_i is renewed to policy P_j.

using Shamir's scheme. Next, the client requests each key manager j for the public control key (n_{ij}, e_{ij}). Then the client computes $S_{ij}^{e_{ij}}$ (mod n_{ij}) for each j, and sends $\{K\}_{S_i}$, $S_{i1}^{e_{i1}}, S_{i2}^{e_{i2}}, \ldots, S_{iN}^{e_{iN}}$, and $\{F\}_K$ to the cloud. Finally, the client discards K, S_i, and $S_{i1}, S_{i2}, \ldots, S_{iN}$.

File Download. Figure 7 shows the file download operation with multiple key managers. After retrieving the encrypted key shares $S_{i1}^{e_{i1}}, S_{i2}^{e_{i2}}, \ldots, S_{iN}^{e_{iN}}$ from the cloud, the client needs to request each key manager to decrypt a share. For the jth share $S_{ij}^{e_{ij}}$ $(j = 1, 2, \ldots, N)$, the client blinds it with a randomly generated number R, and sends $S_{ij}^{e_{ij}} R^{e_{ij}}$ to key manager j. Then, key manager j responds the client with $S_{ij}R$. It also encrypts the response with ABE. After unblinding, the client knows S_{ij}. After collecting M decrypted shares of S_{ij}, the client can combine them into S, and hence decrypts K and F.

Policy Renewal. The policy renewal operation is similar to our original operation discussed in Section 4.1.4. The only difference is that the client needs to renew every share of S_i. Note that in this operation we do not need to combine or split the shares. Figure 8 shows a special case of renewing a policy P_i to another P_j (cf. Figure 4 in Section 4).

Policy Revocation. The client needs to ask every key manager to revoke the policy. As long as at least $(N - M + 1)$ key managers remove the private control keys corresponding to the policy, all files associated with this policy become assuredly deleted.

4.3 Security Analysis

FADE is designed to protect outsourced data from unauthorized access and to assuredly delete outsourced data. We now briefly summarize how FADE achieves its security properties as described in Section 3.4.

In our context, the cloud storage is untrusted and insecure. The cloud may still keep backup copies of any outsourced file after it is requested for deletion. Suppose that an attacker gains access to the cloud storage and obtains the (encrypted) copies of all active and deleted files. Now we argue that the attacker cannot recover any data from those files protected with FADE.

Active files. An active file on the cloud is encrypted with a data key, which can only be decrypted by the key manager. In order to reveal the original data, the attacker has to request the key manager to decrypt the data key. As discussed in Section 4.2.1, the response from the key manager is protected with the ABE-based access key. As long as the attacker does not have the access key, it cannot decrypt the data key, and hence cannot decrypt the original data.

Deleted files. A file becomes deleted when its associated policy is revoked. A deleted file is still encrypted with a data key. However, since the key manager has purged the control key for the revoked policy permanently, it loses the ability to decrypt the data key. Therefore, the attacker cannot recover the original data. Moreover, even if the attacker is powerful enough to get the ABE access key or compromise the key manager to get all control keys, the original data of the deleted file is still unrecoverable as the corresponding control key is already disposed.

5 Implementation

We implement a working prototype of FADE using C++ on Linux. Our implementation is built on off-the-shelf library APIs. Specifically, we use the OpenSSL library [22] for the cryptographic operations, the `cpabe` library [34] for the ABE-based access control, and the `ssss` library [31] for sharing control keys to a quorum of key managers. The `ssss` library is originally designed as a command-line utility to deal with keys in ASCII format. We slightly modify `ssss` and add two functions to split and combine keys in binary format, so as to make it compatible with other libraries. In addition, we use Amazon S3 [4] as our cloud storage backend.

In the following, we define the metadata of FADE being attached to individual data files. We then describe how we implement the client and a quorum of key managers and how the client interacts with the cloud.

5.1 Representation of Metadata

For each data file protected by FADE, we include the metadata that describes the policies associated with the file as well as a set of cryptographic keys. More precisely, the metadata contains the specification of the Boolean combination of policies, and the corresponding cryptographic keys including the encrypted data key of the file and the control keys associated with the policies. Here, we assume that each (atomic) policy is specified by a unique 4-byte integer identifier. To represent a Boolean combination of policies, we express it in *disjunctive canonical form*, i.e., the disjunction (OR) of conjunctive policies, and use the characters '`*`' and '`+`' to denote the AND and OR operators. We upload the metadata as a separate file to the cloud. This enables us to renew policies directly on the metadata file without retrieving the entire data file from the cloud.

In our implementation, individual data files have their own metadata, each specifying its own data key. To reduce the metadata overhead as compared to the data file size, we can form a tarball of multiple files under the same policy combination and have all files protected with the same data key.

5.2 Client

Our client implementation uses four function calls to enable end users to interact with the cloud:

- `Upload(file, policy)`. The client encrypts the input file according to the specified policy (or a Boolean combination of policies). Here, the file is encrypted using the 128-bit AES algorithm with the cipher block chaining (CBC) mode. After encryption, the client also appends the encrypted file size (8 bytes long) and the HMAC-SHA1 signature (20 bytes long) to the end of encrypted file for integrity checking in later downloads. It then sends the encrypted file and the metadata onto the cloud.
- `Download(file)`. The client retrieves the file and policy metadata from the cloud. It then checks the integrity of the encrypted file, and decrypts the file.
- `Revoke(policy)`. The client tells the key managers to permanently revoke the specified policy. All files associated with the policy will be assuredly deleted. If a file is associated with the conjunctive policy combination that contains the revoked policy, then it will be assuredly deleted as well.
- `Renew(file, new_policy)`. The client first fetches the metadata of the given file from the cloud. It updates the metadata with the new policy. Finally, it sends the metadata back to the cloud. Note that the operation does not involve transfer of the input file.

We export the above function calls exported as library APIs. Thus, different implementations of the client can call the library APIs and have the protection offered by FADE. In our current prototype, we implement the client as a user-level program that can access files under a specified folder.

The above interfaces *wrap* the third-party APIs for interacting with the cloud. As an example, we use LibAWS++ [18], a C++ library for interfacing with Amazon S3 using plain HTTP.

5.3 Key Managers

We implement a quorum of key managers, each of which supports two major types of functions: (i) *policy management*, in which a key manager creates or revokes policies, as well as their associated control keys (for assured deletion) and access keys (for access control), and (ii) *key management*, in which a key manager performs the encryption or decryption on the (blinded) data key.

We implement the basic functionalities of a key manager so that it can perform the required operations on the cryptographic keys. In particular, all the policy control keys are built upon 1024-bit blinded RSA (see Section 4.1.1). Besides, each individual key manager supports ABE for access control.

6 Evaluation

We now evaluate the empirical performance of our implemented prototype of FADE atop Amazon S3. It is crucial that FADE does not introduce substantial performance or monetary overhead that will lead to a big increase in data management costs. In addition, the cryptographic operations of FADE should only bring insignificant computational overhead. Therefore, our experiments aim to answer the following questions: What is the performance and monetary overhead of FADE? Is it feasible to use FADE to provide file assured deletion for cloud storage?

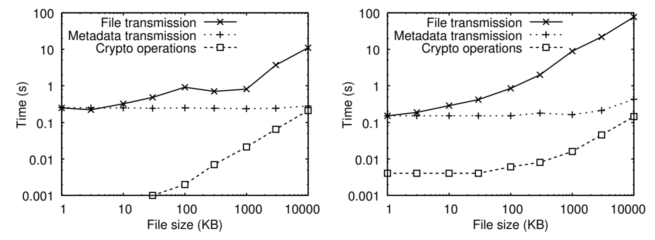
Our experiments use Amazon S3 APAC servers that reside in Singapore for our cloud storage backend. Also, we deploy the client and the key managers within a university department network in Hong Kong. We evaluate FADE on a per-file basis, that is, when it operates on an individual file of different sizes. We can proportionally scale our results for the case of multiple files.

6.1 Time Performance of FADE

We first measure the time performance of our FADE prototype. In order to identify the time overhead of FADE, we divide the running time of each measurement into three components:

- *file transmission time*, the uploading/downloading time for the data file between the client and the cloud.
- *metadata transmission time*, the time for uploading/downloading the metadata, which contains the policy information and the cryptographic keys associated with the file, between the client and the cloud.
- *cryptographic operation time*, the total time for cryptographic operations, which includes the total computational time used for performing AES and HMAC on the file, and the time for the client to coordinate with the quorum of key managers on operating the cryptographic keys.

We average each of our measurement results over 10 different trials.



(a) Upload (b) Download

Fig. 9: Experiment A.1 (Performance of file upload/download operations).

6.1.1 Evaluation of Basic Design

We first evaluate the time performance of the basic design of FADE (see Section 4), in which we use a single key manager and do not involve ABE.

Experiment A.1 (Performance of file upload/download operations). In this experiment, we measure the running time of the file upload and download operations for different file sizes (including 1KB, 3KB, 10KB, 30KB, 100KB, 300KB, 1MB, 3MB, and 10MB).

Figure 9 shows the results. First, the cryptographic operation time increases with the file size, mainly due to the symmetric-key encryption applied to a larger file. Nevertheless, we find that in all cases of file upload/download operations, the time of cryptographic operations is no more than 0.2s (for a file size within 10MB), and accounts for no more than 2.6% of the file transmission time. We expect that FADE only introduces a small time overhead in cryptographic operations as compared to the file transmission time, where the latter is inevitable even without FADE.

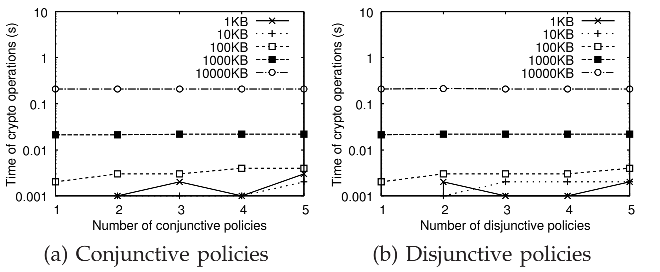
Also, the metadata transmission time is always around 0.2s, regardless of the file size. This is expected, since the metadata file only stores the policy information and cryptographic keys, both of which are independent of the data files. The file transmission time is comparable to the metadata transmission time for small files. However, for files larger than 100KB, the file transmission time becomes the dominant factor. For instance, to upload or download a 10MB file, the sum of the metadata transmission time and the cryptographic operation time (both are due to FADE) account for 4.1% and 0.7% of the total time, respectively.

Note that the upload and download operations are asymmetric and use different times to complete the operations. Nevertheless, the performance overhead of FADE drops when the size of the data file being protected is large enough, for example, on the megabyte scale.

Experiment A.2 (Performance of policy updates). Table 2 shows the time used for renewing a single policy of a file (see Figure 4 in Section 4.1.4), in which we update the policy metadata on the cloud with the new set of cryptographic keys. We conduct the experiment on various file sizes ranging from 1KB to 10MB. Our experiments show that the total time is generally small (about 0.3 seconds) regardless of the file size, since we operate

File size	Total time	Metadata transmission				Crypto operations	
		Download	(%)	Upload	(%)	Time	(%)
1KB	0.294s	0.117s	39.9%	0.173s	58.8%	0.004s	1.3%
10KB	0.268s	0.089s	33.0%	0.176s	65.6%	0.004s	1.3%
100KB	0.259s	0.083s	32.2%	0.171s	66.3%	0.004s	1.5%
1MB	0.252s	0.082s	32.7%	0.166s	65.8%	0.004s	1.6%
10MB	0.275s	0.106s	38.5%	0.165s	60.2%	0.004s	1.3%

TABLE 2: Experiment A.2 (Performance of policy updates).



(a) Conjunctive policies (b) Disjunctive policies

Fig. 10: Experiment A.3 (Performance of multiple policies).

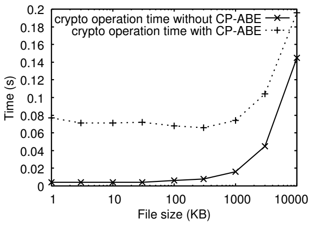
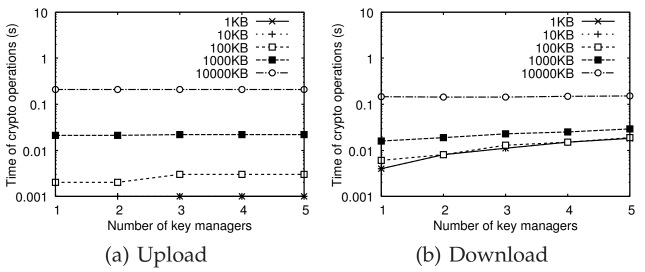


Fig. 11: Experiment B.1 (Performance of CP-ABE).



(a) Upload (b) Download

Fig. 12: Experiment B.2 (Performance of multiple key managers).

on the policy metadata only. Also, the cryptographic operation time only takes about 0.004s in renewing a policy, and this value is independent of the file size.

Experiment A.3 (Performance of multiple policies). We now evaluate the performance of FADE when multiple policies are associated with a file (see Section 4.1.3). Here, we focus on the file upload operation, as we have similar observation for the file download operation. We look at two specific combinations of policies, one on the conjunctive case and one on the disjunctive case.

Figure 10a shows the cryptographic operations time for different numbers of conjunctive policies, and Figure 10b shows the case for disjunctive policies. A key observation is that for each file size, the cryptographic operation time is more or less constant (less than 0.22s) within five policies. It is reasonable to argue that the time will increase when a file is associated with a significantly large number of policies. On the other hand, we expect that in practical applications, a file is associated with only a few policies, and the overhead of cryptographic operations is still minimal.

6.1.2 Evaluation of Extensions

We now evaluate the time performance of the extensions that we add to FADE (see Section 4.2). This includes the use of ABE and a quorum of key managers.

Experiment B.1 (Performance of CP-ABE). In the file download operations, the key manager encrypts the decrypted keys with the ABE-based key of the corresponding policy (or combination of policies) (see Section 4.2). In this experiment, we examine the overhead of this additional encryption. We focus on downloading a file that is associated with a single policy, assuming that a single key manager is used.

Figure 11 shows the cryptographic operation time for downloading a file with CP-ABE and without CP-ABE. We find that CP-ABE introduces a constant overhead of 0.06-0.07 seconds, which is reasonable. This shows the trade-off between better performance and better security.

Experiment B.2 (Performance of multiple key managers). We now analyze the performance of using multiple key managers. Here, we do not enforce access control with ABE, in order to focus on the overhead introduced by multiple key managers. In particular, we use the N-out-of-N scheme for key sharing, i.e., the client needs to retrieve key shares from *all* key managers. This puts the maximum load on the key managers.

Figure 12 shows the cryptographic operation time using different number of key managers. For the file upload operation, the cryptographic operation time stays nearly constant (less than 0.22s) when the number of key managers increases. For the file download operation, the cryptographic operation time only increases by about 0.01s when the number of key managers increases from one to five. Again, this value is less significant for uploading/downloading larger data files.

Experiment B.3 (Combining everything together). Lastly, we combine multiple policies, CP-ABE, and multiple key managers together. The enables us to understand the maximum load of FADE with all the available security protection schemes. In this experiment, we measure the time overhead when downloading a 10MB file with different number of policies and key managers. We

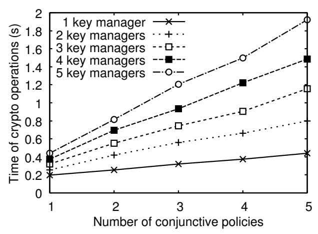


Fig. 13: Experiment B.3 (Performance of multiple policies and multiple key managers with CP-ABE).

consider the case where all policies are conjunctive. For the multiple key managers, we use the N-out-of-N key sharing scheme.

Figure 13 shows the cryptographic operation time for each case. We find that when turning on CP-ABE, the time of cryptographic operations increases almost linearly with both the number of policies and the number of key managers. Even for the worst case (five policies and five key managers), the cryptographic operation time is still less than two seconds, which is small compared with the file transmission time.

6.2 Space Utilization of FADE

We now assess the space utilization. As stated in Section 5.1, each data file is accompanied with its file size (8 bytes), the HMAC-SHA1 signature (20 byte), and a metadata file that stores the policy information and cryptographic keys. For the metadata file, its size differs with the number of policies and the number of key managers used. Here, we analyze the space overhead due to the metadata introduced by FADE.

Table 3 and Table 4 show the different sizes of the metadata based on our implementation prototype for a variable number of (a) conjunctive policies ($P_1 \wedge P_2 \wedge \cdots \wedge P_m$), and (b) disjunctive policies ($P_1 \vee P_2 \vee \cdots \vee P_m$). To understand how each metadata size is obtained, we consider the simplest case where there is only a single policy and a single key manager. Then we need: (i) 128 bytes for each share of the policy-based secret key $S_i^{e_i}$ for policy i, (ii) 16 bytes for the encrypted copy of K based on 128-bit AES, (iii) 4 bytes for the policy identifier, and (iv) 1 byte for the delimiter between the policy identifier and the keys. In this case, the metadata size is 149 bytes. Note that in the case of multiple policies, we need to store more policy identifiers as well as more cryptographic keys, and hence the metadata size increases. Also, the metadata size increases with the number of key managers (see Section 4.2.2). This space overhead becomes less significant if the file size is large enough (e.g., on the megabyte scale).

6.3 Cost Model

We now evaluate the monetary overhead of FADE using a simple pricing model. Here, we use a simplified pricing scheme of Amazon S3 in Singapore, in which we assume that our storage usage is less than 1TB

	Num. of key managers				
Num. of policies	1	2	3	4	5
1	149	277	405	533	661
2	282	538	794	1050	1306
3	415	799	1183	1567	1951
4	548	1060	1572	2084	2596
5	681	1321	1961	2601	3241

TABLE 3: Size of the policy metadata for conjunctive policies (in bytes).

	Num. of key managers				
Num. of policies	1	2	3	4	5
1	149	277	405	533	661
2	298	554	810	1066	1322
3	447	831	1215	1599	1983
4	596	1108	1620	2132	2644
5	745	1385	2025	2665	3305

TABLE 4: Size of the policy metadata for disjunctive policies (in bytes).

and our monthly data outbound transfer size is less than 10TB. We estimate the cost of FADE based on Cumulus [35], a snapshot-based backup system. In [1], it is shown that a typical compressed snapshot consists of hundreds of segments, each of which is around five megabytes. Here, we assume that our data source has s files (segments) and each file is f bytes. Suppose that each segment is associated with p policies[4], and there are N key managers. We evaluate the cost when each file is uploaded u times and downloaded d times. We denote by $meta(p, N)$ the size of the metadata, which is a function of p (number of policies) and N (number of key managers).

Table 5 shows our simplified pricing scheme (as of July 2011) and the corresponding cost results. To illustrate, we plug in some example values as follows. We let $s = 300$ and $f =$ 5MB, for a total of 1.5GB data. We use 3 conjunctive policies and 3 key managers. We assume that each file is uploaded once and downloaded once. From the table, we can see that the extra cost that FADE incurs is less than 1.3% per month.

6.4 Lessons Learned

In this section, we evaluate the performance of FADE in terms of the overheads of time, space utilization, and monetary cost. It is important to note that the performance results depend on the deployment environment. For instance, if the client and the key manager all reside in the same region as Amazon S3, then the transmission times for files and metadata will significantly reduce; or if the metadata contains more descriptive information, the overhead will increase. Nevertheless, we emphasize that our experiments can show the feasibility of FADE in

4. In Cumulus, each segment may be composed of multiple small files. We assume the simplest case that all the files are associated with the same combination of policies.

	Pricing	Without FADE	With FADE
Storage (c_s)	\$0.14 per GB	$c_s \cdot s \cdot f = \$0.210$	$c_s \cdot s \cdot (f + 28 + meta(p, N)) = \$0.210$
Data transfer in (c_i)	\$0.00 per GB	$c_i \cdot s \cdot f \cdot u = \$0.000$	$c_i \cdot s \cdot (f + 28 + meta(p, N)) \cdot u = \$0.000$
Data transfer out (c_o)	\$0.19 per GB with first 1GB free	$c_o \cdot s \cdot f \cdot d = \$0.095$	$c_o \cdot s \cdot (f + 28 + meta(p, N)) \cdot d = \$0.095$
PUT requests (c_p)	\$0.01 per 1,000 requests	$c_p \cdot s \cdot u = \$0.003$	$c_p \cdot s \cdot 2u = \$0.006$
GET requests (c_g)	\$0.01 per 10,000 requests	$c_g \cdot s \cdot d = \$0.000$	$c_g \cdot s \cdot 2d = \$0.001$
Total cost		\$0.308	\$0.312

TABLE 5: A simplified pricing scheme of Amazon S3 in Singapore and the corresponding cost report (in US dollars). All numbers are rounded off to 3 decimal places. Note that FADE only adds very small overheads to the storage and data transfer costs, which are rounded off to the same values as in without FADE.

providing an additional level of security protection for today's cloud storage.

We note that the performance overhead of FADE becomes less significant when the size of the actual data file content increases (e.g. on the order of megabytes or even bigger). Thus, FADE is more suitable for enterprises that need to archive large files with a substantial amount of data. On the other hand, individuals may generally manipulate small files on the order of kilobytes. In this case, we may consider associating the same metadata with a tarball of multiple files (see Section 5) to reduce the overhead of FADE.

7 Related Work

In this section, we review other related work on how to apply security protection to outsourced data storage.

Cryptographic protection on outsourced storage. Recent studies (see survey in [17]) propose to protect outsourced storage via cryptographic techniques. Plutus [16] is a cryptographic storage system that allows secure file sharing over untrusted file servers. Ateniese *et al.* [6] and Wang *et al.* [36] propose an auditing system that verifies the integrity of outsourced data. Wang *et al.* [37] propose a secure outsourced data access mechanism that supports changes in user access rights and outsourced data. However, all the above systems require new protocol support on the cloud infrastructure, and such additional functionalities may make deployment more challenging.

Secure storage solutions for public clouds. Secure solutions that are compatible with existing public cloud storage services have been proposed. Yun *et al.* [40] propose a cryptographic file system that provides privacy and integrity guarantees for outsourced data using a universal-hash based MAC tree. They prototype a system that can interact with an untrusted storage server via a modified file system. JungleDisk [15] and Cumulus [35] protect the privacy of outsourced data, and their implementation use Amazon S3 [4] as the storage backend. Specifically, Cumulus focuses on making effective use of storage space while providing essential encryption on outsourced data. On the other hand, such systems do not consider file assured deletion in their designs.

Access control. One approach to apply access control to outsourced data is by *attribute-based encryption (ABE)*, which associates fine-grained attributes with data. ABE is first introduced in [27], in which attributes are associated with encrypted data. Goyal *et al.* [13] extend the idea to key-policy ABE, in which attributes are associated with private keys, and encrypted data can be decrypted only when a threshold of attributes are satisfied. Pirretti *et al.* [25] implement ABE and conduct empirical studies, and also point out th. Nair *et al.* [20] consider a similar idea of ABE, and they seek to enforce a fine-grained access control of files based on identity-based public key cryptography.

Policy-based deletion follows the similar notion of ABE, in which data can be accessed only if the corresponding attributes (i.e., atomic policies in our case) are satisfied. However, policy-based deletion has a different design objective from ABE. Policy-based deletion focuses on how to *delete* data, while ABE focuses on how to *access* data based on attributes. A major feature of ABE is to issue users the decryption keys of the associated attributes so that they can access files that satisfy the attributes, and hence existing studies of ABE seek to ensure that no two users can collude if they are tied with different sets of attributes.

On the other hand, in policy-based deletion, since each policy is possessed by multiple users, revoking a policy requires a centralized administrator to manage the revocation [25]. Boldyreva *et al.* [8] and Yu *et al.* [39] combine ABE with attribute revocation, and both of the studies require the use of some centralized key server to manage the attributes and the corresponding keys (i.e., policy-based control keys in our case). For example, in [39], there is a semi-trustable on-line proxy server, in which data is re-encrypted with new keys upon attribute revocation.

In FADE, each policy is associated with two keys. One is the access key, which is issued to users, and another is the control key, which is maintained by the key server. Both access and control keys are required to decrypt a file. This type of separation is similar to the approaches in [8], [39]. On the other hand, the main focus of our work is to empirically evaluate the feasibility of our system via practical implementation, while [8], [39] mainly focus on security analysis.

Assured deletion. In Section 2.1, we discuss time-based deletion in [12], [23], which we generalize into policy-based deletion. There are several related systems on assured deletion (which come after our conference version of the paper [33]). Keypad [11] protects data in theft-prone devices (e.g., laptops, USB sticks) by encrypting such data and maintaining keys in an independent, centralized key server, similar to FADE. It removes all data of a protected device upon requests of deletion, and does not consider fine-grained deletion as in FADE. Nasuni announced the support of assured deletion in backup snapshots in March 2011 [21]. However, there is no formal study about their implementation methodologies and performance evaluation. In our recent work [26], we extend the idea of assured deletion to cloud backup systems with version control, but the work [26] does not consider access control and the use of multiple key managers for key management.

8 Conclusions

We propose a practical cloud storage system called FADE, which aims to provide access control assured deletion for files that are hosted by today's cloud storage services. We associate files with file access policies that control how files can be accessed. We then present policy-based file assured deletion, in which files are assuredly deleted and made unrecoverable by anyone when their associated file access policies are revoked. We describe the essential operations on cryptographic keys so as to achieve access control and assured deletion. FADE also leverages existing cryptographic techniques, including attribute-based encryption (ABE) and a quorum of key managers based on threshold secret sharing. We implement a prototype of FADE to demonstrate its practicality, and empirically study its performance overhead when it works with Amazon S3. Our experimental results provide insights into the performance-security trade-off when FADE is deployed in practice.

Source code of FADE (including the new features of this journal paper) is available at **http://ansrlab.cse.cuhk.edu.hk/software/fade**.

References

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

[2] Amazon. Case Studies. http://aws.amazon.com/solutions/case-studies/#backup.

[3] Amazon. SmugMug Case Study: Amazon Web Services. http://aws.amazon.com/solutions/case-studies/smugmug/, 2006.

[4] Amazon S3. http://aws.amazon.com/s3, 2010.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Comm. of the ACM*, 53(4):50–58, Apr 2010.

[6] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and Efficient Provable Data Possession. In *Proc. of SecureComm*, 2008.

[7] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-Policy Attribute-Based Encryption. In *Proc. of IEEE Symp. on Security and Privacy*, May 2006.

[8] A. Boldyreva, V. Goyal, and V. Kumar. Identity-based Encryption with Efficient Revocation. In *Proc. of ACM CCS*, 2008.

[9] T. Dierks and E. Rescorla. The transport layer security (tls) protocol version 1.2, Aug 2008. RFC 5246.

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

[11] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: Auditing File System for Mobile Devices. In *Proc. of EuroSys*, April 2011.

[12] R. Geambasu, T. Kohno, A. Levy, and H. M. Levy. Vanish: Increasing Data Privacy with Self-Destructing Data. In *Proc. of USENIX Security Symp.*, Aug 2009.

[13] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data. In *Proc. of ACM CCS*, 2006.

[14] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proc. of USENIX Security Symp.*, 1996.

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

[16] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proc. of USENIX FAST*, 2003.

[17] S. Kamara and K. Lauter. Cryptographic Cloud Storage. In *Proc. of Financial Cryptography: Workshop on Real-Life Cryptographic Protocols and Standardization*, 2010.

[18] LibAWS++. http://aws.28msec.com/, 2010.

[19] A. J. Menezes, P. C. van Oorschot, and S. A.Vanstone. *Handbook of Applied Cryptography*. CRC Press, Oct 1996.

[20] S. Nair, M. T. Dashti, B. Crispo, and A. S. Tanenbaum. A Hybrid PKI-IBC Based Ephemerizer System. *IFIP International Federation for Information Processing*, 232:241–252, 2007.

[21] 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/.

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

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

[24] R. Perlman, C. Kaufman, and R. Perlner. Privacy-Preserving DRM. In *IDtrust*, 2010.

[25] M. Pirretti, P. Traynor, P. McDaniel, and B. Waters. Secure Attribute-Based Systems. In *Proc. of ACM CCS*, 2006.

[26] A. Rahumed, H. C. H. Chen, Y. Tang, P. P. C. Lee, and J. C. S. Lui. A Secure Cloud Backup System with Assured Deletion and Version Control. In *3rd International Workshop on Security in Cloud Computing*, 2011.

[27] A. Sahai and B. Waters. Fuzzy Identity-Based Encryption. In *EUROCRYPT*, 2005.

[28] B. Schneier. File Deletion. http://www.schneier.com/blog/archives/2009/09/file_deletion.html, Sep 2009.

[29] A. Shamir. How to Share a Secret. *CACM*, 22(11):612–613, Nov 1979.

[30] SmugMug. http://www.smugmug.com/, 2010.

[31] ssss. http://point-at-infinity.org/ssss/, 2006.

[32] W. Stallings. *Cryptography and Network Security*. Prentice Hall, 2006.

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

[34] The CPABE Toolkit. http://acsc.cs.utexas.edu/cpabe/, 2010.

[35] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem backup to the cloud. *ACM Trans. on Storage*, 5(4), Dec 2009.

[36] C. Wang, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for storage security in cloud computing. In *Proc. of IEEE INFOCOM*, Mar 2010.

[37] W. Wang, Z. Li, R. Owens, and B. Bhargava. Secure and Efficient Access to Outsourced Data. In *ACM CCSW*, Nov 2009.

[38] S. Wolchok, O. S. Hofmann, N. Heninger, E. W. Felten, J. A. Halderman, C. J. Rossbach, B. Waters, and E. Witchel. Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs. In *Proc. of NDSS*, 2010.

[39] S. Yu, C. Wang, K. Ren, and W. Lou. Attribute Based Data Sharing with Attribute Revocation. In *Proc. of ACM ASIACCS*, Apr 2010.

[40] A. Yun, C. Shi, and Y. Kim. On Protecting Integrity and Confidentiality of Cryptographic File System for Outsourced Storage. In *ACM CCSW*, Nov 2009.