2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing

TrustStore: Making Amazon S3 Trustworthy with Services Composition

Jinhui Yao
School of Electrical and Information Engineering, University of Sydney
Sydney, Australia
jinhui@ee.usyd.edu.au

Shiping Chen
Networking Technologies Laboratory, CSIRO ICT Centre, Australia
Sydney, Australia
shiping.chen@csiro.au

Surya Nepal
Networking Technologies Laboratory, CSIRO ICT Centre, Australia
Sydney, Australia
surya.nepal@csiro.au

David Levy
School of Electrical and Information Engineering, University of Sydney
Sydney, Australia
dlevy@ee.usyd.edu.au

John Zic
Networking Technologies Laboratory, CSIRO ICT Centre, Australia
Sydney, Australia
john.zic@csiro.au

***Abstract*—The enormous amount of data generated in daily operations and the increasing demands for data accessibility across organizations are pushing individuals and organizations to outsource their data storage to cloud storage services. However, the security and the privacy of the outsourced data goes beyond the data owners' control. In this paper, we propose a service composition approach to preserve privacy for data stored in untrusted storage service. A virtual file system, called TrustStore, is prototyped to demonstrate this concept. It allows users utilize untrusted storage service provider with confidentiality and integrity of the data preserved. We deployed the prototype with Amazon S3 and evaluate its performance.**

Keywords-Trustworthy Storage Services, Services Composition, Cloud Storage

I. Introduction

Storage system has become a critical component for enterprises (as well as individual)'s daily operations. Data contents (transactions, sales records, market analysis data, personal video collections, etc) will accumulate to an enormous volume over the time. This quantity can eventually go beyond the capability of organizations and individuals to store and manage. Moreover, some data needs to be available on the Internet for universal access and easy sharing among multiple parties.

Those needs have triggered the emergence of cloud storage services known as Storage Services Providers (SSP) that allow customers (organizations and individuals) to outsource the burden of data storage and management. The well-known SSPs include: Amazon S3 [13], Megaupload.com, Yahoo Briefcase, Gmail, Ncipher [18], GoGrid [19]. Considerable applications have been built on top of this new storage concept, such as Jungle Disk [3], Gmail Drive [19], S3 BackUp [22]. These applications provide convenient interface for clients to interact with those storage "unlimited" capacity. However, the main drawback of using SSP is the loss of control on the outsourced data, which can be vulnerable to the following attacks:

- *External attacks*: Hackers break into the SSP's system to steal and/or browse the stored customer data. This kind of attacks can be protected through traditional approaches such as the techniques under the generic umbrella of "intrusion tolerance" [28][29]. However the implementation of the security techniques is entirely up to the SSP which exhibits great uncertainty about its bearability to the attacks.
- *Internal attacks*: Malicious employees can easily breach into the system to steal/browse the data and profit from it. Studies revealed that the majority of confidential data leaking are from internal [21]. This kind of attacks can't be protected by classical security technologies (firewall, anti-virus, etc).

The concerns with loss of privacy control has significant impact on SSP's trustworthiness and result great reluctance when using it [24]. In this paper, we address those concerns with the following contributions: (a) we propose a service-oriented architecture for provisioning Trustworthy Storage Services (TSS) with untrusted storage service providers; (b) we propose a simple data model, to analysis both confidentiality and integrity of the data outsourced to the cloud storage (c) we implemented a prototype called TrustStore to illustrate the concept; (d) we conducted a performance evaluation of the TrustStore using Amazon S3.

II. Provisioning Trustworthy Storage Service

Fig. 1 illustrates our architectural design of Provisioning Trustworthy Storage Service (TSS). As we can see, there are three parties involved in our design, they are: client computer, key management service provider (KMSP) and storage service provider (SSP). The basic roles of the three parties in the composition are described as following:

978-0-7695-4039-9/10 $26.00 © 2010 IEEE
DOI 10.1109/CCGRID.2010.17

- *Storage Service Provider (SSP)*: The outsourcing data contents are encrypted and stored in the SSP. Therefore the SSP holds only the cipher of the data contents.
- *Key Management Service Provider (KMSP)*: KMSP acts as a key registry that issues, stores and manages the keys for the client, it has no knowledge about the use of the stored keys.
- *Client Computer*: The client has an application installed on the local machine (TrustStore), which is responsible for conducting data outsourcing process by composing SSP and KMSP.

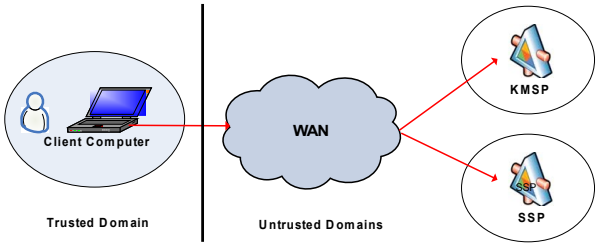

Figure 1. Overall architecture of Trustworthy Storage Services

The main intuition of the above architecture is to minimize the capability of each of the participants to threat the privacy of the outsourced data. In another word, we intend to properly distribute trust onto SSP and KMSP. Our architecture is based on the following assumptions on the trust relationship:

- The client's machine is entirely trusted and secured for sensitive data operations and computations (creating, accessing, outsourcing, etc.). Note the client computer security is a traditional security issue and is out of the scope of this paper.
- Both SSP and KMSP are semi-trusted. They will manage to provide services they claim, such as: (a) persistently storing the customer data; (b) certain access control. However, they are subject to both internal and external attacks.
- SSP and KMSP are trusted to have little or no knowledge of each other. In fact, neither of them should be aware of the existence of the other.

Our trust storage architecture described above makes intuitive sense. The client is trusted to operate with its own sensitive data but not trusted to be capable of keeping them safe in time. Persistent storage services SSP and KMSP on the other hand are not trusted with their privacy protections. However, given that both of them are randomly selected, the trust on the composition is acceptable, because the possibility of both SSP and KMSP being jeopardized simultaneously is extremely low. With this trust distribution, we are able to develop the process for outsourcing operation as follows:

- In the client computer, sensitive data is first processed with encryption mechanism to transform the data into two forms: cipher text form and key form.
- The cipher text form data is uploaded to SSP. Without the key form, the SSP will not be able to reverse the process and access to the sensitive information.
- The key form data is stored with KMSP. It has the capability to reverse the encryption process however it does not have the cipher text form to apply.

The concept of separating the encryption key from the cipher text is not new in the area. In fact, it has been long proposed and adopted in protecting privacy [8]. Our work here is a demonstration of this concept to achieve trustworthy storage outsourcing in the cloud storage domain. We take advantage of the persistency of storage services for archiving both forms generated from the sensitive data. Then the services from different providers will achieve the separation unawarely. Only the encryption computation is done locally in the most trusted party. This kind of triangular entity composition demonstrates a method for the client to achieve provisioning trustworthy storage by utilizing existing services. With this main concept, in the next section we will illustrate the details of the encryption mechanisms that can significantly improve the protection offered by this topology.

III. Data Modeling

The growth of computing power also gives a huge raise to brute forcing attacks. Breaking a cipher by brute forcing becomes possible though costly [26]. Therefore simply encrypt & separate puts heavy trust on the encryption algorithm. In this section, we present the detailed data modeling method we use in TSS. The modeling significantly enhance the protection offered by the encryption and ensures that any corruptions carried out by SSP or/and KMPS face a high probability of detection. We will discuss this in two parts. The first part addresses the confidentiality while the second part addresses the integrity.

A. Data Confidentiality

Information can be observed from a file in different ways. Apart from its content, the meta-data and file directory structure also can tell much about the secrets contained in the file. With this understanding, a file (ψ) is modeled as below:

$$\psi = \{M, X, \sum_{i=1}^{n} \psi_i\} \quad (1)$$

where M represents meta-data, including filename, size, type, modified time. A file can be either a concrete file, whose contents is not null: $X \neq \emptyset$, or a directory that contains n child files.

To encrypt a concrete file, it is important the disclosure of its meta-data in the generated cipher text form is minimal. To be particular, the name and the size of the file must be hidden as well (type and modified time will be changed by the operating system). To do so, we propose to partition the file into a set of fragment, which is defined as:

$$f = \{fID, m, x\} \qquad m = \{kID, filename, order\} \quad (2)$$

where *fID* is the Fragment ID, it is randomly generated on fly and used as the filename of that fragment. m represents the

fragment meta-data, including the ID of the key used to encrypt this fragment, the filename of the original file and the fragmentation order. x is the fragment of the content. We can redefine a concrete file ($\bar{\psi}$) as a set of fragments as follows:

$$\bar{\psi} = \sum_{i=1}^{p} f_i = \{\sum_{i=1}^{p} fId_i, \sum_{i=1}^{p} m_i, \sum_{i=1}^{p} x_i\} = \{\omega, \sum_{i=1}^{p} x_i\} \quad (3)$$

where p is the number of fragments for a file, and ω represents all the meta-info of these fragments. Note that not only p needs to be generated randomly on the fly, but also the size of each fragment need to be different. In this way, we can have the meta-data (M in (1)) completely hidden into the fragments.

As illustrated in (1) that hierarchical structure in folders also contain certain amount of information about the files. We need a model to contain and reflect such structure so they can be stripped off. Here we define *File Object* (ϕ):

$$\phi = \{ID, \omega, \sum_{i=1}^{n} \phi_i\} \quad (4)$$

Each file will have an associated *File Object*, it has the unique ID which is randomly assigned to the file, and the meta-info of the file's fragments in case of concrete files. In case of a directory, the *File Object* will contain the *File Objects* of all its underlying child files. With this model, the meta-info and file structure of a whole collection of files can be encapsulated into one single *File Object*, we call it the *Root File Object* (ϕ_R). Therefore the encryption of a collection of files can be expressed as below:

$$Enc(\sum_{i=1}^{n} \psi_i) = \{\sum_{i=1}^{q} x_i \otimes k_i, \phi_R \otimes k_{q+1}\} \quad (5)$$

where $\otimes$ represents the encryption operation. q+1 keys are involved in which one is used for the *Root File Object* and the the rest used for all the fragments generated from n files in the collection. This equation illustrates the cipher text form data after the encryption process, this form will be stored at the SSP. We can see that the file meta-data and the file structure are indeed all transformed into the cipher text same as the content.

The keys used in (5) are issued from the KMSP upon request. They are the main part of the key form, which is described as follows:

$$Keys = \{\sum_{i=1}^{q+1} (k_i, kID_i), R_{ID}, R_{kID}\} \quad (6)$$

the key form data includes a collection with size q+1 of encryption keys in pair with their key IDs, it also includes the ID of the *Root File Object* (R_{ID}) and its associated key ID (the ID of the key used to encrypt the *Root File Object* - R_{kID}). Since the fragment IDs and the associated fragment key IDs are all contained in the *Root File Object*. This is the first file that needs to be fetched and decrypted so that other fragments can be processed, thus its file name (R_{ID}) and the key ID (R_{kID}) is required for the key form. The key form is stored at the KMSP but neither cipher text form nor original content is disclosed to it, hence the purpose of separation of encryption key from the cipher text can be achieved.

B. Data Integrity

Preserving data integrity is another important aspect of privacy and hence trustworthiness. Intuitively, one will not have the sense of privacy if his data can be altered by others without his awareness. In our model, we approach the data integrity from the viewpoint of *soundness* and *freshness*.

Soundness for integrity refers to the trueness of the data stored. This property requires the awareness of the data owner for any tampering happened to the data [1]. To incorporate this property, we utilize collision-resistant hash function [4] to calculate the hash of each of the fragments, and put the hashes into their meta-data (m in (2)) :

$$m = \{kID, filename, order, hash(x)\} \quad (7)$$

where x refers to the content in this fragment. Every time a fragment is fetched from SSP, the hash contained in the *Root File Object* will be used to validate its *soundness*, if the hash of the fetched data is different, the data must have been tampered.

Freshness requires the fetched data to be in its most recent version . Any attempts of SSP/KMSP to revoke the change done by the owner must be caught. In our model, this is achieved trivially. The cipher text in SSP must have a correct match with the key in KMSP. If either of the two service providers roll back the stored data to an earlier state, this will be noticed by the owner when he/she cannot conduct a successful decryption.

C. Security Analysis

In this section, we will analysis the degree of protection offered by our model and the difficulty for a successful attack, this will also be discussed in the two aspects: *confidentiality* and *integrity*.

Confidentiality, according to our model, the information stored in two service providers have the following distribution:

$$SSP = \{Auth_{ssp}, \sum_{i=1}^{q} x_i \otimes k_i, \phi_R \otimes k_{q+1}\} \quad (8)$$

$$KMSP = \{Auth_{kmsp}, \sum_{i=1}^{q+1} (k_i, kID_i), R_{ID}, R_{kID}\} \quad (9)$$

The most convenient way to view the content is to have the authentication credentials of both services $\{Auth_{ssp}, Auth_{kmsp}\}$. However, one of the strengths of authentications is its limited number of attempts. The chance an attacker can guess out the credentials under this restriction is negligible. Instead, attackers may seek other weaknesses of the service provider to gain access. However, given that two service providers are randomly chosen in a vast amount of options, the chance both get intruded simultaneously by the same attacker is extremely lower compare to the single service provider scenario.

Although intrusions to both KMSP and SSP are highly unlikely, by managerial loopholes (e.g. badly managed internal access control) attackers may gain access to one of the service providers. Here we will discuss the protection TSS can offer when an attacker successfully intrude SSP (intrusions on KMSP are trivial as it does not store any content).

After a successful intrusion, the intruder needs to identify the fragments for the target file he intends to browse. Intuitively, the most efficient way to know that is to brute force the *Root File Object*. *A*veragely it takes half of the population to hit the *Root File Object* among the whole collection of encrypted fragments. Once the half of the population have been successfully brute forced, averagely we can expect half of the fragments of the target file are already included in the brute forced ones. Therefore, in average, the number of brute forcing needed for one target file is half of the total number of the fragments in SSP plus half of the number the target file is fragmented. Compare to simply encrypt a file with a single key, we can see there is a significant protection gain offered by our model. Furthermore, the more files stored in SSP the greater the gain will be. For any type of encryption algorithm, its strength can be significantly improved by this quantity.

Integrity, attacks to the data integrity are mainly in two forms: *deletion* and *corruption*. If *deletion* happens to either cipher text form or key form data, there will be fragments without keys or keys without target fragments. The specific *deletion* will be detected when any broken-pair is noticed. Because all retrieved files are to be hashed again to compare with the hash stored in the *Root File Object*. For the SSP to modify fragments without being detected, it will require finding collision in a collision resistant hash function, therefore the probability is negligible.

IV. Prototype - TrustStore

In order to illustrate our concept, we have implemented a prototype of our architecture. The prototype is consisted of two remote services and a client side application. These three components represent the three parties in Fig. 1 to form the composition. In this section, we basically describe the deployment of SSP and KMSP, then we explore some details of the client side application – TrustStore.

SSP: Amazon S3 [15] is used to store the cipher text form data in (5). S3 allows standard storage operations such as: PUT, GET, LIST & DELETE. The client end – TrustStore conducts these operations through sending SOAP messages to S3.

KMSP: A key management web service is implemented and deployed in EC2. It generates the encryption keys upon request and stores the key form data in (6).

TrustStore: It is implemented as a GUI-based java application. Fig. 2 is a screen-shot of TrustStore, we can see its appearance is very similar to a normal split-view file explorer in modern operating systems. The application is to be used as a tool to take care of the complications in data processing and service coordinating at the background. Users can utilize TrustStore to achieve TSS by simple standard operations such as drag & drop. Its detailed functionalities are illustrated in the following standard operations:

- *Log in*: The user will be authenticated by both SSP and KMSP. *Root File Object* is first fetched. Upon its successful decryption the file collection will be displayed. An empty one is created for the first time.
- *PUT*: Client can upload a file by simply drag-n-drop it into the GUI. The target file will be transformed according to the model and uploaded to SSP & KMSP.
- *GET*: Double click on a file, if its status is "uploaded" (see Fig. 2), its respective cipher text form and key form data will be fetched. After the decryption, its hash will be computed to be validated with the stored hash value.
- *Structural operation*: Changes to the structure of the file collection will only cause changes to the *Root File Object*. The structure information will only be used for display and do not affect the stored fragments.
- *Delete & Update*: *Delete* can be done by pressing the "Del" key to the target file, related data will be deleted from both services. *Update* is achieved through a *Delete* followed by a *Put*.
- *Log off*: When exiting, the updated *Root File Object* will be encrypted and uploaded to replace the old version. Changes to the file structure and meta-data (e.g. file name) will not suffer delay because they are done locally and transmitted at the end of every user session.

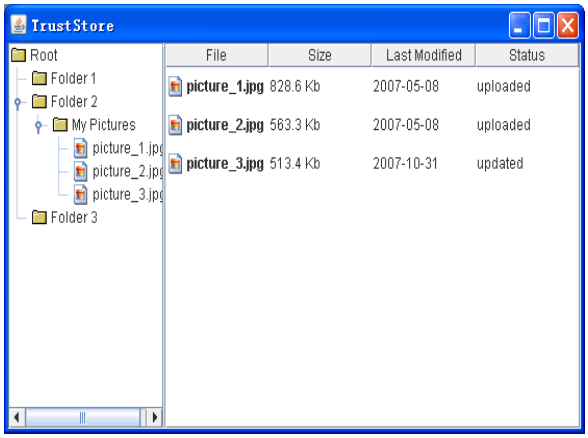

Figure 2. Screen shot of TrustStore GUI

V. Performance Evaluation

To demonstrate the viability of our design, we have conducted testings to observe the latency of TrustStore compared to the ordinary use of Amazon S3. As described in previous section, the PUT/GET operations in TrustStore involve several extra processes apart from the transmission. They are key requesting, content hashing, fragmentation and cryptography. In this section we first show the testing results on each of the extra process, then we demonstrate our approach to minimize the overall costs through multi-threading. The tests are carried out on a computer with an Intel Pentium 1.72Ghz processor and 512 MB Ram, the computer is located in Sydney Australia while the Amazon S3 and EC2 servers are located in America. The Internet access we tested with is ADSL2+,

theoretically it can achieve 24,000kbps (about 2.4MB/sec) download speed and 1000kbps (about 0.1MB/sec) upload speed. Below are the latency of individual processes:

TABLE I. PROCESS TESTING RESULTS

Key Requesting	1.08 sec	
Hashing (SHA-1)	0.03 sec/MB	
Fragmentation	negligible	
Cryptography (AES 256 bit key)	Encryption	0.16 sec/MB
	Decryption	0.12 sec/MB

The key requesting test is conducted by transmitting 50 keys with length of 256 bit. We noticed that because the size of keys are relatively small compare to messages, the quantity of keys in the request does not affect the requesting latency much, the time is mainly determined by the distance to the KMSP. Other processes are all internal computations whose latency grows linearly with the amount of data to process. According to Table I, the extra latency for a GET of a 10MB file will be around 2.98 sec, while directly transmitting the file will take around 14.7 sec. Intuitively, this latency must be noticeable if not significant, however, this cost can be substantially reduced by applying multi-threading techniques. As aforementioned, the data are stored at S3 as a collection of fragments (5), this approach itself provides convenience for deploying multiple threads to conduct PUT/GET.

We first experimented the data transmission rate with S3 by using multiple threads. An interesting fact we noticed is that Amazon S3 allocates limited transmission bandwidth to an active transmission thread. As the number of the threads increases, the transmission rate of each thread will decrease. Nevertheless, the aggregate rate of transmission will increase until a peak is reached and thereafter no more bandwidth gain can be achieved by using more threads (this is also noticed in [27]). The peak transmission rate found in our testing is much slower than the processing speeds in Table I. This result is quite as expected as the internet speed is significantly slower than the internal computing speed. Therefore, if the extra processes introduced in TrustStore are carried out by different threads while other threads are transmitting, their latency will not be noticed.

For instance, to conduct PUT, the processing threads can start by processing only a small portion of the data to produce the first batch of fragments. When this batch is ready, different threads will upload them while the processing thread continues to process the remaining data. In this way, users will experience processing latency only for the first batch . Similar method can be applied to GET. Downloading threads are deployed to continuously fetch keys and data fragments while processing threads process the arrived key/fragment pairs to restore the data. In contrast to PUT, the noticeable processing latency for GET will be the time used to process the last batch.

Fig. 3 and Fig. 4 displays the overall performance of Truststore for PUT/GET operations compared to the ordinary use of Amazon S3. The term "ordinary" here refers to the normal use with S3 without any extra process introduced in TrustStore (encryption, fragmentation, multi-threading, etc.). From the graphs, we can see except for small files (100kb), the performance of TSS actually outperforms the latency of ordinary use. We have achieved a peak PUT rate of 10.7 sec/MB with a speed up of 2.2, and a peak GET rate of 1.98 sec/MB with a speed up of 12. These peak speeds are very close to the direct upload/download speed with S3 by multiple threads. Similar performance gain can be expected for larger data sets as long as the number of threads are tuned to suit the environments. Therefore, it is reasonable to draw a conclusion from the experiments that the performance of our design is quite acceptable and is suitable to be implemented for real practice.

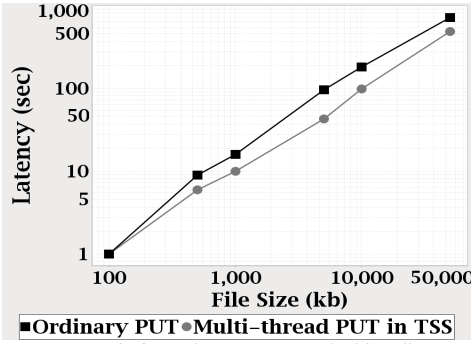

Figure 3. Speed of PUT in TSS compared with ordinary PUT

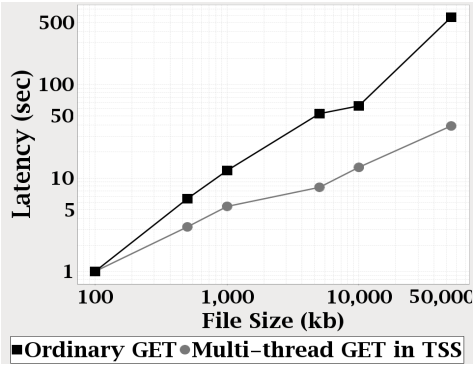

Figure 4. Speed of GET in TSS compared with ordinary GET

VI. RELATED WORK

In the field, there are other research works conducted to enhance the security of storage outsourcing. [1] and [3] deployed a master password to generate encryption keys, which are used to encrypt the data stored remotely. [2] uses a similar method, instead, a random binary key is used as the master key. SHAROES [25] issues each user a private/public key pair. The private key is used to encrypt both the content and the meta-data of the data stored. Aforementioned approaches share the concept with ours that different keys are used when encrypting a file collection. However, since those keys are all from the same master key, this master key becomes ultra important in the system and is the only key needs to be brute forced. Sirius

[5] and Plutus [6] allows clients to use different keys generated on-the-fly without any master key. Clients are then responsible for the key distribution and management. [4] and [9] use asymmetric cryptography to maintain undeniable records to protect data integrity. They do not focus on the privacy of the data.

Some studies also propose to compose multiple service entities for storage outsourcing. [8] breaks down the relational information in the sensitive data (e.g. user-name & password) and store them in different service providers. [10] computes the hashes of all files stored in one service provider and store them in another. Cepheus [7] introduces a key service to contain the keys while the other data service contains the cipher. Each client manages a private/public key pair to encrypt the keys before they are stored. Again the private key is the only secret in the entire protection. Their concept of separation is greatly refined in our approach. We rely on the authentication credentials in two different service providers to separate the cipher form and the key form data. This allows us improve the strength of the encryption by the quantity of keys used without introducing a single master key.

VII. Conclusions

In this paper, we present the design of Trustworthy Storage Service, and the implementation – TrustStore, which allows the client to leverage the resource of Provisioning Storage Service Provider with both confidentiality and integrity preserved. Our prototype implementations and corresponding experimental results show: (a) the confidentiality of the outsourced data is completely preserved. Not only the contents but also meta data and the file structure are encrypted; (b) our system is portable, it requires no data being stored locally; (c) our integrity checking ensures any corruption must be detected; (d) our system adopts heterogeneity, the design is based on standard interface of storage service providers; (e) performance degradation due to service composition is negligible, and for most of the cases, it actually outperforms the ordinary usage latency with Amazon S3.

References

[1] R. C. Jammalamadaka, R. Gamboni, S. Mehrotra, K. E. Seamons, N. Venkatasubramanian. A middleware approach for building secure network drives over untrusted internet data storage. In Proc. ACM conference on Extending database technology: Advances in database technology, pp. 710-714, 2008.

[2] R. C. Jammalamadaka, R. Gamboni, S. Mehrotra, K. E. Seamons, and N. Venkatasubramanian. gVault: A gmail based cryptographic network file system. In Proc. Working Conference on Data and Applications Security, pp. 161-176, 2007.

[3] http://JungleDisk.com

[4] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. In ACM Trans. On Computer Systems, pp. 1-24, 2002.

[5] E. J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In Proc. Network and Distributed System Security, pp. 131-145, 2003.

[6] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus – scalable secure file sharing on untrusted storage. In Proc. USENIX Conference on File and Storage Technologies, pp. 29-42, 2003

[7] K. Fu. Group sharing and random access in cryptographic storage file system. Master's thesis, MIT, 1999 June.

[8] G. Agarwal, M. Bawa, P. Ganesan, H. G. Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, Y. Xu. Two Can Keep a Secret: A Distributed Architecture for Secure Database Services. In Proc. Biennial Conference on Innovative Data Systems Research, pp. 186-199, 2005.

[9] V. Kher, Y. Kim. Building trust in storage outsourcing: secure Accoutning of Utility Storage. In Proc. IEEE International Symposium on Reliable Distributed Systems, pp 55-64, 2007.

[10] A. Heitzmann, B. palazzo, C. Papamanthou, R. Tamassia. Efficient integrity checking of untrusted network storage. In Proc. ACM workshop on Storage security and survivability, pp 43-54, 2008.

[11] http://ws.apache.org/rampart/

[12] http://ws.apache.org/wss4j/

[13] AES, J. Daemen and V. Rijmen, The Design of Rijndael: AES - The Advanced Encryption Standard. Springer-Verlag, 2002.

[14] SHA-1,http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf

[15] http://www.aws.amazon.com/s3

[16] http://aws.amazon.com/ec2

[17] http://developer.amazonwebservices.com/connect/entry.jspa?externalID=132

[18] http://www.ncipher.com/

[19] http://www.gogrid.com/

[20] http://www.viksoe.dk/code/gmail.htm

[21] G. Dhillon and S. Moores. 2001. Computer crimes: theorizing about the enemy within. In Computers & Security, volume 20, number 8, pp. 715-723, 2001 December.

[22] http://www.maluke.com/software/s3-backup

[23] R. Perlman. Ephemerizer: Making Data Disappear, Sun Microsystems Whitepaper, 2005 February
http://research.sun.com/techrep/2005/smli_tr-2005-140.pdf

[24] V. Kher and Y. Kim. Securing distributed storage: challenges, techniques, and systems. In Proc. ACM Workshop on Storage Security and Survivability, pp. 9-24, 2005.

[25] A. Singh and L. Liu, SHAROES: A Data Sharing Platform for Outsourced Enterprise Storage Environments. In Proc. International Conference on Data Engineering, pp. 993-1002, 2008.

[26] http://www.cl.cam.ac.uk/~rnc1/brute.html

[27] M. R. Palankar, A. Iamnitchi, M. Ripeanu, S. Garfinkel. Amazon S3 for science grids: a viable solution. In Proc. Workshop on Data-aware distributed Computing, pp. 55-64, 2008.

[28] F. Wang, R. Uppalli, C. Killian. Analysis of techniques for building intrusion tolerant server systems. In Proc. Millitary Communications Conference, pp. 729-734, 2003, Vol. 2.

[29] W. Zhao. Towards practical intrusion tolerant systems: a blueprint. In Proc. Annual workshop on Cyber security and information intelligence challenges ahead, article No. 19, 2008.