Available online at www.isr-publications.com/jmcs
J. Math. Computer Sci., 17 (2017), 488–505
Research Article

Online: ISSN 2008-949x

Journal of Mathematics and Computer Science

Journal Homepage: www.tjmcs.com - www.isr-publications.com/jmcs

Distributed security storage model for large-scale data

Ming Zhang[a,b,*], Wei Chen[a], Yunpeng Cao[b]

[a]School of Information Engineering, Wuhan University of Technology, Wuhan 430070, China.
[b]School of Information Science and Engineering, Linyi University, Linyi 276000, China.

Abstract

With the development of large-scale data, the increasingly users need to store the data in the distributed storage system due to the fact that the signal computer can not hold the massive data. However, the users can not control the data access rules. So the transparent security management of Large-scale data in distributed networks is a challenge. To solve this issue, a distributed security storage model is proposed. This security storage model can deal with the high concurrency and the complexity of large-scale data management in the distributed environment. The detailed designed of the transparent security storage system is provided based on the security storage model. This system allows the users manage their data and provides confidentiality protection, integrity protection, and access permission control. Experiments exhibit that the distributed storage model can improve the data security with I/O performance loss less than 5%.

Keywords: Distributed storage system, transparent encryption, confidentiality, integrity control model.
2010 MSC: 93A15.

1. Introduction

The emergence of cloud computing technology has brought many new developments for storage system; many companies began to build their own cloud storage platforms. At present the main cloud storage platforms used widely are Amazon Simple Storage Service, RackSpace Openstack and Microsoft Azure. Cloud storage platform has spawned many cloud storage application systems [20]. At the same time, the relevant new technologies and research results are also emerging [15–17]. With the development of cloud storage technology, more and more users or services started to use cloud storage environment to store data. Cloud storage environment is generally used by payment and bring a lot of convenience to users: no first-phase investment, saving management cost, good scalability and higher storage resource utilization rate. However, cloud storage environment makes data owner lose complete control on data, data security facing a series of threats [8]. For example, in cloud storage environment data is usually stored in plain text, lacking of integrity protection, reliable user authentication and access control mechanism. If important sensitive data is stored in the cloud storage environment, with the increase of users, the problem will become more serious.

*Corresponding author
Email address: zhangming@lyu.edu.cn (Ming Zhang)

doi:10.22436/jmcs.017.04.05

Received 2017-04-25

Currently data encryption is the mainstream method for protecting data privacy [9, 13]. Most of the cloud storage service providers require users to trust their storage servers and system administrators, part of service providers claim that they provide good security mechanism to ensure the security of user data. However, in 2010 data leak investigation report, Verizon pointed out that 49% of data leakage is caused by internal staff and privilege abuse accounted for a large part of data leakage, 48% of data leakage is caused due to users' malicious abusing the privilege of accessing information. Therefore, it is difficult for cloud storage users to fully trust service providers.

In some storage systems data access control is performed by data owner. If other users want to access data, they should contact with data owner. This reduces security threats in a certain extent. But this scheme brings new problems: firstly, data owners need to provide more complex data management services, and even online services; secondly, when the number of users is large, and there are many co-sharing users, the management is more difficult.

To solve above problems, this paper proposes a multi-user shared cloud storage system. In this system, data owners store their data in untrusted cloud storage service providers; other users access the data in untrusted network environment according to their assigned permissions. It is supposed that during data storage and access cloud storage service providers and malicious network users may probe or even tamper with it, users may try to operate it beyond their legal privileges. Based on this assumption, this paper presents a transparent secure cloud storage system framework. It allows users to protect data security and data access control to be efficient and reliable in untrusted cloud storage service provider and unreliable network environment, meanwhile ensuring that users cannot access beyond their privileges. According to this framework, this paper implements a secure cloud storage system prototype–TSCSS. TSCSS is a transparent secure cloud storage system with stacked encryption file system, which can be transparently deployed on file systems with POSIX standard interface, and it is not required to make changes to existing file systems. TSCSS is independent of cloud storage service providers, exists with the identity of third party, provides data confidentiality, integrity protection and access control service for users, and eliminates users' worry on data security.

Section 2 of this paper introduces system design principle and target; Section 3 introduces key technologies and system design and implementation scheme; Section 4 describes system performance test results and analysis; Section 5 analyzes and compares related works and Section 6 makes summary.

2. Design principles and objectives

(1) Independent of underlying file system. TSCSS is designed to provide security mechanism for existing cloud storage systems. It is independent of underlying file system to ensure that any changes to underlying file system are not required in using TSCSS.

(2) File sharing and access control. TSCSS provides secure and easy-to-use file level sharing and access control mechanism for users. File owner can specify who and how to access a data file.

(3) End-to-end confidentiality and integrity protection. TSCSS ensures that only authorized users can access data plaintext, illegal users and administrators of underlying file system are unable to obtain data plaintext. Malicious tampering with data can be found to ensure that the information gotten by users is correct.

(4) Key management. Users do not need to store any key locally in using TSCSS. That is to say, the TSCSS key management mechanism is transparent to users to increase the ease of use and security of the system.

(5) Key distribution. TSCSS needs a reasonable and efficient key distribution mechanism to ensure that legal users can obtain the key of the files they want to access.

(6) Lazy revocation. In TSCSS lazy revocation mechanism [5] is introduced to reduce the performance overhead caused by security mechanism. When privilege revocation occurs, TSCSS does not re-encrypt file immediately, but re-encrypt the modified content until the file is modified.

(7) Performance. Apart from PKI identity authentication system, all the encryption and decryption in TSCSS use symmetric keys. TSCSS also needs to use caching mechanism to avoid reduplicative computational and I/O overhead. In addition, TSCSS minimizes the consumption of disk space and network bandwidth caused by security mechanisms.

3. System design and implementation

3.1. Architecture design

There are many terms in this paper. For convenience of narration, the English abbreviations and meanings of used terms are sorted, as shown in Table 1.

Table 1: Glossary of terms.

Abbreviations	Meanings
SS	Storage Server
CS	Control Server
CSEK	Control Server Encryption Key
CSSK	Control Server Signature Key
ACL	Access Control List
EA	Encryption Algorithm
EM	Encryption Mode
KL	Key Lock
SL	Signature Lock
ACB	Access Control Block
ACB-HMAC	ACB Hash-based Message Authentication Code
RHi	ith Root Hash in root hash list

TSCSS consists of three parts: SS (storage server), client (client), and CS (control server), as shown in Figure 1.

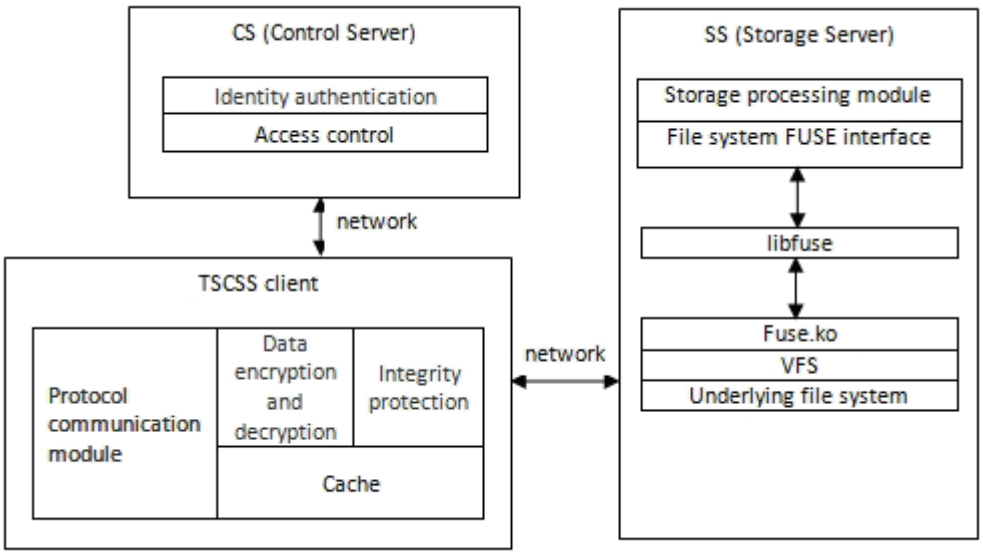

Figure 1: TSCSS architecture.

Storage server stores data files. A file is divided into two files stored in storage server and they are called data file (d-file) and metadata file (md-file), respectively. File cipher text is stored in data file, and

file security information such as access control block (ACB), root hash list (RHL), SBT tree, etc. is stored in metadata file. The specific content will be narrated later in detail.

Control server is the root trusted by the whole system. CS logic is very simple. It is responsible for users' identity authentication, processing users' file access request, distributing related keys to legitimate users and so on. CS only needs to store two symmetric keys CSEK and CSSK. The involved operation is only a small amount of encryption and decryption, calculating MAC, etc. This simple design can bring the following advantages.

(1) Low overhead. Because calculation is simple and there is no cost of disk I/O, CS can easily respond to multiple requests at the same time.
(2) The system is more reliable and the availability is stronger. Apart from two symmetric keys, CS has no other information. So if the CS crashes, another server with the same CSEK and CSSK can replace it immediately, without complex data and state recovery to maintain consistency.
(3) The extendibility is strong. Simple logic makes CS easily be extended to CS cluster, eliminating performance bottlenecks.

The client is responsible for handling users' requests, performing various operations on files. At the same time, file encryption and decryption and data integrity check are done on the client. When necessary, the client will also need to communicate with server to get keys. All these are transparent to users. The client only stores users' identity certificates and does not store additional information. This increases the usability and security of system. In technology selection, TSCSS is a user mode file system based on FUSE, so it can be installed on any file systems which provide standard POSIX interface and provide security functions for users. This also makes TSCSS be completely independent of underlying file system, and the scope of application is wider.

3.2. Symmetric key hierarchy

Key management has two critical issues: how to reduce the number of keys needed to maintain and how to handle key updating when privilege revocation occurs. In TSCSS, key is organized with three levels: file key, metadata file and control server, as shown in Figure 2.

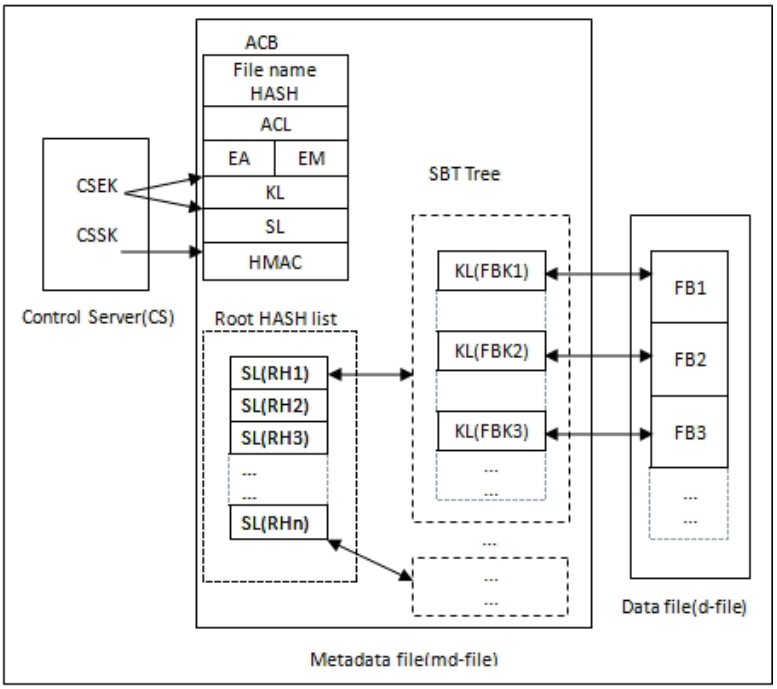

Figure 2: Key hierarchy management.

The first layer is file key. In order to deal with large file more efficiently and securely, TSCSS processes file data by block. Each file is divided into some file blocks (FB), each file block FBi uses a separate symmetric key FBKi for encryption and decryption, FBKi is calculated as follows: FBKi=Hash(FBi)||offseti||Ri

The symbol "||" means concatenating, Hash means calculating hash value, offseti is the offset of block i in file, and Ri is a random number. That is to say, the file block key is produced by concatenating its plaintext hash value, its offset and a random number. This method can bring the following benefits: (1) To protect data integrity, it is necessary to calculate hash. Using plaintext hash value as key can make this information to be reused, greatly saving key storage space; (2) Concatenating hash value, offset and random number as the key can make file blocks with same content generate different cipher text, so the confidentiality is enhanced; (3) When file block content changes, the key changes. This is good for system security and privilege revocation (to be narrated later in detail).

The second layer of key hierarchy is metadata fie (md-file). In the access control block ACB of md-file, there is a key lock KL. All the file block key FBKi are encrypted with KL, and stored in the form of SBT Tree in metadata file. Only users getting KL can decrypt to get file block key and then decrypt data file to get plaintext. There is a signature key SL in ACB. Only users getting SL have the ability to modify file legally, i.e., have write privilege. This part of content will be narrated in detail in Section 3.3.

The third layer of key hierarchy is control server (CS). Two keys are stored in CS: CS encryption key (CSEK) and CS signature key (CSSK). These two keys are known by CS only. In the access control block ACB of md-file, key lock Kl and signature key SL are all encrypted with CSEK by CS. Only by communicating with CS users can get KL or SL. CSSK is used to calculate the HMAC value of ACB. Using HMAC value, CS can determine whether the integrity of ACB is destroyed. Other entities do not have the ability to modify ACB, because they cannot obtain CSSK.

Through such three-layer key management structure, numerous keys can be organized efficiently, while ensuring data privacy and integrity, improving the efficiency of key management, and are good for user identity authentication, access authorization and privilege revocation. This will be described in detail in Section 3.5.

3.3. Integrity protection

The main idea of data integrity verification in cloud storage is to store users' data by using tree or tree-like data structure, and combine this data structure with appropriate cryptography technology to make it have authentication ability, and form authentication dictionary. In this paper authentication dictionary is constructed based on size balanced tree (SBT), and data integrity verification scheme is designed on this authentication dictionary. Data integrity verification scheme based on SBT structure can better support the integrity verification of dynamic data set, avoiding periodic reconstruction caused by data update. Compared with the authentication structure based on AVL tree and Treap, the authentication dictionary based on SBT structure has better balance and higher data efficiency [22].

3.3.1. Node size balanced tree SBT

SBT is a kind of binary search tree, with the following properties:

(1) if left sub-tree is not empty, the values of all nodes on left sub-tree are less than the value of root node;
(2) if right sub-tree is not empty, the values of all nodes on right sub-tree are greater than the value of root node;
(3) Left and right sub-trees are binary search trees, respectively.

SBT maintains balance by sub-tree size (number of nodes). This design structure can effectively and dynamically maintain the binary search tree, and even in worst case good running speed can be kept. When tree nodes become unbalanced, rotation operation is needed to maintain balance. For each node t in SBT, it must meet the following natures:

(1) s[right[t]]$\geqslant$ s[left[left[t]]], s[right[left[t]]];
(2) s[left[t]]$\geqslant$ s[right[right[t]]], s[left[right[t]]].

Among them, left[t] denotes t's left child node; right[t] denotes t's right child node; s[t] is the size of the sub-tree whose root is t, i.e., the number of nodes in this tree.

3.3.2. Data operation on SBT

In cloud storage system, data is searched, inserted, updated and deleted frequently. Search is the most essential operation. Whether it is inserted, updated or deleted, search is needed first to locate target node. The search process of SBT tree is as follows:

(1) if the tree is an empty tree, then search fails;
(2) if the target value is equal to the value of root node, search is successful;
(3) if the target value is less than the value of root node, search left sub-tree recursively;
(4) if the target value is greater than the value of root node, search right sub-tree recursively.

Size balance tree is implemented with linked list. During insertion and removal, it is not necessary to move node location; it is enough to change the pointer to node. So it's time complexity is consistent with that of search operation.

When inserting a data element into the authentication data structure based on SBT, first find the location of the inserted node, then execute insert. Because the inserted node is inevitable the leaf node of SBT, if the insert operation does not cause rotation and adjustment operation, only the hash values of all the nodes in the paths from root node to new leaf node are needed to be recalculated; if the insert operation causes rotation and adjustment operation, the hash values of all the nodes affected after rotation and adjustment are needed to be recalculated. The affected nodes include all nodes in the rotated sub-tree and all the nodes on the paths from the sub-tree root node to the root node of SBT.

When removing a data element in a SBT based on authentication data structure, there are three cases:

(1) if x, the node to be deleted is a leaf node, it can be deleted directly. The nodes whose information needs to be modified are all the nodes on the path from root node to the parent node of x;
(2) if x, the node to be deleted is a single branch node, i.e., it only has a left sub-tree or a right sub-tree, connect the parent node of x with the child node of x, and then delete x. The nodes whose information needs to be modified are all the nodes on the path from root node to the parent node of x;
(3) if x, the node to be deleted has left sub-tree and right sub-tree, then it is needed to change the tree structure. Delete can be completed by two ways: ① replace x with y, the node which has maximum value in the left sub-tree of x, and then delete y; ② replace x with z, the node which has minimum value in the right sub-tree of x, and then delete z. The nodes whose information needs to be modified are all the nodes on the path from root node to the parent node of y or z.

In above three cases, only the hash values of the nodes on the path from the root node of SBT to deleted node (or its child nodes) need to be calculated. If rotation and adjustment are caused, the update procedure of node's hash value is consistent with insertion procedure.

3.3.3. SBT analysis

(1) For query and verification operations, SBT authentication dictionary has the same algorithm complexity as Merkle tree authentication dictionary, they are all $O(\log(n))$. They all use hash functions instead of other complex operations, so the calculation is simpler.
(2) SBT has good balance, and can better control tree height, so that it tends to be close to complete binary tree. The time complexity of data update and query on binary tree is related to tree height. SBT tree has higher efficiency than non-balanced tree, and data operation time can be kept in $O(\log(n))$. SBT authentication dictionary can also completely avoid periodic reconstruction of authentication structure.

Therefore, for data set which has large number of members and needs to be updated frequently, the scheme based on SBT has obvious advantages [22].

3.3.4. The implementation of TSCSS integrity protection

By calculating the plaintext hash value of each file block, TSCSS ensures its integrity (as stated above, the hash value is also part of the encryption and decryption key FBK). Furthermore, these hash values are organized to form one or many SBT trees. File block hash value and joint hash value are stored in SBT tree nodes to ensure the integrity of nodes. Every node v of T, storage tree of all nodes based on SBT must store two entities:

(1) the Hash value Hv=Hash(FB), block offset, random number, block number, etc. of the corresponding file block of the node;
(2) joint hash value Hsum(v). Its calculation method is as follows:
   - if node v is leaf node, then Hsum(v)= Hv;
   - if node v is not leaf node, and only has left child node, then Hsum(v)=Hash(Hv, Hsum(left[v])), where Hsum(left[v]) represents the joint hash value of the left child node left[v] of node V;
   - if node v is not leaf node, and only has right child node, then Hsum(v)=Hash(Hv, Hsum(right[v])), where Hsum(right[v]) represents the joint hash value of the right child node right[v] of node V;
   - if node v is not leaf node, and has both left and right node, then Hsum(v)=Hash(Hv, Hsum(left[v]), Hsum(right[v])).

Global hash value Hash(T) is the joint hash value of the root node of T, storage tree of full nodes. Its structure is shown as Figure 3.

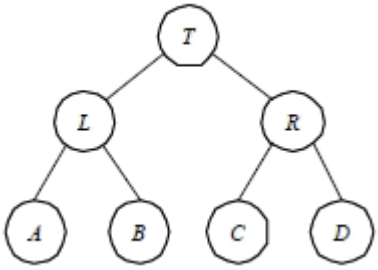

Figure 3: SBT Tree in TSCSS.

The content of each node in Figure 3 is H ||Hsum(v). The content of H is shown in Table 2.

Table 2: The content of H.

File block Hash: Hv=Hash(FB)	Block offset	Random number	Block serial number	Block existence tag	Node existence tag
20 Bytes	8 Bytes	4 Bytes	2 Bytes	1 Byte	1 Byte

In SBT Tree of TSCSS, Hv of node i is actually the hash value of file block i. Hsum(v) is the hash value calculated after concatenating the content of node i and all its child nodes. Moreover, "block existence tag" is also used to identify whether file block i exists (there may be empty holes in file) and "node existence tag" is used to identify whether node i in SBT tree exists (if Hv or Hsum(v) exists, it is considered that node i exists). These two tags can help to identify file holes and improve the efficiency of integrity verification (nonexistent SBT tree node does not participate in calculation).

Finally, SBT Tree's root is encrypted with signature lock SL and stored in the root hash linked list in md-file. In order to provide possibility of concurrent writing, the integrity of a file is ensured with many SBT trees and a linked list linking the root hashes of these SBT trees.

In TSCSS, the integrity of file block i is ensured with Hv, the integrity of the sub-tree whose root is node i is ensured with Hsum(v), so the root of SBT tree ensures the integrity of whole SBT tree. Because

only legitimate users with file write privilege can obtain signature lock SL (this will be described in detail in Section 3.5, so the root hash stored in md-file ensures that once an illegal user tampers with file content, he/she can be found. In general, the root hash encrypted with SL ensures the integrity of SBT tree, and SBT tree stores the plaintext hash of all file blocks, and then the integrity of the entire file data is protected.

The benefit of using SBT tree to protect file integrity is obvious. When the content of one or some blocks of the file is legally modified, only the Hv of these blocks and $\mathrm{Hsum}(v)$ of the nodes passed along the paths from these blocks to root nodes needed to be recalculated. Finally the updated root hash is reencrypted with SL and stored in md-file. The time complexity is $O(\log(n))$. If SBT is not used, a hash value is calculated by concatenating all FBKs together to guarantee the integrity of all FBKs, then even if only a block is modified, it is required to concatenate all FBKs and calculate hash value again, and such cost is difficult to accept for large files.

In addition, using the hash value of file block plaintext instead of cipher text for integrity check, firstly, the plaintext hash value is reused as key to save space; secondly, this can ensure that the information users get is indeed what they want (because only guaranteeing the cipher text integrity is not enough. If the integrity protection mechanism is not perfect, the key may be tampered with, the decrypted plaintext is wrong).

In TSCSS, before every read or write to file, the integrity of the accessed content is checked first. Firstly, the integrity of root hash of SBT tree is checked, then the integrity of SBT tree nodes involved in access and the nodes on the paths from these nodes to root node is checked, this can ensure the integrity of Hv, i.e., FBKs of these nodes. Checking the integrity of SBT tree node is recalculating its $\mathrm{Hsum}(v)$ value according to its child nodes, comparing with the $\mathrm{Hsum}(v)$ value stored in md-file, and finally, checking the integrity of each file block (calculating the hash value of decrypted plaintext, and comparing with the plaintext hash in FBK).

3.4. Lazy privilege revocation

In cloud storage system with large number of users, permission revocation often occurs [9]. Most cloud storage systems mainly rely on storage server to manage file access privileges; the overhead of privilege revocation is little. But this requires users to completely trust storage server; it is not safe for users' data. Relatively, in encrypted storage system, the performance overhead caused by permission revocation is much greater, because in order to avoid users whose privileges are revoked from continuing to access files, it is needed to regenerate related keys, re-encrypt and distribute new keys to users who still have access privileges. File re-encryption will seriously affect system performance, and cause that the file cannot be accessed in this process.

In order to reduce the extra cost caused by privilege revocation, TSCSS uses lazy revocation technology [5]. For each file block, after privilege revocation, only when its content is changed, it is re-encrypted. This largely reduces the effect of privilege revocation on performance. For md-file, once privilege revocation occurs, it is required to regenerate the key lock KL and signature lock SL for the file, and then use the new KL to encrypt all file block key FBK, and use new SL to encrypt all the root hash RH. Since TSCSS only re-encrypts all the FBK and RH, and does not re-encrypt file block, the data volume is reduced much, the performance overhead is also greatly reduced. This also reflects the superiority of the key hierarchy management mechanism. In addition, when the content of file block is modified, its hash value will be changed, so the file block key FBK will be changed automatically. So TSCSS is not required to use a complex method and additional space to record information such as the file's history keys and history states etc. as in existing lazy privilege revocation mechanisms [3, 9]. This further saves the time and space. In addition, some existing lazy privilege revocation mechanisms need to re-encrypt the entire file, but TSCSS is accurate to file block level. This also improves performance to some extent.

3.5. Access protocol and process

In TSCSS file access protocol, the communication between client and control server uses SSL encryption, this can effectively solve the problem that network is not reliable. In addition, control server CS

only needs to maintain two symmetric keys, the client only needs to maintain user's identity certificate and does not need to maintain any key. This is not only simple and efficient, but also reduces the risk of key leakage, and it is much safer. TSCSS can do this, thanks to its simple, safe, and efficient file access protocol. The protocol is described in detail below.

Identity authentication. TSCSS uses x509 standard to realize identity authentication. Each user must apply for a certificate from CA to identify his/her identity. When client communicates with control server CS, it will try to establish a SSL connection with CS. When establishing connection, the client will send user's identity certificate to CS. After the certificate passes CS verification, CS will also send its own certificate to the client. Then the SSL connection is confirmed. That is to say, in control server CS, each connection is bounded with user certificate, i.e., every communication can be identified by its identity.

The process of creating a file is as follows:

(1) Client initializes a file creating request including file name, encryption algorithm, encryption mode, access control list, and other informations, then sends the request to control server CS.
(2) After receiving the request, CS generates key lock KL and signature lock SL for the file, uses control server encryption key CSEK to encrypt them, creates ACB, then uses control server signature key CSSK to calculate the HMAC value of ACB, and finally initializes root hash linked list and returns the generated ACB to client.
(3) Client creates two files on storage server according to ACB returned by CS: data file d-file and metadata file md-file.

The process of reading a file is as follows:

(1) Client finds metadata file md-file from storage server, reads the content of ACB from it, and sends the read request and ACB to control server CS.
(2) CS first checks the integrity of ACB, then determines whether the user has read permissions according to ACL access control list, then uses CSEK to decrypt KL and SL and uses SL to decrypt the root hash linked list (to verify the integrity of SBT tree), and finally returns KL and root hash linked list to client.
(3) Client uses KL to decrypt the block key FBK of file blocks to be accessed, verifies the integrity of SBT tree and its root hash, then reads out the relevant file blocks from d-file and uses corresponding FBK to decrypt to obtain plaintext, and verifies the integrity of file block by calculating plaintext hash.

The process of writing file is as follows:

(1) Client finds metadata file md-file from storage server, reads out the content of ACB, and sends the write request and ACB to control server CS.
(2) CS first checks the integrity of ACB, then determines whether the user has write permissions according ACL access control list, then uses CSEK to decrypt KL and SL and uses SL to decrypt root hash linked list, and finally returns KL, SL, and root hash linked list to client.
(3) Client uses KL to decrypt the block key FBK of file blocks to be accessed, verifies the integrity of SBT tree and its root hash, then calculates hash and new FBK for the data to be written at the file block granularity, uses the new FBK to encrypt data, writes them to d-file. Meanwhile it is also needed to update SBT tree, recalculate root hash and encrypt with SL, write them to md-file.

The process of sharing files is as follows:

(1) File owner finds md-file from storage server, reads out ACB, sends file sharing request and ACB to CS. The file sharing request includes which users will be added into the access control list (ACL), what permissions each user has and other information.
(2) CS first checks the integrity of ACB, and checks whether the user is file owner, and then inserts the access control entries in the client request into ACL, and recalculates the HMAC value of ACB with CSSK, finally returns the updated ACB to client.
(3) Client writes new ACB to md-file.

The process of privilege revocation is as follows:

(1) Client finds md-file from storage server, reads out ACB and sends the privilege revocation request and ACB to CS. Privilege revocation request includes which users permissions will be revoked from, which permissions each user should have after privilege revocation (e.g. downgraded from read-write to read-only permission) and other information.
(2) CS first checks the integrity of ACB, and checks whether the user is file owner, then updates access control list (ACL) according to the client request and regenerates key lock KL and signature lock SL, and encrypts new KL and SL with CSEK, writes into ACB. Finally recalculates the HMAC value of ACB with control server signature key CSSK, and returns updated ACB, old KL and SL and new KL and SL to client.
(3) Client decrypts all file block key FBKs with old KL, encrypts them with new KL and writes into md-file; then decrypts all root hashes with old SL, encrypts them with new SL and writes to md-file. Finally, the updated ACB is written into md-file.

3.6. Ensuring correctness and performance tuning

TSCSS uses lock mechanism (including file read/write lock and thread mutual exclusion lock) to achieve read/write mutual exclusion and ensure file data consistency. TSCSS supports multi-threaded concurrent reading same file.

In order to improve the performance of TSCSS, this paper uses caching mechanism to reduce the overhead of encryption, decryption, and integrity check. For example, the accessed plaintext of first twelve layers of nodes of SBT tree is cached in memory, they are not re-encrypted and written back to md-file until it is necessary (e.g. when file is closed). This can reduce the overhead caused by multiple I/O accesses and encryption and decryption on the first twelve layers of nodes of tree SBT in the process of integrity verification. Accordingly, a "whether the integrity is checked" tag can be set for cached SBT tree nodes to avoid duplicated integrity verification on cached SBT tree nodes.

If a user reads or writes a section of data repeatedly, each time of read or write operation, corresponding content is needed to be read out from encrypted data file, integrity check is performed, then reads the related file data encryption key (cipher text form) from metadata, then uses key lock KL to decrypt these keys, and uses these keys to decrypt data file to get plaintext data. Write operation is similar. To improve performance, by using a cache system in TSCSS, recently accessed plaintext of file blocks is cached. This can omit above steps, reducing the overhead of unnecessary I/O operations, integrity verification and encryption and decryption.

In concrete implementation, this paper uses RadixTree [3] to organize file block cache. This can efficiently search, insert, and delete cached blocks. At the same time, LRU linked list is used to manage buffer pool, improving the cache hit rate. Perfect lock mechanism is also used to ensure the correctness of cache system.

4. Function and performance test

In this paper a series of tests were done on TSCSS functions and performance, including verifying the secure functions TSCSS can provide in untrusted network and storage environment, measuring the overhead of encryption and decryption, integrity check, file sharing and privilege revocation etc., and using Bonnie++ and IOzone to test the overall performance of TSCSS.

4.1. Function test

Three servers were used to test TSCSS on functions. One server was used as the CS of TSCSS and the server of NFSv4, the other two were used as the clients of NFSv4 and TSCSS. TSCSS was mounted on NFSv4 with the identity of users A and B, respectively. The content of test is shown as Table 3.

Table 3: TSCSS function test.

Test content	Test results
Data privacy protection: bypass TSCSS to view user's file content.	File content is garbled code after encryption, file plaintext cannot be obtained.
Data integrity protection: bypass TSCSS to tamper with data files or metadata files.	TSCSS reports the warning that the file integrity has been damaged.
Permission management: test the results of read/write operation performed by file accessor before being granted read/write permission, after being granted read/write permission and after being revoked permissions.	Read/write operation on file fails before being granted read/write permission and after being revoked permission; the read/write operation on file is successful after read/write permission is granted.

4.2. Performance test environment and parameter selection

The hardware environment of performance test is two servers with same configuration. The server is Lenovo Erazer x700, Intel Core i7-3930k CPU, 3.2GHz, 16GB memory. The two servers are connected with LAN, one is used as CS, the other is used as client. The software environment is Ubuntu Linux 14.04 LTS, fuse2.8.5, openssl1.0.1h.

In the selection of security mechanism, SHA-1 function is used to calculate hash. HMAC, the MAC algorithm based on SHA-1 is used to calculate MAC, AES-256 series function is the default encryption and decryption function, cfb is the default encryption mode (user can choose encryption algorithm and encryption mode by configuration file), x509 series function is used to realize identity verification.

TSCSS file block size is set to 64KB, because the test scene mainly faces large file application. If TSCSS is needed to be used in cloud storage environment of small file, it can be easy to adjust the size of file block.

In the structure of SBT tree, the height of SBT tree is not more than 16 layers. This selection is to reduce the number of I/O operations at integrity check of SBT tree. The less the layers of SBT tree is, the less the number of needed disk I/O operations is. In fact, specific implementation cache mechanism is used to ensure that each time SBT tree integrity is checked there is no more than one I/O operation. In addition, a single file supports up to 1024 SBT trees, so the size of a file TSCSS can support is up to about 4TB.

4.3. Encryption and decryption overhead

First create a file, then open the file with read/write mode, each time write 500MB content with the granularity of 64KB, and then read out the 500MB content, finally close the file. Table 4 lists the encryption and decryption overhead of various file operations in test.

From Table 4 we can see that the majority of overhead of encryption and decryption is in the data file encryption and decryption. In addition, "calculating SBT tree node value" in write operation and "verifying file block hash" in read operation also consume a part of time. These are two parts belonging to overhead of integrity check. Above overhead is proportional to the size of the read/written content. It also can be seen that because the operation that control server CS participates in is simple in logic, time-consuming is very short.

In order to test the overhead of file sharing and privilege revocation, the following tests are carried out: the owner of file A first gives 500 different users read-only privilege to file A, then upgrades the 500 users' privilege to read/write permission, and finally revokes all their privilege. The time cost of each step (the consumed time from user's beginning operation to the completion of operation) is recorded, as shown in Table 5.

Table 4: Encryption and decryption overhead of file operations.

File operation	Encryption and decryption operation	Total overhead/ms	Executor Executor	Operation frequency
Create	Generate FSK, LBK	0. 009	CS	File level
	Encrypt FSK, LBK	0. 006	CS	File level
	Calculate ACB HMAC	0.034	CS	File level
Open	Verify ACB HMAC	0. 035	CS	File level
	Decrypt FSK, LBK	0. 009	CS	File level
	Decrypt root hash linked list	0.009	CS	File level
Close	Encrypt SBT tree	1.717	Writer	File level
	Encrypt root hash linked list	0. 004	Writer	File level
Write	Verify SBT tree node value	0. 220	Writer	File block level
	Verify SBT tree root hash	0. 463	Writer	SBT tree level
	Calculate SBT tree node value	1789.539	Writer	File block level
	Calculate SBT tree root hash	11.617	Writer	SBT tree level
	Encrypt file block	6997.840	Writer	File block level
Read	Verify SBT tree node value	1.348	Reader	File block level
	Verify SBT tree root hash	1.611	Reader	SBT tree level
	Decrypt file block	6085.778	Reader	File block level
	Verify file block hash	1437.307	Reader	File block level

Table 5: The time overhead of TSCSS privilege operation.

Operation description	Time overhead/ms
File read only and share	2.487
Privilege upgrade	2.505
Privilege revocation	3.668

It can be seen from Table 5 that because file sharing or permission upgrade only simply modifies file access control list ACL and recalculates the HMAC of ACB, the speed is very fast. Privilege revocation needs to regenerate KL and SL and re-encrypt existing file block keys and root hash linked list, so time-consuming is relatively longer, but because the content of file block is not reencrypted immediately, the speed is still very fast.

4.4. File read and write test

4.4.1. Big file read and write test

In the same test environment as Section 4.2, Bonnie++1.03e is used to test Ext3 and TSCSS. First Bonnie++ is run on local file system Ext3 of client to test performance, then TSCSS is deployed on Ext3 and Bonnie++ is run again to test the performance of TSCSS. The test results are shown in Figure 4.

As can be seen from Figure 4, compared with Ext3, TSCSS's performance of read and write decreases by 31.3% and 17.4%, respectively, the main reason is that file content encryption and decryption introduces overhead. The read and write performance of byte granularity decreases by 53.6% and 55.6%, respectively. The more important reason that the read and write performance of byte granularity decreases is that each time read and write operation occurs, TSCSS needs to check integrity, including the integrity of involved file block in access and the integrity of SBT tree. Because in implementation TSCSS selects 64KB as the file block size, even if accessing a byte in file block, it is required to calculate the hash of entire file block to check integrity, so read and write of byte granularity will introduce more overhead of integrity check.

Then TSCSS is tested in NFSv4 environment, the same test environment is still used, Bonnie++ is used to test. There are two servers. One is used as control server CS and also as NFSv4 server to provide

storage service, the other is NFSv4 client and also TSCSS client (TSCSS is mounted on NFSv4). Bonnie++ is run to test on TSCSS mount point and NFSv4 mount point successively; the test result is as shown in Figure 5.

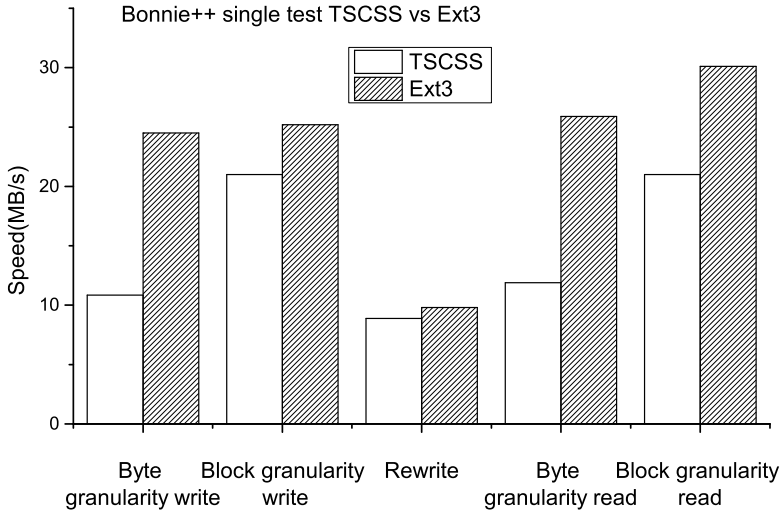

Figure 4: Bonnie++ single test TSCSS vs Ext3.

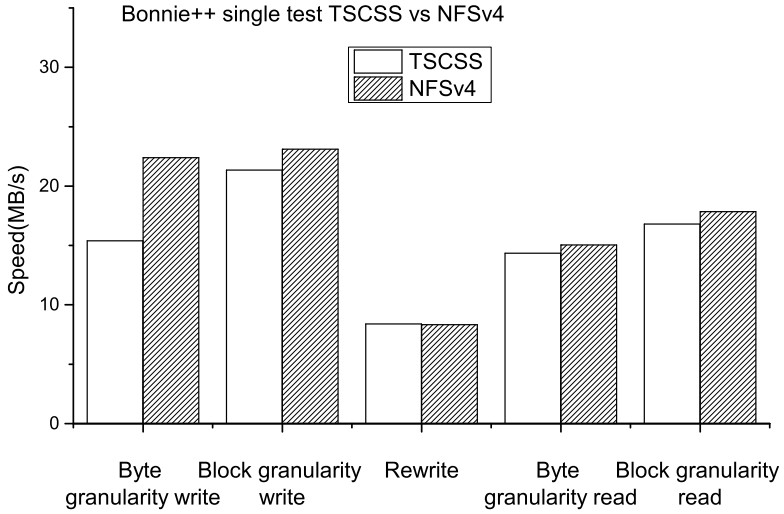

Figure 5: Bonnie++ single test TSCSS vs NFSv4.

By contrasting Figure 4 and Figure 5, it can be found that, compared with Ext3, the write performance of NFSv4 slightly drops by about 10%, but read performance drops by more than 40%. This is mainly caused by NFSv4's introducing network overhead and protocol overhead. The performance change of TSCSS is very small, and is only a slight decline, this is because that TSCSS's introducing the overhead of file content encryption and decryption, and integrity verification, etc. is the main reason of performance decline. Because these operations are computationally intensive, the performance of CPU will become bottleneck. So although NFSv4 has weakened the underlying storage I/O capabilities, but does not weaken to the extent to make it become bottleneck. That is to say, the performance of CPU is still the

bottleneck. So the performance of TSCSS only drops slightly. Thus, in cloud storage environment, with the increase of the number of users, it means the increase of the number of clients and the enhancement of computing capability, computing part will not be the bottleneck, the performance overhead of TSCSS will become slighter and slighter. In order to verify this idea, TSCSS is tested on cluster.

In cluster test, the hardware environment uses 5 servers of same configuration; the model is Lenovo Erazer x700, Intel Core i7-3930k CPU, 3.2GHz, 16GB memory. The software environment and configuration are the same as prior test. 16GB is selected as the size of tested file. Among the 5 servers, one is used as the NFSv4 server and an EXT3 partition of the local disk is exported to provide external storage service, one is used as control server and the rest 3 are used as clients. NFSv4 is mounted on them all and TSCSS is mounted on NFSv4.

This time IOzone3.347 is used to perform test, because IOzone has cluster test function. Firstly 3 clients perform IOzone cluster test on native NFSv4 mount points, then change to perform test on TSCSS mount points similarly. The obtained result is shown as in Figure 6.

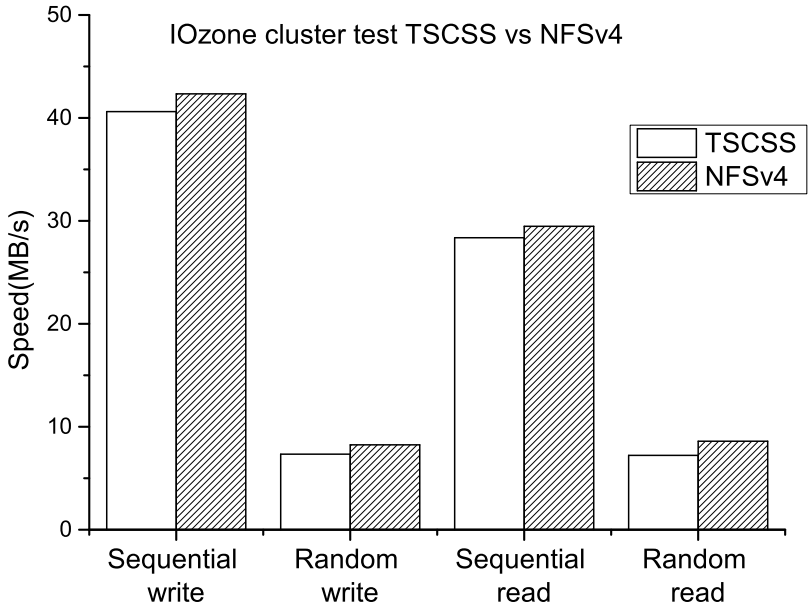

Figure 6: IOzone cluster test TSCSS vs NFSv4.

As can be seen from Figure 6, the aggregation access speed of TSCSS being mounted on NFSv4 reaches more than 95% of NFSv4, and reaches the limit of the performance of a single disk. This shows that when the number of clients increases, storage service becomes the bottleneck. The calculation cost introduced by TSCSS is not obvious. Experimental results show that in cloud storage environment TSCSS can be applied well.

4.4.2. Read and write test of massive small files

Aiming at existing operation needs for massive small files in actual environment, the performance comparison of operations under network environment for massive small files of TSCSS and NFS are tested respectively. The test selects 3 servers as file server of cloud storage, control server, and client, respectively, its hardware and software environment configuration is the same as above servers testing IOzone. In the process of testing NFS and TSCSS, the client performs create, write, and read operation for 1000 small files under corresponding mount points respectively, the size of each file is set to 512KB. The test result is shown as Figure 7.

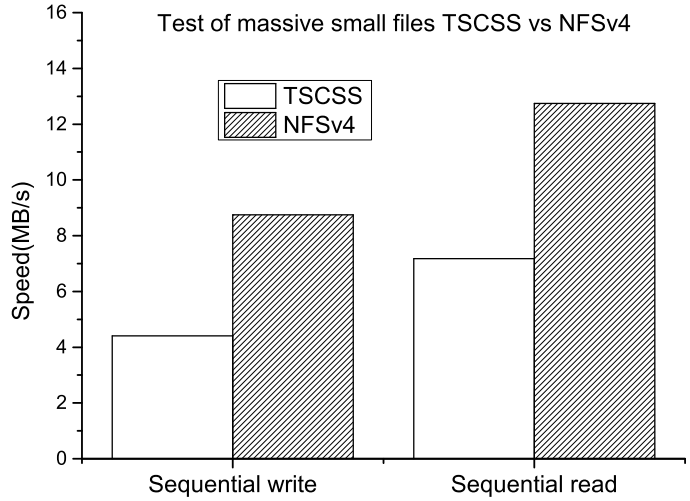

Figure 7: Test of massive small files TSCSS vs NFSv4.

As seen from Figure 7, compared with NFSv4, the read and write operation performance of small file of TSCSS system decreases by 50.7% and 43.1%, respectively. This is because while TSCSS accesses data file each time, it needs to access metadata file. When dealing with small files, the proportion of the time occupied by handling metadata is larger than that of dealing with large files, so the processing performance of small file in TSCSS is not as good as that of large file.

In addition, in this read and write test, the number of I/O is also compared. The test result is shown in Figure 8.

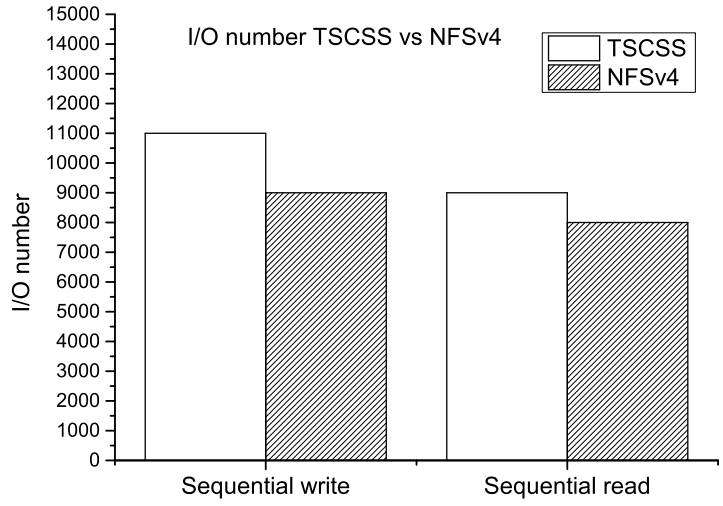

Figure 8: I/O number TSCSS vs NFSv4.

From Figure 8 it can be seen that the number of I/O increases by 22.2% and 12.5% respectively at two situations of sequential write and sequential read. Compared with the results in Figure 7, it

can be seen that the decline in performance is more serious than the increase of the number of I/O, because the decline of performance is not all caused by the rise of I/O number, the reason resulting in performance degradation also includes network transmission overhead, encryption and decryption, data integrity verification, etc..

In actual access, in order to optimize performance, TSCSS provides caching mechanism to reduce the number of I/O. This can improve the performance of small file read and write to a large extent. For example, when TSCSS system accesses a file once again, its contents have been in memory, so access speed will improve greatly.

5. Analysis and comparison between TSCSS and related works

CFS [2] is one of the earliest encrypted file systems. It is a virtual encrypted file system. Before data is written to disk, file name and file data are encrypted. CFS encrypts and decrypts files in the entire directory with one key. Its access control rule is sharing keys to other users. This determines that CFS only permits coarse granularity share on a machine, and does not distinguish between read permission and read/ write permission.

Cryptfs [21], Cepheus [5] and TCFS [4] are famous varieties of CFS. Cryptfs gives file group symmetric key to allow group file sharing. Cepheus introduces lock box to realize sharing management between user groups, relies on a trusted key server to store user group member information to realize identity authentication, also depends on storage server to realize access control. It is the first secure file system which proposed lazy privilege revocation. Cepheus has realized integrity protection. TCFS awards each user a master key to protect user's file key. The three file systems are all unable to distinguish between read sharing and read-write sharing.

Tahoe [18] is a distributed secure file system, including access control, encryption, integrity check and other functions. And it uses erasure code to achieve fault tolerance. It is deployed in a commercial backup service. Round-Trip Privacy with NFSv4 [14] is an improvement on NFSv4 which has modified the RPCSEC-GSS protocol in NFSv4 and made files on file server to be stored in cipher text. But RTP-NFSv4 does not perform integrity protection for files and its key mechanism is too simple. Farsite [1] is a secure file system, providing a centralized file server function, but in fact it is composed of multiple distributed untrusted computers. Farsite provides file availability and reliability by multi-copy mechanism, ensures the confidentiality of file contents by encryption, and ensures the integrity of file and directory data by a protocol which can prevent Byzantine mistake. The support of access permission control mechanism of Tahoe, RTP-NFSv4 and Farsite for frequent file share and permission revocation of large number of users is not good.

CryptosFS [12] and SNAD [11] use public and private key encryption system to achieve access control, and verify user's access permissions by file server, therefore they need to fully trust file server. CryptosFS users need to use asymmetric key to decrypt corresponding symmetric key from metadata file, and then use symmetric key to decrypt data file, so its key management mechanism is out of band.

NCryptfs [19] is a secure file system implemented in kernel state, it can support multi-user file sharing on a machine, but cannot support large-scale file sharing.

SiRiUS [7] is a stacked file system providing security mechanism for existing file systems. It uses a large number of asymmetric keys to perform permission control, and a specialized public and private key server is also needed. In SiRiUS, the file is encrypted entirely, integrity check is calculating hash for the entire file, when revoking privileges re-encryption is performed immediately, performance overhead is larger. Plutus [9] also uses public and private key encryption system, providing group share and lazy revocation, random access, file name encryption and other functions. It uses the key management mechanism of sharing file keys between users. When other users want to access files, they need to ask file owner for key. This mechanism requires file owner to be online in real time. Like SiRiUS, CRUST [6] is also a stacked file system, but it does not use public and private key encryption system, and symmetric key is used for all encryption and decryption. It relies on some public data structures to achieve distributed

key management, key rollback, permission revocation and so on. So with the increase of user number, information to be maintained will increase by square order, it is not suitable in the environment of large number of users. CRUST and Plutus all use Merkle tree [10] to ensure file integrity.

Compared with above works, TSCSS has obvious advantages. TSCSS is a secure cloud storage system with stacked encryption file system; it can be mounted on existing file systems. TSCSS has a set of its own permission control mechanisms; it does not depend on cloud storage server. Apart from x509 identity authentication, all the encryption and decryption use symmetric keys. Compared with public and private key encryption system, the overhead is less. TSCSS encrypts file by block, uses a hierarchical key management mechanism and in-band key distribution mechanism, users do not maintain any key information. This makes the key management safer and more efficient. TSCSS uses lazy permission revocation mechanism which is more efficient than existing implementations, without storing key historical information, saving time and space. TSCSS adopts improved SBT Hash tree and cache mechanism, making integrity check quicker and providing probability for concurrent write.

6. Conclusion

The secure cloud storage system architecture proposed in this paper makes users ensure data security in untrusted network environment and cloud storage environment. By introducing trusted control server, the dependence on cloud storage server is eliminated; it is suitable for more and more popular cloud storage application scenes. At the same time, because the logic is simple, the control server has higher performance, flexibility, and extendibility.

The specific realization of the secure cloud storage system architecture proposed in this paper-TSCSS provides security protection for the users of existing cloud storage systems, including privacy protection, integrity protection, file access control and so on. At the same time, TSCSS supports random access and concurrent access better. The results of TSCSS performance test show that the performance degradation caused by mounting TSCSS on NFSv4 cluster is less than 5%. Thus, TSCSS achieves powerful and easy-to-use data security protection with smaller and acceptable additional performance overhead.

Acknowledgment

This research is supported by Shandong Provincial Natural Science Foundation (No. ZR2017MF050, ZR2015FL014), the special project of Shandong Provincial Independent Innovation and Achievement Transformation (No. 2014ZZCX02702), Shandong Province Key Research and Development Project (No. 2016GGX109001), Shandong Provincial Major Science and Technology Innovation Project (No. 2017CXGC0701) and Shandong Provincial Technical Plan Project of Higher Education (No. J17KA049).

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, R. P. Wattenhofer, *FARSITE: Federated, available, and reliable storage for an incompletely trusted environment*, Proceedings of the 5th Symposium on Operating Systems Design and Implementation, San Francisco, USA, **36** (2002), 1–14. 5

[2] M. Blaze, *A cryptographic file system for UNIX*, 1st ACM Conference on Computer and Communications Security, ACM Press, (1993), 9–16. 5

[3] P. D. Bovet, M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management*, O'Reilly Media, Inc., (2005). 3.4, 3.6

[4] G. Cattaneo, L. Catuogno, A. Del Sorbo, P. Persiano, *The design and implementation of a transparent cryptographic file system for Unix*, Proc. USENIX Annual Technical Conference, Boston, USA, (2001), 199–212. 5

[5] K. E. Fu, *Group sharing and random access in cryptographic storage file systems*, Master's thesis, Massachusetts Institute of Technology, Tech. Rep., Boston, USA, (1999). 6, 3.4, 5

[6] E. Geron, A. Wool, *CRUST: cryptographic remote untrusted storage without public keys*, Int. J. Inf. Secur., **8** (2007), 357–377. 5

[7] E.-J. Goh, H. Shacham, N. Modadugu, D. Boneh, *SiRiUS: Securing remote untrusted storage*, Proceedings of the 10th Network and Distributed Systems Security Symposium, San Diego, USA, **3** (2003), 131–145. 5

[8] R. Hasan, S. Myagmar, A. J. Lee, W. Yurcik, *Toward a threat model for storage systems*, Toward a threat model for storage systems, FairFax, VA, USA, (2005), 94–102. 1
[9] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, K. Fu, *Plutus: Scalable secure file sharing on untrusted storage*, Proceedings of the 2nd USENIX File and StorageTechnologies, FairFax, VA, USA, **3** (2003), 29–42. 1, 3.4, 5
[10] R. C. Merkle, *A digital signature based on a conventional encryption function*, Conference on the Theory and Application of Cryptographic Techniques, Springer, Berlin, Heidelberg, Santa Barbara, USA, (1987), 369–378. 5
[11] E. L. Miler, D. D. Long, W. E. Freeman, B. Reed, *Strong security for network-attached storage*, Proceedings of the 1st USENIX File and Storage Technologies, Monterey, USA, (2002), 1–13. 5
[12] D. P. O'Shanahan, *CryptosFS: Fast cryptographic secure NFS*, Master's Thesis, The University of Dublin, Ireland, (2000). 5
[13] E. Riedel, M. Kalahala, R. Swaminathan, *A framework for evaluating storage system security*, Proceedings of the USENIX Conference on File and Storage Technology, Monterey, USA, **2** (2002), 15–30. 1
[14] A. Traeger, K. Thangavelu, E. Zadok, *Round-trip privacy with NFSv4*, Proceedings of the 2007 ACM workshop on Storage security and survivability, Alexandria, Virginia, USA, (2007), 1–6. 5
[15] L. Wang, B. Yang, A. Abraham, *Distilling middle-age cement hydration kinetics from observed data using phased hybrid evolution*, Soft Comput., **20** (2016), 3637–3656. 1
[16] L. Wang, B. Yang, Y.-H. Chen, X.-Q. Zhang, J. Orchard, *Improving neural-network classifiers using nearest neighbor partitioning*, IEEE Trans. Neural Netw. Learn. Syst., **28** (2016), 2255–2267.
[17] L. Wang, B. Yang, J. Orchard, *Particle swarm optimization using dynamic tournament topology*, Appl. Soft Comput., **48** (2016), 584–596. 1
[18] Z. Wilcox-O'Hearn, B. Warner, *Tahoe: the least-authority filesystem*, Proceedings of the 4th ACM International Workshop on Storage Security and Survivability, Alexandria, Virginia, USA, (2008), 21–26. 5
[19] C. P. Wright, M. C. Michael, E. Zadok, *NCryptfs: A secure and convenient cryptographic file system*, Proceedings of the USENIX Annual Technical Conference, San Antonio, USA, (2003), 197–210. 5
[20] F. Yingxun, L. Shengmei, S. Jiwu, *A secure network disk system in cloud storage environment*, J. Softw., **25** (2014), 1831–1843. 1
[21] E. Zadok, I. Badulescu, A. Shender, *Cryptfs: A stackable vnode level encryption file system*, Technical Report CUCS-021-98, Computer Science Department, Columbia University, New York, (1998). 5
[22] T. Zhong, J.-Z. Geng, H. Xiong, Z.-G. Qin, *The data integrity verification mechanism based on SBT in cloud storage*, J. Univ. Electron. Sci. Technol. China, **6** (2014), 929–933. 3.3, 3.3.3