International Journal of scientific research and management (IJSRM)
\||Volume||3||Issue||5||Pages|| 2893-2899||2015||
Website: www.ijsrm.in ISSN (e): 2321-3418

Secure Overlay Cloud Storage with File Assured Deletion

Dayananda RB[1] , Prof. Dr. G.Manoj Someswar[2] ,T.P. Suryachandra Rao[3]

1. Associate Professor, Department of CSE, RRIT, Bangalore–90, Karnataka, India
2. Principal & Professor, Department of CSE, Anwar-ul-uloom College of Engineering & Technology (Affiliated to JNTU, Hyderabad), Vikarabad – 501101, RR District, Telangana, India
3. Principal, IMS PG Centre, Uppariguda (V), Ibrahimpatnam Mandal, R. R. District, Telangana, India

Abstract: In our research paper, we present *policy-based file assured deletion,* the major design building block of our FADE architecture. Our main focus is to deal with the cryptographic key operations that enable file assured deletion. We first review time-based file assured deletion. We then explain how it can be extended to policy-based file assured deletion.

Keywords: Policy-Based File Assured Deletion, Time-Based File Assured Deletion, Data Key, Data Owner, Key Manager, Conjunctive Policies, Disjunctive Policies, File Metadata, Policy Metadata

INTRODUCTION

Time-based file assured deletion, which is first introduced in, means that files can be securely deleted and remain permanently inaccessible after a predefined duration. The main idea is that a file is encrypted with a *data key,* and this data key is further encrypted with a *control key* that is maintained by a separate key manager service (known as *Ephemerizer*). In, 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.[1]

Time-based file assured deletion is later prototyped in Vanish. Vanish divides a data key into multiple key shares, which are then stored in different nodes of a peer-to-peer network. Nodes remove the key shares that reside in their caches for 8 hours. If a file needs to remain accessible after 8 hours, then the file owner needs to update the key shares in node caches.

However, both and target only the assured deletion upon time expiration, and do not consider a more fine-grained control of assured deletion with respect to different file access policies.

Policy-based Deletion

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. Similar to time-based deletion, the file content is encrypted with a data key, and the data key is further encrypted with the control keys corresponding to the policy combination. When a policy is revoked, the corresponding control key will be removed from the key manager.[2] Thus, when the policy combination 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 keys of the policies. In this case, we say the file is deleted. The main idea of policy-based deletion is to delete files that are associated with revoked policies.

The definitions of policies vary depending on applications. Time-based deletion is a special case under our framework, and 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.[3]

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,..., 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.[4]

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.

Policy-based deletion follows the similar notion of *attribute-based encryption (ABE)* in which data can be accessed only if a subset of attributes (policies) are satisfied. [5]However, our work is different from ABE in two aspects. First, we focus on how to *delete* data, while ABE focuses on how to *access* data based on attributes. Second, because of the different design objectives, ABE gives users the decryption keys of the associated attributes, so that they can access files that satisfy the attributes. On the other hand, in policy-based deletion, we do *not* share with users any decryption keys of policies, which instead are all maintained in the key manager. Our focus is to appropriately remove keys in the key manager so as to guarantee file assured deletion, which is an important security property when we outsource data storage to the cloud. This guides us into a different design space in contrast with existing ABE approaches.

DESIGN AND IMPLEMENTATION

Our system is composed of three participants: the *data owner,* the *key manager,* and the *storage cloud.* They are described as follows.

Data owner. The data owner is the entity that originates file data to be stored on the cloud. It may be a file system of a PC, a user-level program, a mobile device, or even in the form of a plug-in of a client application.

Key manager. The key manager maintains the policy-based control keys that are used to encrypt data keys. It responds to the data owner's requests by performing encryption, decryption, renewal, and revocation to the control keys.

Storage cloud. The storage cloud is maintained by a third-party cloud provider (e.g., Amazon S3) and keeps the data on behalf of the data owner. We emphasize that we do *not* require any protocol and implementation changes on the storage cloud to support our system. Even a naive storage service that merely provides file upload/download operations will be suitable.

Threat Models and Assumptions

Our main design goal is to provide assured deletion of files produced by the data owner. A file is deleted (or permanently inaccessible) if its policy is revoked and becomes obsolete. Here, we assume that the control key associated with a revoked policy is reliably removed by the key manager.[6] Thus, by assured deletion of files, we mean that any existing file copy that are associated with revoked policies will remain permanently encrypted and unrecoverable.

The key manager can be deployed as a minimally trusted third-party service. By minimally trusted, we mean that the key manager reliably removes the control keys of revoked policies. However, it is possible that the key manager can be compromised. In this case, an attacker can recover the files that are associated with existing active policies. On the other hand, files that are associated with revoked policies still remain inaccessible, as the control keys are removed. Hence, file assured deletion is achieved.

It is still important to improve the robustness of the key manager service to minimize its chance of being compromised. To achieve this, we can use a quorum of key managers, in which we create n key shares for a key, such that any $k < n$ of the key shares can be used to recover the key. While the quorum scheme increases the storage overhead of keys, this is justified as keys are of much smaller size than data files.

Before accessing the active keys in the key manager, the data owner 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. We assume that the data owner does not disclose any successfully decrypted file to unauthorized parties.

The Basics - File Upload/Download

We now introduce the basics of uploading/downloading files to/from the cloud storage. We first assume that each file is associated with a single policy, and then explain how a file is associated with multiple policies.

Our design is based on *blinded RSA* in which the data owner 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 data owner can then recover the data key. In this way, 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 data owner and the key manager.

We first summarize the major notation used throughout the paper. For each policy *i*, the key manager generates two secret large RSA prime numbers *pi* and *qi* and computes the product $n_i = piqi$[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.[7] On the other hand, when the data owner encrypts a file *F*, it randomly generates a data key *K*, and a secret key *Si* that corresponds to policy Pi. We let *{m}k* denote a message m encrypted with key *k* using symmetric-key encryption (e.g., AES). 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 metadata, will be discussed in Section 3. 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 1 shows the file upload operation. The data owner first requests the public key (n_i, e_i) of policy Pi from the key manager, and caches (n_i, e_i) for subsequent uses if the same policy Pi is associated with other files. Then the data owner generates two random keys *K* and S_i, and sends $\{K\}S_i$, 5?% and $\{F\}_K$ to the cloud[2]. Then the data owner can discard *K* and S_i.

File download. Figure 2 shows the file download operation. The data owner fetches $\{K\}_{S_i}$, $S_i^{e_i}$, and $\{F\}_K$ from the storage cloud. Then the data owner generates a secret random number *R*, computes R^{ei}, and sends $S_i^{e_i} \cdot R^{e_i} = (S_i R)^{e_i}$ to the key manager to request for decryption. The key manager then computes and returns $((S_i R)^{e_i})^{d_i} = S_i R$ to the data owner. The data owner can now remove *R* and obtain *Si*, and decrypt $\{K\}_{Si}$ and hence $\{F\}_K$.

Integrity. To protect the integrity of a file, the data owner needs to compute an HMAC on every encrypted file and stores the HMAC, together with the encrypted file, in the cloud storage. When a file is downloaded, the data owner will check whether the HMAC is valid before decrypting the file. We assume that the data owner has a long-term private secret for the HMAC computation.

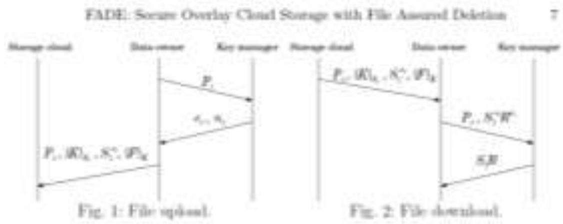

Fig. 1: File upload. Fig. 2: File download.

Policy Revocation for File Assured Deletion

If a policy *Pi* is revoked, then the key manager completely removes the private key *di* and the secret prime numbers *pi* and qi. Thus, we cannot recover *Si* from $S_i^{e_i}$, and hence cannot recover *K* and the file *F*. We say that the file *F*, which is tied to policy P_i, is assuredly deleted. Note that the policy revocation operations do not involve interactions with the storage cloud.[8]

Multiple Policies

In addition to one policy per file, 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.

- **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 storage cloud, the data owner first randomly generates a data key *K*, and secret keys S_1, S_2,...,S_m. It then sends the following to the storage cloud: $\{\{K\}_{S_1}\}_{S_2 \cdots S_m}, S_1^{e_1}, S_2^{e_2}, \ldots, S_m^{e_m}$, and $\{F\}_K$. On the other hand, to recover *F*, the data owner generates a random number *R* and sends $(S_1R)^{e_1}, (S_2R)^{e_2}, \ldots, (S_mR)^{e_m}$ to the key manager, which then returns S_1R, S_2R,..., S_mR. The data owner can then recover S_1, S_2,..., 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 data owner 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 $\{F\}_K$, Smem, and $\{F\}_K$. Therefore, the data owner 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.[9]

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.[10]

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. However, to guarantee file assured deletion, policy renewal can be performed only when the following condition holds: *the old policy will always be revoked first before the new policy is revoked.* The reason is that after policy renewal, there will be two versions of a file: one is protected with the old policy, and one is protected with the new policy. If the new policy is revoked first, then the file version that is protected with the old policy may still be accessible when the control keys of the old policy are compromised, meaning that the file is not assuredly deleted.

It is important to note that it is a non-trivial task to enforce the condition of policy renewal, as the old policy may be associated with other existing files. In this research paper, we do not consider this issue and we pose it as future work.

Suppose that we have enforced the condition of policy renewal. A straightforward approach of implementing policy renewal is to combine the file upload and download operations, but without retrieving the encrypted file from the cloud.[11] The procedures can be summarized as follows: (i) download all encrypted keys from the storage 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 policies, and finally (v) send the newly encrypted keys back to the cloud.

In some special cases, optimization can be made so as to save the operations of decrypting and re-encrypting the data key. Suppose that the Boolean combination structure of policies remain unchanged, but one of the atomic policies P_i is changed $P_{i'}$. 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. In this case, the data owner simply sends the blinded version $S_i^{e_i} R^{e_i}$ to the key manager, which then returns S_iR. The data owner then recovers Si. Now, the data owner re-encrypts S_i into $S_i^{e_{i'}}$ (mod $n_{i'}$), where $(n_{i'}, e_{i'})$ is the public key of policy $P_{i'}$, and sends it to the cloud. Note that the encrypted data key K remains intact. Figure 3 illustrates this special case of policy renewal.

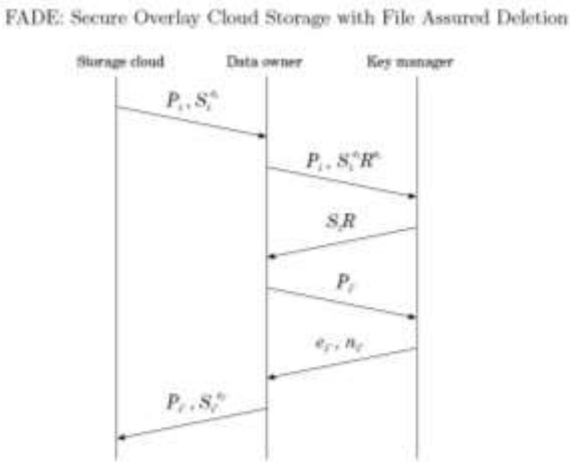

Figure 3: Policy renewal

THE FADE ARCHITECTURE

We implement a working prototype of FADE using C++ on Linux, and we use the OpenSSL library for the cryptographic operations. In addition, we use Amazon S3 as our storage cloud. This section is to address the implementation issues of our FADE architecture, based on our experience in prototyping FADE. Our goal is to show the practicality of FADE when it is deployed with today's cloud storage services.

Figure 4 shows the FADE architecture. In the following, we define the metadata of FADE attached to individual files. We then describe how we implement the data owner and the key manager, and how the data owner interacts with the storage cloud.

Representation of Metadata

For each file protected by FADE, we include the metadata that describes the policies associated with the file as well as a set of encrypted keys. In FADE, there are two types of metadata: *file metadata* and *policy metadata.*

File metadata. The file metadata mainly contains two pieces of information: file size and HMAC. We hash the encrypted file with HMAC-SHA1 for integrity checking. The file metadata is of fixed size (with 8 bytes of file size and 20 bytes of HMAC) and attached at the beginning of the encrypted file. Both the file metadata and the encrypted data file will then be treated as a single file to be uploaded to the storage cloud.

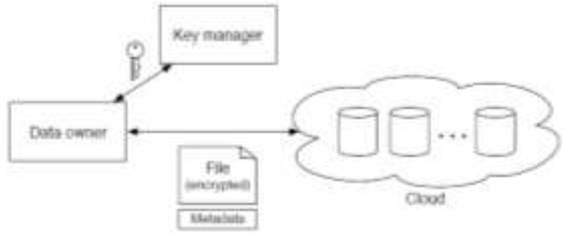

Figure 4: The FADE architecture

Policy metadata. The policy metadata includes the specification of the Boolean combination of policies and the corresponding encrypted cryptographic keys. Here, we assume that each single 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. Then we upload the policy metadata as a separate file to the storage cloud. This enables us to renew policies directly on the policy metadata without retrieving the entire file from the storage cloud.

In our implementation, individual files have their own policy metadata, although we allow multiple files to be associated with the same policy (which is the expected behavior of FADE). In other words, for two data files that are under the same policy, they will have different policy metadata files that specify different data keys, and the data keys are protected by the control key of the same policy. In this research paper, we discuss how we may associate the same policy metadata file with multiple data files so as to reduce the metadata overhead.

Data Owner and Storage Cloud

Our implementation of the data owner uses the following four function calls to enable end users to interact with the storage cloud:

- **Upload (file, policy).** The data owner encrypts the input file using the specified policy (or a Boolean combination of policies). It then sends the encrypted file and the metadata onto the cloud. [12] In our implementation, the file is encrypted using the 128-bit AES algorithm with the cipher block chaining (CBC) mode, yet we can adopt a different symmetric-key encryption algorithm depending on applications.

- **Download(file).** The data owner retrieves the file and the policy metadata from the cloud, checks the integrity of the file, and decrypts the file.

- **Delete (policy).** The data owner tells the key manager to permanently revoke the specified policy. All files associated with the policy will be assuredly deleted.

- **Renew (file, new policy).** The data owner first fetches the policy metadata for the given file from the cloud. It then updates the policy metadata with the new policy. Finally, it sends the policy metadata back to the cloud.

The above function calls can be exported as library APIs that can be embedded into different implementations of the data owner. In our current prototype, we implement the data owner as a user-level program that can access files under a working directory of a desktop PC.

The above exported interfaces *wrap* the third-party APIs for interacting with the storage cloud. As an example, we use LibAWS++, a C++ library for interfacing with Amazon S3. We note that the LibAWS++ library uses HTTP to communicate with the cloud, and it does not provide any security protection on the data being transferred. To interact with different cloud storage services, we can use different third-party APIs, provided that the APIs support the basic file upload/download operations.

Key Manager

We implement the key manager that supports the following four basic functions.

- *Creating a policy.* The key manager creates a new policy and returns the corresponding public control key.

- *Retrieving the public control key of a policy.* If the policy is accessible, then the key manager returns the public control key. Otherwise, it returns an error.[13]

- *Decrypting a key with respect to a policy.* If the policy is accessible, then the key manager decrypts the (blinded) key. Otherwise, it returns an error.

- *Revoking a policy.* The key manager revokes the policy and removes the corresponding keys.

We implement the basic functionalities of the 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. To make the key manager more robust, we can extend the key manager to a quorum of key managers as stated in and implement a PKI-based certification system for policy checking.

Evaluation

We implement a prototype of FADE atop Amazon S3, and we now evaluate the empirical performance of FADE. It is crucial that FADE does not introduce substantial performance 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 issue: *What is the performance overhead of FADE, and is it feasible to use FADE to provide file assured deletion for cloud storage?*

Our experiments use Amazon S3, residing in the United States, as the storage cloud. Also, we deploy the data owner and the key manager within an organization's network that resides in an Asian country. In the experiments, we evaluate FADE when it operates on an individual file of different sizes: 1KB, 10KB, 100KB, 1MB, and 10MB.

EXPERIMENTAL RESULTS & CONCLUSION ON TIME PERFORMANCE OF FADE

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

- *data transmission time*, the data uploading/

downloading time between the data owner and the storage cloud. We further divide it into two components: the *file* component, which measures the transmission time for the file body and the file metadata, and the *policy* component, which measures the transmission time for the policy metadata. We upload/download these two components as two separate copies to/from the storage cloud.

- *AES and HMAC time,* the total computational time used for performing AES and HMAC on the file.

- *Key management time,* the time for the data owner to coordinate with the key manager on operating the cryptographic keys. For the file upload operation, we require the data owner to obtain the public control key for the corresponding policy; for the download operation, the data owner works with the key manager to obtain the data key.

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

Experiment 1 (Performance of file upload/download operations)

First, we measure the running time of the file upload and download operations for different file sizes. Table 1 shows the results. We find that the transmission time is the dominant factor (over 99%). The AES and HMAC time increases linearly with the file size. However, the key management time stays constant on the order of milliseconds, regardless of the file size. In other words, compared with the basic encryption and integrity check provided by AES and HMAC, FADE only involves a small time overhead in key management.

We note that when the file size is small, the transmission time for the policy metadata is comparable with the transmission time for the file. To understand this, we capture and analyze the data traffic, and find that the round-trip time between our network (in Asia) and Amazon S3 (in the United States) is 200-300 milliseconds. Because the file and the policy metadata are stored on the cloud as two separate copies, they are transferred through two different TCP connections, and a significant portion of data transmission time is actually due to the TCP connection setup. In this research paper, we will show that the actual number of bytes stored for the policy metadata is in fact much less than that for the file.

Experiment 2 (Performance of policy updates). Table 2 shows the time used for renewing a single policy of a file in which we update the policy metadata on the storage cloud with the new set of cryptographic keys. Our experiments show that the total time is generally small (less than a second) regardless of the file size, as we operate on the policy metadata only. Also, the key management time only takes about 0.004s in renewing a policy, and this value is again independent of the file size.

FADE: Secure Overlay Cloud Storage with File Assured Deletion 13

File size	Total time	Data transmission				AES+HMAC		Key management	
		File	(%)	Policy	(%)	Time	(%)	Time	(%)
1KB	1.260s	0.724s	57.4%	0.537s	42.6%	0.000s	0.0%	0.000s	0.0%
10KB	1.552s	1.020s	65.7%	0.532s	34.3%	0.001s	0.0%	0.000s	0.0%
100KB	2.452s	1.903s	77.6%	0.546s	22.3%	0.002s	0.1%	0.001s	0.0%
1MB	4.194s	3.646s	86.9%	0.527s	12.6%	0.022s	0.5%	0.000s	0.0%
10MB	16.275s	15.463s	95.0%	0.595s	3.7%	0.218s	1.3%	0.000s	0.0%

(a) Upload

File size	Total time	Data transmission				AES+HMAC		Key management	
		File	(%)	Policy	(%)	Time	(%)	Time	(%)
1KB	0.843s	0.485s	57.5%	0.355s	42.1%	0.000s	0.0%	0.003s	0.4%
10KB	0.912s	0.615s	67.4%	0.294s	32.2%	0.000s	0.0%	0.003s	0.3%
100KB	1.968s	1.682s	85.5%	0.282s	14.3%	0.002s	0.1%	0.002s	0.1%
1MB	4.696s	4.360s	92.8%	0.317s	6.7%	0.017s	0.4%	0.002s	0.1%
10MB	33.746s	33.182s	98.3%	0.395s	1.2%	0.166s	0.5%	0.002s	0.0%

(b) Download

Table 1: Experiment 1 (Performance of upload/download operations)

File size	Total time	Data transmission				Key management	
		Download	(%)	Upload	(%)	Time	(%)
1KB	0.923s	0.315s	34.1%	0.605s	65.5%	0.004s	0.4%
10KB	0.805s	0.266s	33.0%	0.536s	66.6%	0.004s	0.4%
100KB	0.821s	0.271s	33.0%	0.546s	66.5%	0.004s	0.5%
1MB	0.813s	0.273s	33.5%	0.537s	66.0%	0.003s	0.4%
10MB	0.832s	0.266s	32.0%	0.562s	67.6%	0.004s	0.5%

Table 2: Experiment 2 (Performance of policy updates). We do not show the AES+HMAC time as it is not involved in policy renewal

Experiment 3 (Performance of multiple policies). We now evaluate the performance of FADE when multiple policies are associated with a file. Here, we focus on the file upload operation, and fix the file size at 1MB. We look at two specific combinations of policies, one on the conjunctive case and one on the disjunctive case.

Table 1 shows different components of time for different numbers of conjunctive policies, and Table 2 shows the case for disjunctive policies. A key observation is that the AES and HMAC and the key management time remain very low (on the order of milliseconds) when the number of policies increases.

Number of policies	Total time	Data transmission				AES+HMAC		Key management	
		File	(%)	Policy	(%)	Time	(%)	Time	(%)
1	5.141s	4.562s	88.7%	0.557s	10.8%	0.022s	0.4%	0.000s	0.0%
2	4.970s	4.352s	87.6%	0.595s	12.0%	0.022s	0.4%	0.000s	0.0%
3	4.667s	3.983s	85.3%	0.662s	14.2%	0.022s	0.5%	0.001s	0.0%
4	4.976s	4.397s	88.4%	0.557s	11.2%	0.022s	0.4%	0.001s	0.0%
5	4.962s	4.406s	88.8%	0.533s	10.7%	0.021s	0.4%	0.001s	0.0%

(a) Conjunctive policies

Number of policies	Total time	Data transmission				AES+HMAC		Key management	
		File	(%)	Policy	(%)	Time	(%)	Time	(%)
1	3.927s	3.364s	85.7%	0.541s	13.8%	0.022s	0.6%	0.000s	0.0%
2	4.015s	3.460s	86.2%	0.534s	13.3%	0.021s	0.5%	0.000s	0.0%
3	3.923s	3.390s	86.4%	0.511s	13.0%	0.022s	0.6%	0.001s	0.0%
4	3.859s	3.322s	86.1%	0.515s	13.3%	0.022s	0.6%	0.000s	0.0%
5	4.118s	3.559s	86.4%	0.536s	13.0%	0.022s	0.5%	0.001s	0.0%

(b) Disjunctive policies

Table 3: Experiment 3 (Performance of multiple policies)

Number of policies	Policy metadata size (bytes)
1	149
2	282
3	415
4	548
5	681

(a) Conjunctive policies

Number of policies	Policy metadata size (bytes)
1	149
2	298
3	447
4	596
5	745

(b) Disjunctive policies

Table 4: Size of the policy metadata

Space Utilization of FADE

We now assess the space utilization. In our research work, there are file metadata and policy metadata for each file, and this metadata information is the space overhead introduced by FADE. For the file metadata, it is always fixed at 28 bytes. On the other hand, for the policy metadata, its size differs with the number of policies. For instance, we need 128 bytes for the policy-based secret key $S_i^{e_i}$ for some policy *i*. The size of an encrypted copy of K is 16 bytes, and this size increases with the number of terms in the case of disjunctive (OR) policies. Table 4 shows the different sizes of the policy 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 $(P_1 \vee P_2 \vee \cdots \vee P_m)$. For instance, if the file size is 1MB and there is only one policy, then the size of the file metadata is 28 bytes and the policy metadata is 149 bytes, and hence the space overhead is 0.017%.

References

[1] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou, "Enabling public verifiabil-ity and data dynamics for storage security in cloud computing," in *Proc. of ESORICS '09,* 2009.

[2] L. Youseff, M. Butrico, and D. D. Silva, "Toward a unified ontology of cloud computing," in *Proc. of GCE'08,* 2008.

[3] A. Yun, C. Shi, and Y. Kim. On Protecting Integrity and Confidentiality of Cryp tographic File System for Outsourced Storage. In *A CM Cloud Computing Security Workshop (CCSW),* Nov 2009

[4] W. Wang, Z. Li, R. Owens, and B. Bhargava. Secure and Efficient Access to Outsourced Data. In *ACM Cloud Computing Security Workshop (CCSW),* Nov 2009.

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

[6] 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.

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

[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[9] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, A Break in the Clouds: Towards a Cloud Definition, Proc. ACM SIGCOMM Computer Communication Review, 39(1), Jan 2009, pp. 50-55.

[10] S. Kamara, and K. Lauter, "Cryptographic Cloud Storage," in Proc. Of Financial Cryptography: Workshop on real life cryptographic protocolsand standardization, 2010, fromhttp://research. microsoft.com/pubs/112576/cry pto-cloud.pdf

[11] S. Yu, C. Wang, K. Ren, and W. Lou, "Achieving Secure, Scalable,and Fine-grained Data Access Control in Cloud Computing," in Proc.of IEEE INFOCOM 2010, 2010.

[12] S. Yu, C. Wang, K. Ren, and W. Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *Proceedings of the IEEE International Conference on Computer Communications* (IN-*FOCOM),* pages 534-542, 2010.

[13] G. Wang, Q. Liu, and J. Wu. Achieving fine-grained access control for secure data sharing on cloud servers. *Concurrency and Computation: Practice and Experience,* 23(12):1443-1464, 2011.