Secure Multi-owner Data Sharing For Dynamic Group In Cloud

C.Kayalvizhi, S.Arun Prasath, S.ArunKumar, C.Broons Gandhi

Computer Science and Engineering,
Dhanalakshmi College of Engineering.

Abstract

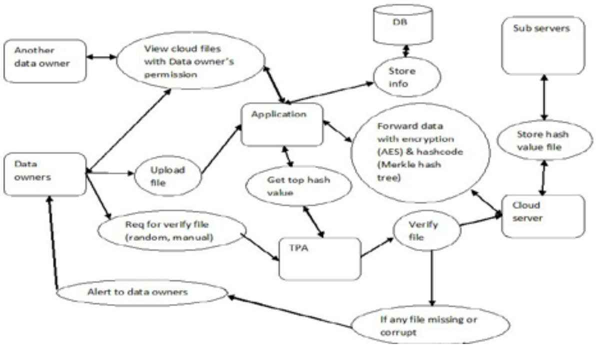
Security is a major issue in cloud computing environment as the resources are dynamic, virtualized, scalable and elastic in nature. Data Integrity is ensured. Auditing plays a vital role in providing solution to the data integrity in cloud. Highly distributed and non-transparent nature of cloud increases the complexity of Auditing process. Auditing deals with monitoring and compliance. A third party auditor is essential to perform auditing to ensure data integrity on cloud services. In this paper, a Dynamic Third Party Auditing System is proposed in which a third party entity dynamically provides auditing services on cloud computing environment. TPA makes task of Client by verifying the integrity of data stored in cloud. The Dynamic third party auditing system does auditing using public key based homomorphic authentication.

Index Terms

Cloud Computing, Auditing, Load Balancing, Third Party Auditor.

1. Introduction

Cloud computing is the new paradigm of data storing and data sharing. Cloud is a large number of interconnected computers [1] [2]. It allows the users to access the resources like operating systems and professional software like visual studio and adobe etc. The users can pay for what they use as it reduces cost. Although the advent of cloud computing ameliorates the problem of data loss, the need for security is not contented. Security is a vital part for consideration. To ensure security, cryptic techniques cannot be directly employed as service provider preserves their reputation by hiding the data corruption [5] [6] [7]. This problem can be eliminated by both manual auditing and automatic auditing, this process collectively performed by Trusted Party Auditor [3] [4]. Automatic auditing examines every data while manual auditing examines particular part of data. These auditing tasks ensure the data possessor that his data are safe. Load balancing concept is employed for resource allocation and job scheduling in a distributed environment. The entire data is encrypted using Advanced Encryption Standard algorithm. The encrypted data is divided and stored using Merkle Hash Tree algorithm.


DB
Sub servers
Another data owner
View cloud files with Data owner's permission
Store info
Application
Forward data with encryption (AES) & hashcode (Merkle hash tree)
Store hash value file
Data owners
Upload file
Get top hash value
Cloud server
Req for verify file (random, manual)
TPA
Verify file
Alert to data owners
If any file missing or corrupt

2. Load Balancing

Load balancing concept performs two major tasks, one is the resource allocation and other is job scheduling. It ensures resources are easily available on demand and are efficiently used under the condition of high or low load. Thus cost of using resources is reduced. Energy is also saved in case of low load. Resource allocation is the task of mapping of the resources to different entities of cloud on demand basis. Resources must be allocated in such a manner that no node in the cloud is overloaded and all the available resources in the cloud do not undergo any kind of wastage. Task scheduling is done after the resources are allocated to all cloud entities. Scheduling defines the manner in which different entities are provisioned. Resource provisioning defines which resource will be available to meet user requirements whereas task scheduling defines the manner in which the allocated resource is available to the end user (i.e. whether the resource is fully available until task completion or is available on sharing basis). Task scheduling provides “Multiprogramming Capabilities” in the cloud.

Task scheduling can be done in two modes:
a. Space shared
b. Time shared

3. Random Key Generation

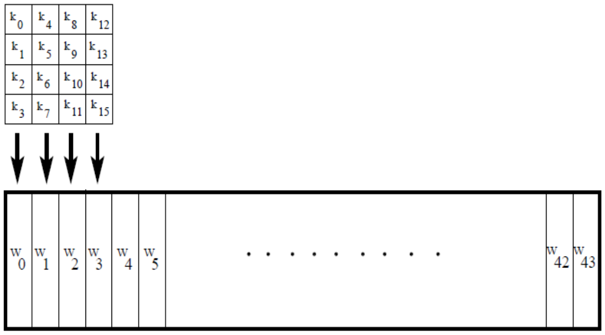
For the generation of the key, a random string generation algorithm is used to create a unique key. The key so

Manuscript received March 5, 2016
Manuscript revised March 20, 2016

generated is then encrypted by using AES Algorithm for security purpose. It can be used as a replacement for the DES Algorithm. It takes variable key length ranging from 32 bits to 448 bits and the default size is 128 bits. In this paper, this algorithm is used to encrypt the key which has to be passed in the AES algorithm. This is done to provide extra security.

4. Advanced Encryption Standard Algorithm

Assuming a 128-bit key, the key is also arranged in the form of a matrix of 4 × 4 bytes. As with the input block, the first word from the key fills the first column of the matrix, and so on. The four column words of the key matrix are expanded into a schedule of 44 words. In every round it takes four words of the key schedule. It also depicts the arrangement of the encryption key in the form of 4-byte words and the expansion of the key into a key schedule consisting of 44 4-byte words.

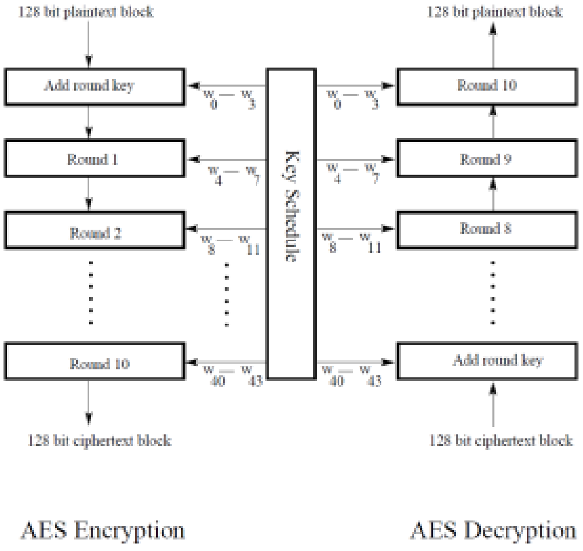

k_0 k_4 k_8 k_{12}
k_1 k_5 k_9 k_{13}
k_2 k_6 k_{10} k_{14}
k_3 k_7 k_{11} k_{15}
w_0 w_1 w_2 w_3 w_4 w_5 . . . . w_{42} w_{43}

4.1 The Overall Structure

The number of rounds is for the case when the encryption key is 128 bit long. If the keys are 192 and 256 bits, then the number of rounds are 12 and 14 respectively. Before any round-based processing for encryption can begin, the input state array is XORed with the first four words of the key schedule. The same thing happens during decryption except that now we XOR the cipher text state array with the last four words of the key schedule.

For encryption, each round consists of the following four steps: 1) Substitute bytes, 2) Shift rows, 3) Mix columns, and 4) Add round key. The last step consists of XORing the output of the previous three steps with four words from the key schedule.

For decryption, each round consists of the following four steps: 1) Inverse shift rows, 2) Inverse substitute bytes, 3) Add round key, and 4) Inverse mix columns. The third step consists of XORing the output of the previous two steps with four words from the key schedule. Note the differences between the order in which substitution and shifting operations are carried out in a decryption round the order in which similar operations are carried out in an encryption round. The last round for encryption does not involve the “Mix columns” step. The last round for decryption does not involve the “Inverse mix columns” step.



5. Merkle Hash Tree

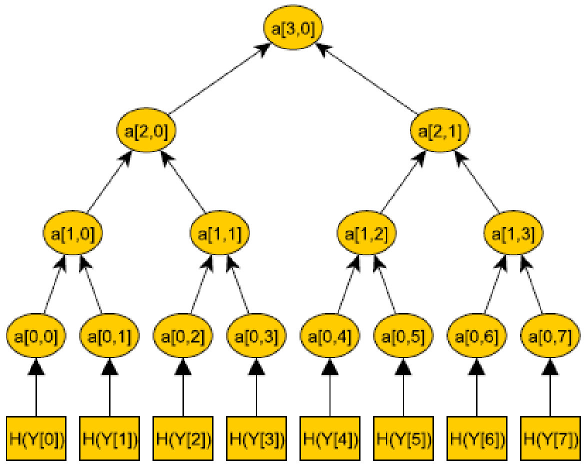
By using Merkle Hash Tree algorithm the data will be audited via multiple level of batch auditing process. The top hash value is stored in local database and other hash code files are stored in cloud. Thus the original data cannot be retrieved by anyone from cloud, since the top hash value is not in cloud. Even if any part of data gets hacked, it is of no use to the hacker [5] [8].Thus, the security can be ensured.

Step 1: A file is split up into n number of data blocks.
Step 2: Each data block is hashed and these hashes of data blocks are the leaves in hash tree.
Step 3: Nodes further up in the tree are the hashes of their respective children.
Step 4: Final hash value in a single node becomes a top hash value.

5.1 MERKLE-SIGNATURE SCHEME

The biggest problem of One-Time Signature Schemes is the key management. Exchanging a public key is very complex. It must be guaranteed, that the public key belongs to the intended communication partner and that the public key has not been modified. Therefore, few public keys should be used and the public keys should be rather short. But in One-Time Signature Schemes, a new public key is used for every signature and the public key is quite big, compared with other

Signature schemes. To make One-Time Signature Schemes feasible, an efficient key management, that reduces the amount of public keys and their size, is needed. In Merkle introduced the Merkle Signature Scheme (MSS), in which one public key is used to sign many messages.



5.2 KEY GENERATION

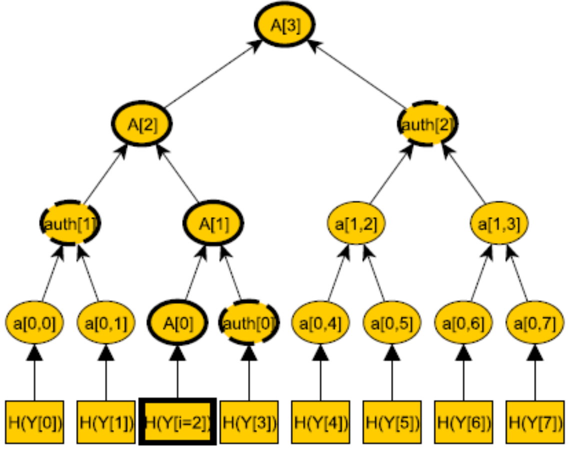
The Merkle Signature Scheme can only be used to sign a limited number of messages with one public key pub. The number of possible messages must be a power of two, so that we denote the possible number of messages as N = 2n. The first step of generating the public key pub is to generate the public keys Xi and private keys Yi of 2n one-time signatures, as described in chapter 2. For each public key Yi, with 1 ≤ i ≤ 2n, a hash value hi = H(Yi) is computed. With these hash values hi a Merkle Tree (also called hash tree) is build. We call a node of the tree ai, j where i denotes the level of the node. The level of a node is defined by the distance from the node to a leaf. Hence, a leaf of the tree has level i = 0 and the root has level i = n. We number all nodes of one level from the left to the right, so that ai,0 is the leftmost node of level i. In the Merkle Tree the hash values hi are the leafs of a binary tree, so that hi = a0,i. Each inner node of the tree is the hash value of the concatenation of its two children.
So a1, 0 = H(a0,0||a0,1) and a2,0 = H(a1,0||a1,1)
In this way, a tree with 2n leafs and 2n+1 − 1 nodes is build. The root of the tree an, 0 is the public key pub of the Merkle Signature Scheme.

5.3 SIGNATURE GENERATION

To sign a message M with the Merkle Signature Scheme, the message M is signed with a one-time signature scheme, resulting in a signature sig′, first. This is done, by using one of the public and private key pairs (Xi, Yi). The corresponding leaf of the hash tree to a one-time public key Yi is a0,i = H(Yi). We call the path in the hash tree from a0,i to the root A. The path A consists of n + 1 nodes, A0, ...An, with A0 = a0,i being the leaf and An = an,0 = pub being the root of the tree. To compute this path A, we need every child of the nodes A1, ..., An. We know that Ai is a child of Ai+1. To calculate the next node Ai+1 of the path A, we need to know both children of Ai+1. So we need the brother node of Ai. We call this node authi, so that Ai+1 = H(Ai||authi). Hence, n nodes auth0, ..., authn−1 are needed, to compute every node of the path A. We now calculate and save these nodes auth0, ..., authn−1. These nodes, plus the one-time signature sig′ of M is the signature sig = (sig′||auth2||auth3||...||authn−1) of the Merkle Signature Scheme.

5.4 SIGNATURE VERIFICATION

The receiver knows the public key pub, the message M, and the signature sig = (sig′||auth0||auth1||...||authn−1). At first, the receiver verifies the one-time signature sig′ of the message M. If sig′ is a valid signature of M, the receiver computes A0=H(Yi) by hashing the public key of the one-time signature. For j =1, .., n−1, the nodes of Aj of the path A are computed with Aj = H(aj−1||bj−1).



The desired security requirements and it guarantees efficiency as well.

5.5 COST ANALYSIS

The big advantage of the Merkle Signature Scheme is, that many signatures can be generated with using only one public key. However, this advantage comes with an increase of computation time and signature length. In the following we will examine the computation time of each part of the signature process. To generate the public key pub, 2n one-time signature keys must be generated. Then every node of the hash tree must be computed. The tree consists of 2n+1−1 nodes. One hash operation is needed to calculate a node, so that 2n+1 − 1 hash operations are needed to generate the public key. It is obvious, that the

size of such a tree is limited. To compute 240 nodes is very costly, to compute 280 nodes is impossible.
To generate a signature the nodes auth0, ..., authn−1 are needed. If you do not store the nodes of the tree, the nodes must be generated again for every signature. Generating the tree is very expensive, so that generating the entire tree for every signature is impracticable for bigger trees. But saving all 2n+1−1 nodes would result in huge storage requirements. Hence, a good strategy is needed, to generate the signature without saving too many nodes, at a still efficient time.
This problem is called The Merkle tree traversal problem. The verification time is quite fast, compared to the signature time. At first, the one-time signature must be verified. After that, the path A = A1, ..., An must be computed. To do this, only n hash operations are needed, one for every node. The signature of the Merkle Signature Scheme consists of the one-time signature sig′ and n nodes auth0, ..., authn−1. If a 160 bit hash function is used, the signature size would be |sig| = |sig′| + n ∗ 160 bits.

5.6 MERKLE TREE TRAVERSAL TECHNIQUES

For the traversal techniques, we need an algorithm, that computes efficiently the nodes of the tree. Assume a binary tree with 2n leafs. The height H of a node, is defined by the distance of the node to a leaf. So the root has the height H = n, while the leafs have the height H = 0. We define the node ai,j as the jth node from the left (starting with j = 0) of the height i. So a0,0 is the leftmost leaf of the tree, and an,0 the root. To compute a node of the height H = h, 2h − 1 nodes must be computed. The tree hash algorithm needs 2h −1 operations, to calculate a node of the height h, while saving as few nodes at once as possible. The main idea of the tree hash algorithm is to calculate the needed subtree from left to right and only saving the nodes, that are still needed. This is done by using a stack. At first the stack only consists of the leftmost leaf. Then the next leaf is added. The algorithm now checks whether the last two nodes on the stack are of the same height or not. If they are of the same height, the two nodes are removed from the stack, and their parent is built and pushed on the stack. If
the last two nodes on the stack are of different height, then a new leaf is pushed on the stack. This step is repeated, until the node of the wanted height has been generated.

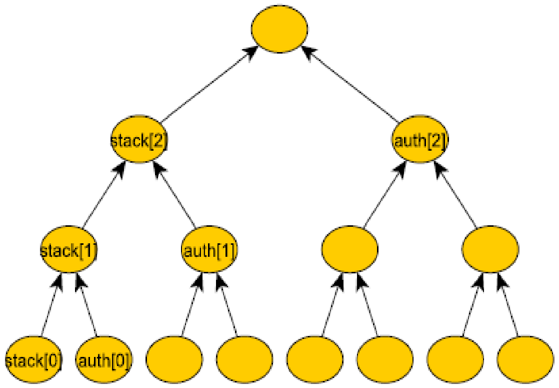
ALGORITHM: TREEHASH (start, maxheight)

```
1.  Set leaf = start and create empty stack.
2.  Consolidate: If top 2 nodes on the stack are equal
    height:
    • Pop node value P(nright) from stack.
    • Pop node value P(nleft) from stack.
    • Compute P(nparent) = f(P(nleft||P(nright)).
    • If height of P(nparent) = maxheight, output
      P(nparent).
    • Push P(nparent) onto the stack.
3. New Leaf: Otherwise:
    • Compute P(nl) = LEAFCALC(leaf).
    • Push P(nl) onto the stack.
    • Increment leaf.
4.  Loop to step 2.
```

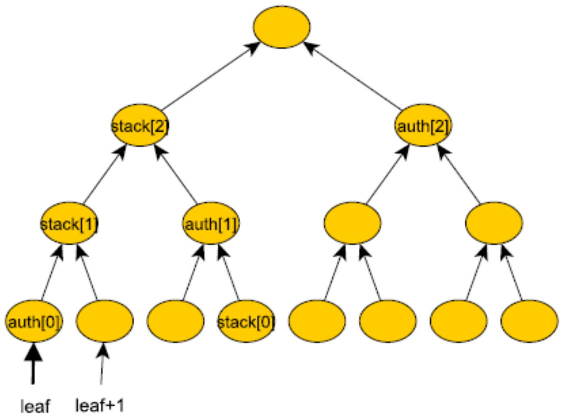
To be able to run multiple instances of tree hash, we define an object stackh with two methods, stackh.initialize(startnode, h) and stackh.update(t). Withthe initialize method we simply define the start leaf and the height of the resulting node. The method update runs the steps 2 or 3 of the treehash algorithm t times. For example stack2.initialize (0, 2) means, that in stack2 we compute nodes up to the height h = 2, beginning with the 0th node. stack2.update (3) will now perform 3 operations of tree hash on stack2. The first operation will be to push node a0,0 on the stack. The second operation will be to push the node a0,1 on the stack. Now the last two nodes on the stack are of equal height. So in the third operation these two nodes are removed and a1, 0 gets computed and push on the stack. Because the tree hash should only perform three operations, the algorithm stops at this point. Whenstack2.update(t) is called again, the algorithm will continue at this point, by pushing the node a0, 2 on the stack.

5.7 THE CLASSIC TRAVERSAL

In the first step of the Merkle Signature Scheme, the public key, which is the root of the tree, gets computed. This is done, by using the treehash algorithm. During this computation, every node of the tree is generated, so that we can easily save
the first authentication path auth. We do this, by saving all nodes auth I with authi = ai,1 for i = 1, ..., n − 1. These nodes auth = {auth1, ..., authn−1} are the right brothers of the nodes of the leftmost path. In addition to the authi nodes we also store the nodes of the leftmost path in the objects stacki, with
stacki = ai,0 for i = 1, ..., n−1. We will need these objects, to efficiently generate the next authentication path. The next phase is the output and update phase. In this phase, we output the leaf value together with the authentication path. After that, we generate the next authentication path. Generating the output is quite simple. We use the function LEAFCALC to calculate the value of the leaf (The leaf values is the hash value of the public key of the one-time signature. So LEAFCALC builds the hash value of the one-time signature public key). The authentication path auth = auth1, ..., authn−1 has been already computed. So the important part is to calculate the next authentication path.



To do this, we need a counter leaf, which points to the current leaf to be calculated, and we need the old authentication path auth. In addition to that, we also have the objects stacki for i =0, ..., n−1. We can modify these by the functions stacki.initialize(startnode, h) and stacki.update(t). We now have to determine which authentication nodes authh have to be changed, so that auth = autho, ..., authH−1 is the authentication path for the next leafleaf + 1. The authentication node of the height h only needs an update, if 2h divides leaf + 1 without remainder. The new authentication node authh has already been generated and is saved in the stack stackh. So if 2h divides leaf +1, authh = POP(stackh). Then stackh is empty and we use this stack to precalculate the next authentication node. In 2h steps, when leaf = leaf +1+2h, authh needs an update again. So we search for the leftmost leaf, startnode, of the next authentication node, of the height h. This is startnode = leaf + 1 + 2h + 2h if the current authh is a left-node and startnode = leaf + 1 if the current authh is a right-node. So startnode = leaf + 1 + 2h ⊕ 2h. Hence we set stackh.initialize(startnode, h). The next change of auth1 will be when leaf = leaf+1+21. Hence, we need the authentication node of level 1 for the leaf +1+21. This node is sack1. The leftmost leaf of this node stack1 is leaf+1+21+21 = startnode. SO stack1.initialize(leaf+1+21+21, 1).We could now use the treehash algorithm to compute stackh at once. But this would take 2h+1 − 1 steps. In the worst case, H − 1 nodes authh can change at once, so that we would needH−1 Ph=0 2h+1 − 1 operations to compute one signature.We know, that we do not need to change authh for the next 2h signatures. Hence, we have 2h signatures time, to make the 2h+1 − 1 operations which generate thenext node. Therefore, we only do two operations of updating for h = 0, ...,H −1 per signature, by calling stackh.update(2) for h = 0, ...,H − 1. In this way, we only perform (H − 1) ∗ 2 Operations per signature in the worst case.



ALGORITHM: CLASSIC MERKLE TREE TRAVERSAL

```
1. Set leaf = 0.
2. Output:
 • Compute and output leaf with LEAFCALC(leaf)
 • For each h ∈ [0,H − 1] output {authh}.
3. Refresh Auth Nodes:
For h such that 2h divides leaf + 1:
 • Set authh be the sole node value in stackh.
 • Set startnode = (leaf + 1 + 2h) ⊕ 2h.
 • stackh.initialize(startnode, h).
4. Build Stacks:
For all h ∈ [0,H − 1]:
 • stackh.update(2).
    5. Loop
 • Set leaf = leaf + 1.
 • If leaf < 2H go to Step 2.
```

6. CONCLUSION

In this paper developed an Innovative approach for secure multi-owner data sharing for dynamic groups in an untrusted cloud. In this scheme a user is able to share data with others in the group without revealing identity privacy to the cloud. Efficient user revocation can be achieved through a public revocation list without updating the private keys of the remaining users, and new users can directly decrypt files stored in the cloud before their participation. The storage overhead and the encryption computation cost are varied. Extensive analyses show that the proposed scheme satisfies

References

[1] Y. Deswarte, J. Quisquater, and A. Saidane, “Remote integrity checking”, In Proc. of Conference on Integrity and Internal Control in Information Systems (IICIS’03), November \2003.

[2] T. Schwarz and E.L. Miller, “Store, forget, and check: Using algebraic signatures to check remotely administered storage”, In Proceedings of ICDCS ’06. IEEE Computer Society, 2006.

[3] C. Wang, S.S.M. Chow, Q. Wang, K. Ren, and W. Lou,“Privacy-Preserving Public Auditing for Secure Cloud Storage,” IEEE Trans.Computers, vol. 62, no. 2, pp. 362-375, Feb. 2013.

[4] M. Venkatesh, “Improving Public Auditability, Data Possession in Data Storage Security for Cloud Computing”, ICRTIT-IEEE 2012.

[5] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li, “Enabling Public Auditability and Data Dynamics for Storage Security in CloudComputing,” IEEE Trans. Parallel and Distributed Systems vol. 22,no. 5, pp. 847-859, May 2011.

[6] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z.Peterson, and D. Song, “Provable Data Possession at Untrusted Stores,” Proc. 14th ACM Conf. Computer and Comm. Security (CCS ’07), pp. 598-609, 2007.

[7] M.A. Shah, R. Swaminathan, and M. Baker, “PrivacyPreserving Audit and Extraction of Digital Contents,” Cryptology ePrint Archive, Report 2008/186, 2008.

[8] P. Golle, S. Jarecki, and I. Mironov “Cryptographic primitives enforcing communication and storage complexity”. In Financial Cryptography, pages 120-135, 2002.