Reliable Re-encryption in Unreliable Clouds

Qin Liu[†‡], Chiu C. Tan[‡], Jie Wu[‡], and Guojun Wang[†]
[†]School of Information Science and Engineering, Central South University, Changsha, Hunan Province, P. R. China, 410083
[‡]Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA
Email:{qin.liu, cctan, jiewu}@temple.edu, csgjwang@mail.csu.edu.cn

***Abstract*—A key approach to secure cloud computing is for the data owner to store encrypted data in the cloud, and issue decryption keys to authorized users. Then, when a user is revoked, the data owner will issue re-encryption commands to the cloud to re-encrypt the data, to prevent the revoked user from decrypting the data, and to generate new decryption keys to valid users, so that they can continue to access the data. However, since a cloud computing environment is comprised of many cloud servers, such commands may not be received and executed by all of the cloud servers due to unreliable network communications. In this paper, we solve this problem by proposing a time-based re-encryption scheme, which enables the cloud servers to automatically re-encrypt data based on their internal clocks. Our solution is built on top of a new encryption scheme, *attribute-based encryption*, to allow fine-grain access control, and does not require perfect clock synchronization for correctness.**

***Index Terms*—Attribute-based encryption, cloud computing, proxy re-encryption.**

I. Introduction

The use of cloud computing is increasingly popular due to the potential cost savings from outsourcing data to the *cloud service provider* (CSP). One technique to protect the data from a possible untrusted CSP is for the data owner to encrypt the outsourced data [1], [2]. Flexible encryption schemes such as *attribute based encryption* (ABE) [3]–[5] can be adopted to provide fine grained access control.

ABE allows data to be encrypted using an *access structure* comprised of different *attributes*. Instead of specific decryption keys for specific files, users are issued attribute keys. Users must have the necessary attributes that satisfy the access structure in order to decrypt a file. For example, a file encrypted using the access structure $\{(\alpha_1 \wedge \alpha_2) \vee \alpha_3\}$ means that either a user with attributes α_1 and α_2, or a user with attribute α_3, can decrypt the file.

The key problem of storing encrypted data in the cloud lies in *revoking* access rights from users. A user whose permission is revoked will still retain the keys issued earlier, and thus can still decrypt data in the cloud. A naïve solution is to let the data owner immediately re-encrypt the data, so that the revoked users cannot decrypt the data using their old keys, while distributing the new keys to the remaining authorized users. This solution will lead to a performance bottleneck, especially when there are frequent user revocations.

An alternative solution is to apply the *proxy re-encryption* (PRE) technique [6], [7]. This approach takes advantage of the abundant resources in a cloud by delegating the cloud to re-encrypt data [8], [9]. This approach is also called command-driven re-encryption scheme, where cloud servers execute re-encryption while receiving commands from the data owner.

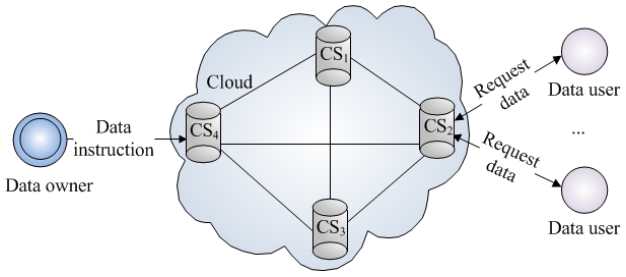

Fig. 1. A typical cloud environment

However, command-driven re-encryption schemes do not consider the underlying system architecture of the cloud environment. A cloud is essentially a large scale distributed system where a data owner's data is replicated over multiple servers for high availability. As a distributed system, the cloud will experience failures common to such systems, such as server crashes and network outages. As a result, re-encryption commands sent by the data owner may not propagate to all of the servers in a timely fashion, thus creating security risks.

To illustrate, let us consider a cloud environment shown in Fig. 1, where the data owner's data is stored on cloud servers CS_1, CS_2, CS_3, CS_4. Assume that the data owner issues to CS_4 a re-encryption command, which should be propagated to CS_1, CS_2, CS_3. Due to a network outage, CS_2 did not receive the command, and did not re-encrypt the data. At this time, if revoked users query CS_2, they can obtain the old ciphertext, and can decrypt it using their old keys.

A better solution is to allow each cloud server to *independently* re-encrypt data without receiving any command from the data owner. In this paper, we propose a *reliable re-encryption scheme in unreliable clouds* (R3 scheme for short). R3 is a time-based re-encryption scheme, which allows each cloud server to automatically re-encrypt data based on its internal clock. The basic idea of the R3 scheme is to associate the data with an *access control* and an *access time*. Each user is issued keys associated with *attributes* and *attribute effective times*. The data can be decrypted by the users using the keys with attributes satisfying the access control, and attribute effective times satisfying the access time. Unlike the command-driven re-encryption scheme, the data owner and the CSP share a secret key, with which each cloud server can re-encrypt data by updating the data access time according to its

own internal clock.

Even through the R3 scheme relies on *time*, it does not require perfect clock synchronization among cloud servers. Classical clock synchronization techniques [10]–[13] that ensure loose clock synchronized in the cloud are sufficient. The main contributions in this paper are as follows:

1) We propose an automatic, time-based, proxy re-encryption scheme suitable for cloud environments with unpredictable server crashes and network outages.
2) We extend an ABE scheme by incorporating timestamps to perform proxy re-encryption.
3) Our solution does not require perfect clock synchronization among all of the cloud servers to maintain correctness.

II. Related Work

Many researchers have proposed storing encrypted data in the cloud to defend against the CSP [1], [2]. Under this approach, users are revoked by having a third party to re-encrypt data such that previous keys can no longer decrypt any data [14]–[16]. The solution by [15] for instance, lets the data owner issue a re-encryption key to an untrusted server to re-encrypt the data. Their solution utilizes PRE [6], which allows the server to re-encrypt the stored ciphertext to a different cipertext that can only be decrypted using a different key. During the process, the server does not learn the contents of the cipertext or the decryption keys.

ABE is a new cryptographic technique that efficiently supports fine grained access control. The combination of PRE and ABE was first introduced by [9], and extended by [8], [17]. In [8], a hierarchical attribute-based encryption (HABE) scheme is proposed to achieve high performance and full delegation. The main difference between prior work and ours is that we do not require the underlying cloud infrastructure to be reliable in order to ensure correctness.

Our scheme relies on time to re-encrypt data. However, in a cloud, the internal clock of each cloud server may differ. There have been several solutions to this problem. For instance, [10] proposed a probabilistic synchronization scheme, which exchanges messages to get remote servers' accurate clocks with high probability. Work by [11] used message delay to estimate the maximal difference between two communicating nodes to synchronize the clocks. Work by [13] proposed a clock synchronization scheme for cloud environments, which uses an authoritative time source shared by all participants in a transaction to achieve clock synchronization between virtual cloud policy enforcement points. By applying these techniques to achieve loose synchronization in the cloud, and to determine the maximal time difference between the data owner and each cloud server, our R3 scheme can always achieve correct access control in unreliable clouds.

III. Preliminary

A. Problem Formulation

We consider a cloud computing environment consisting of a data owner, a cloud service provider (CSP) and multiple data users. The data owner outsources his data in the form of a set of files $F_1, \cdots, F_n$ to the CSP. Each file is encrypted by the data owner before uploading to the CSP. Data users that want to access a particular file must first obtain the necessary keys from the data owner in order to decrypt the file. The data owner can also update the contents of a file after uploading it to the CSP. This is termed a *write* command.

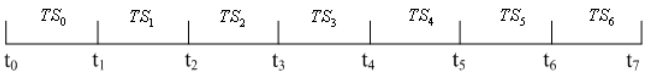

Fig. 2. Sample time slice

TABLE I
Alice's Keys

Key	Description
$SK^1_{a_1}$	Keys for attributes a_1 for TS_1
...	...
$SK^1_{a_m}$	Keys for attributes a_m for TS_1
...	...
$SK^n_{a_1}$	Keys for attributes a_1 for TS_n
...	...
$SK^n_{a_m}$	Keys for attributes a_m for TS_n

Each file, *F*, is encrypted with two parameters, *time slice* and *attributes*. We divide time into time slices, and every time slice is of equal length. We denote a particular time slice, TS, with a subscript, where $TS_i = [t_i, t_{i+1})$. Fig.2 illustrates this concept. Attributes are organized into an *access structure*, $\mathbb{A}$, which regulates access to a file. For example, a file with attributes $\alpha_1, \alpha_2, \alpha_3$ and $\mathbb{A} = \{(\alpha_1 \wedge \alpha_2) \vee \alpha_3\}$, requires either both attributes α_1 and α_2, or just α_3, to satisfy the access structure. A file *F* can only be decrypted with keys that satisfy *both* the access structure and time slice.

A data user, after being authenticated by the data owner, is granted a set of keys, each of which is associated with an *attribute* and an *effective time* that denotes the length of time the user is authorized to possess the attributes. For example, if Alice is authorized to possess attributes $a_1, \ldots, a_m$ from TS_1 to TS_n, she will be issued keys as is shown in Table I.

The security requirements of the R3 scheme are as follows:

1) **Access control correctness.** This requires that a data user with invalid keys cannot decrypt the file.
2) **Data consistency.** This requires that all data users who request file *F*, should obtain the same content in the same time slice.
3) **Data confidentiality.** The file content can only be known to data users with valid keys. The CSP is not considered a valid data user.
4) **Efficiency.** The cloud servers should not re-encrypt any file unnecessarily. This means that a file that has not been requested by any data user should not be re-encrypted.

B. Adversary Model

Our system considers two types of adversaries. The first type of adversary is the CSP. The CSP adversary is considered honest-but-curious. This means that the CSP will always

TABLE II
SUMMARY OF NOTATIONS

Notation	Description
PK	System public key
$\mathbb{UA}$	Universal attributes
PK_a^i	Attribute a's public key at TS_i
sk_a^i	Attribute a's private key at TS_i
MK	Master key
s	Shared secret key
$\mathcal{A}$	Alice's attributes
$\mathcal{T}$	Effective time of Alice's attributes
$\mathbb{A}$	Access control
PK_u	User public key
SK_u	User identity secret key
$SK_{u,a}^i$	User attribute secret key with attribute a and time TS_i

correctly execute a given protocol, but may try to gain some additional information about the stored data.

The second type of adversary is malicious data users. The data user adversary will try to learn the file content that he is not authorized to access. This adversary is assumed to possess invalid keys (either with incorrect attributes or time). We also assume the data user adversary can query any server in the cloud. Note that *both* an honest-but-curious CSP and malicious data users can exist together. However, we assume that the CSP and data users will not collude to break the system, because the CSP is considered to be honest-but-curious.

IV. BASIC R3

In the basic R3 scheme, we consider ideal conditions, where the data owner and all of the cloud servers in the cloud share a synchronized clock, and there are no transmission and queueing delays when executing read and write commands.

A. Intuition

The data owner will first generate a shared secret key to the CSP. Then, after the data owner encrypts each file with the appropriate attribute structure and time slice, the data owner uploads the file in the cloud. The CSP will replicate the file to various cloud servers. Each cloud server will have a copy of the shared secret key.

Let us assume that a cloud server stores an encrypted file *F* with $\mathbb{A}$ and TS_i. When a user queries that cloud server, the cloud server first uses its *own* clock to determine the current time slice. Assuming that the current time slice is TS_{i+k}, the cloud server will automatically re-encrypt *F* with TS_{i+k} without receiving any command from the data owner. During the process, the cloud server cannot gain the contents of the cipertext and the new decryption keys. Only users with keys satisfying $\mathbb{A}$ and TS_{i+k} will be able to decrypt *F*.

B. Protocol Description

We divide the description of the basic R3 scheme into three components: data owner initialization, data user read data and data owner write data. We will rely on the following functions. Table II shows the notations used in the description.

1) $Setup() \rightarrow (PK, MK, s)$: At TS_0, the data owner publishes the system public key PK, keeps the system master key MK secret, and sends the shared secret key s to the cloud.
2) $GenKey(PK, MK, s, PK_{Alice}, \mathcal{A}, \mathcal{T}) \rightarrow (SK_{Alice}, \{SK_{Alice,\mathcal{A}}^{\mathcal{T}}\})$: When the data owner wants to grant data user Alice attributes $\mathcal{A}$ with valid time period $\mathcal{T}$, the data owner generates SK_{Alice} and $\{SK_{Alice,\mathcal{A}}^{\mathcal{T}}\}$ using the system public key, the system master key, the shared secret key, Alice's public key, Alice's attributes and eligible time.
3) $Encrypt(PK, \mathbb{A}, s, TS_t, F) \rightarrow (C_{\mathbb{A}}^t)$: At TS_t, the data owner encrypts file F with access structure $\mathbb{A}$, and produces ciphertext $C_{\mathbb{A}}^t$ using the system public key, access structure, the system secret key, time slice, and plaintext file.
4) $Decrypt(PK, C_{\mathbb{A}}^t, SK_{Alice}, \{SK_{Alice,a_{ij}}^t\}_{1\leq j\leq n_i}) \rightarrow F$: At TS_t, user $\mathcal{U}$, who possesses version t attribute secret keys on all attributes in CC_i, recovers F using the system public key, the user identity secret key, and the user attribute secret keys.
5) $REncrypt(C_{\mathbb{A}}^t, s, TS_{t+k}) \rightarrow C_{\mathbb{A}}^{t+k}$: When the cloud server wants to return a data user with the file at TS_{t+k}, it updates the ciphertext from $C_{\mathbb{A}}^t$ to $C_{\mathbb{A}}^{t+k}$ using the shared secret key.

1) Data owner initialization: The data owner runs the *Setup* function to initiate the system. When the data owner wants to upload file *F* to the cloud server, it first defines an access control $\mathbb{A}$ for *F*, and then determines the current time slice TS_i. Finally, it runs the *Encrypt* function with $\mathbb{A}$ and TS_i to output the ciphertext. When the data owner wants to grant a set of attributes in a period of time to data user Alice, it runs the *GenKey* function with *attributes* and *effective times* to generate keys for Alice.

2) Data user read data: When data user Alice wants to access file *F* at TS_i, she sends a read command $R(F)$ to the cloud server, where *F* is the file name. On receiving the read command $R(F)$, the cloud server runs the *REncrypt* function to re-encrypt the file with TS_i. On receiving the ciphertext, Alice runs the *Decrypt* function using keys satisfying $\mathbb{A}$ and TS_i to recover *F*.

3) Data owner write data: When the data owner wants to write file *F* at TS_i, it will send a write command to the cloud server in the form of: $W(F, seqnum)$, where $seqnum$ is the order of the write command. This $seqnum$ is necessary for ordering when the data owner issues multiple write commands that have to take place in one time slice. On receiving the write command, the cloud server will commit it at the end of TS_i. Algorithm 1 shows the actions of the cloud server.

Algorithm 1 Basic R3 (synchronized clock with no delays)

while Receive a write command $W(F, seqnum)$ at TS_i **do**
  Commit the write command in order at the end of TS_i
while Receive a read command $R(F)$ at TS_i **do**
  Re-encrypt file with TS_i

C. Security analysis

Access control correctness. It is clear that the correctness of access control is most vulnerable when a TS changes. Let us consider the case where Alice has keys with effective time up to TS_i, and Bob has keys with effective time starting from TS_{i+1}. Assuming that the data owner updates file F to F' such that a user querying the file at TS_i should obtain F, and a user querying the file at TS_{i+1} should obtain F'. The property of access control correctness fails if Alice is able to read F' (attack 1), or if Bob is able to read F (attack 2).

In attack 1, Alice's best time to launch an attack is just before t_{i+1}, since she only has the keys to decrypt data up to TS_i. However, the cloud server will commit the write command at t_{i+1} as long as its own clock is consistent with the data owner's clock, so that Alice never reads F', and thus her attack fails.

In attack 2, Bob's best time to launch an attack is just after t_{i+1}. Querying earlier than t_{i+1} does not help Bob since he does not have the keys to decrypt the data. However, since the cloud server will commit the write command at t_{i+1} as long as its own clock is consistent with the data owner's clock, Bob will never access F, but only F'. Therefore, our scheme provides correct access control.

Data consistency. This property requires users that query within the same TS must receive the same data. Let us assume that both Alice and Bob have valid keys for the appropriate time slices, and we now want to show that so long as both Alice and Bob query within the same time slice, they must obtain the same data. Assuming that Alice and Bob both pick TS_i to attack our scheme, the best attack time for Alice is to query just after t_i, and for Bob is to query just before t_{i+1}. This attack is depicted in Fig. 3. According to the R3 scheme, the cloud server will return data that has been committed in t_i to both Alice and Bob.

We first note that any write command that occurs *after* Alice and *before* Bob does not affect the correctness of the R3 scheme since this command will be committed at t_{i+1}. Furthermore, we have to ensure that all writes committed at t_i (what we are returning to Alice and Bob) must have already arrived before t_i. Since all parties' clocks are consistent and there are no delays, any write command committed at t_i can only be received by the cloud server before t_i. Thus, the data returned to Alice and Bob is consistent.

Data confidentiality. In our scheme, we only store encrypted data in the cloud. Since the R3 scheme preserves the data confidentiality operations from HABE scheme, and retain the same confidentiality properties, the cloud without knowledge of keys cannot learn any useful information about the stored data.

Data efficiency. The cloud server does not re-encrypt a file until a data user requests that file. Based on the properties of function *REncrypt*, when $k > 1$, we see that the cloud server can combine the re-encrypt operations until receiving a file access request.

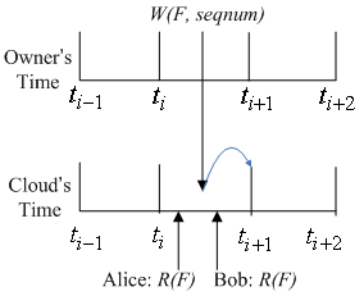

Fig. 3. Attacks to compromise the property of data consistency

Algorithm 2 Extended R3 (asynchronized clock with delays)

while Receive a write command $W(F, t_{i+1}, seqnum)$ **do**
  if Current time is earlier than $t_{i+1} + \alpha$ **then**
    Build Window i for file F
    Commit the write command in Window i at $t_{i+1} + \alpha$
  else
    Reject the write command
    Inform the data owner to send write command earlier
while Receive a read request $R(F, TS_i)$ **do**
  if Current time is later than $t_{i+1} + \alpha$ **then**
    Re-encrypt the file in Window i with TS_i
  else
    Hold on the read command until $t_{i+1} + \alpha$

V. Extended R3

In this section, we consider the scenario where there is no synchronized clocks. We also consider transmission and queueing delays during the write and read commands.

A. Protocol Description

We let the data owner and the cloud server agree on a maximal waiting time α. Thus, the cloud server will wait until $t_i + \alpha$ to commit the write commands that should be committed at t_i, and to respond to the read commands for reading data at TS_i. The data owner and data user will include additional information in their write or read commands. When the data owner wants to update the file F at TS_i, he will issue a command $W(F, t_{i+1}, seqnum)$, where F is the file name, t_{i+1} is when the updates have to take place and $seqnum$ is the order of the write command. When the data user wants to read the file F at TS_i, he will use a command $R(F, TS_i)$.

Then, we need to determine the maximal time difference between the data owner and the cloud server. We denote this time difference as $\triangle$, where $\triangle$ is no larger than the duration of one time slice. In other words, when the data owner is at TS_i, the cloud server's time may be TS_{i-1}, TS_i, or TS_{i+1}. We let the data owner issue his write command before t_{i+1} when he wants this update to be reflected in TS_{i+1}. Algorithm 2 shows the actions of the cloud server.

B. Security Analysis

Access control correctness. Here, we need to show that Algorithm 2 maintains the property of access control correctness

using the same attack 1 and attack 2 as the security analysis in the basic R3 scheme.

In attack 1, Alice's best time to launch her attack is just before t_{i+1}, since she only has the keys to decrypt data up to TS_i. However, the cloud server will commit the write command at $t_{i+1} + \alpha$, so that Alice never reads *F'*, and thus her attack fails.

In attack 2, Bob's best time to launch his attack is just after t_{i+1}. Querying earlier than t_{i+1} does not help Bob since he does not have the keys to decrypt the data. However, the cloud server will commit the write command at $t_{i+1} + \alpha$, and hold the read command until committing all write commands. Therefore, Bob never reads *F*, but only *F'*.

Data consistency. We use the same attack scenario as the security analysis in the basic R3 scheme. The cloud server will reject all of the write commands that should be committed at $t_i + \alpha$ if its time is past $t_i + \alpha$. Then, the cloud server will hang up all of the read commands for TS_i until committing all of the write commands at $t_i + \alpha$. Therefore, data is consistent.

The analyses for both **data confidentiality** and **data efficiency** are the same as the basic R3 scheme.

VI. Additional Discussion

One concern with the R3 scheme design is that associating a different ciphertext for every time slice will require users to manage a lot of keys. The number of keys that the R3 scheme requires is related to the actual length of the time slice. This length can be set according to different application requirements. Thus, an application that expects to revoke users on a monthly basis will have a longer time slice, and hence have far fewer keys, than an application where membership changes by the hour. Furthermore, issuing multiple keys upfront is actually *more* efficient. Consider an alternative design where each valid user was issued just one key. Now, every time any user is revoked, the owner has to inform the CSP to re-encrypt the previous ciphertext so as to prevent the revoked user from decrypting it again. The owner then has to update all of the remaining valid users with the new keys to allow them to decrypt the new ciphertext. We argue that any scheme that stores encrypted data in the cloud has to deal with the issue of re-encryption and re-keying. Since the remaining users may not be online all of the time, the re-keying process is arguably more costly. A possible improvement is to let the owner issue a valid user a special *seed* value which the user can then use to generate keys on his own. The challenge here is to prevent the user from generating additional keys beyond what is authorized. This remains part of our future research.

Furthermore, we only let the data owner perform data updates. This is inflexible for applications where users may need to update the data as well. Our solution can be extended to allow users to perform data updates in addition to data owners. A ticketing scheme can be used. The data owner will issue and sign a timestamp to authorize the user to perform a write. The user will submit the ticket together with his updates to the CSP, which will then apply the updates. The challenge here is the time lag between when the data owner issues the ticket and when the user's request reaches the CSP. This time lag may be unknown since the user may delay sending his update to the cloud. An additional protocol will be required to allow the CSP to reject update requests that are too close to the time slice borders.

VII. Conclusion

In this paper, we proposed the R3 scheme, a new method for managing access control based on the cloud server's internal clock. Our technique does not rely on the cloud to reliably propagate re-encryption commands to all servers to ensure access control correctness. We showed that our solutions remain secure without perfect clock synchronization so long as we can bound the time difference between the servers and the data owner.

Acknowledgements

This research was supported in part by NSF grants ECCS 1128209, CNS 1065444, CCF 1028167, CNS 0948184, CCF 0830289; and National NSF of China under Grant No. 61073037, Hunan Provincial Science and Technology Program under Grant No. 2010GK2003, and National 973 Basic Research Program of China under Grant No. 2011CB302800.

References

[1] S. Kamara and K. Lauter, "Cryptographic cloud storage," *Financial Cryptography and Data Security*, 2010.

[2] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "A view of cloud computing," *Communications of the ACM*, 2010.

[3] A. Sahai and B. Waters, "Fuzzy identity-based encryption," *Advances in Cryptology–EUROCRYPT*, 2005.

[4] 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.

[5] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *Proc. of IEEE Symposium on S&P*, 2007.

[6] M. Blaze, G. Bleumer, and M. Strauss, "Divertible protocols and atomic proxy cryptography," *Advances in Cryptology–EUROCRYPT*, 1998.

[7] A. Boldyreva, V. Goyal, and V. Kumar, "Identity-based encryption with efficient revocation," in *Proc. of ACM CCS*, 2008.

[8] G. Wang, Q. Liu, and J. Wu, "Hierarchical attribute-based encryption for fine-grained access control in cloud storage services," in *Proc. of ACM CCS (Poster)*, 2010.

[9] 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.

[10] F. Cristian, "Probabilistic clock synchronization," *Distributed Computing*, 1989.

[11] K. Romer, "Time synchronization in ad hoc networks," in *Proc. of ACM MobiHoc*, 2001.

[12] P. Ramanathan, K. Shin, and R. Butler, "Fault-tolerant clock synchronization in distributed systems," *Computer*, 2002.

[13] N. Antonopoulos and L. Gillam, "Cloud Computing: Principles, Systems and Applications," *Springer Publishing Company*, 2010.

[14] 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.

[15] G. Ateniese, K. Fu, M. Green, and S. Hohenberger, "Improved proxy re-encryption schemes with applications to secure distributed storage," *ACM Transactions on Information and System Security*, 2006.

[16] S. Di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Over-encryption: management of access control evolution on outsourced data," in *Proc. of VLDB*, 2007.

[17] S. Yu, C. Wang, K. Ren, and W. Lou, "Attribute based data sharing with attribute revocation," in *Proc. of ACM ASIACCS*, 2010.