IPHV7I10005X
International Journal Of Advanced Research and Innovation -Vol.7, Issue .I
ISSN Online: 2319 – 9253
Print: 2319 – 9245

A New Approach To Nurture And Authenticate Aegis In Multi Cloud Out Of Possession Of Single Cloud

[1]M.Kruthika Reddy, [2] A.Srinivas Reddy

Dept of CSE
GITAM University, Hyderabad

Abstract:-Recent interest in Cloud Computing has been driven by new offerings of computing resources that are attractive due to per-use pricing and elastic scalability, providing a significant advantage over the typical acquisition and deployment of equipment that was previously required. The effect has been a shift to outsourcing of not only equipment setup, but also the ongoing IT administration of the resources as well. The Cloud has become a new vehicle for delivering resources such as computing and storage to customers on demand. Rather than being a new technology in itself, the cloud is a new business model wrapped around new technologies such as server virtualization that take advantage of economies of scale and multi-tenancy to reduce the cost of using information technology resources. In relation to data intrusion and data integrity, assume we want to distribute the data into three different cloud providers, and we apply the secret sharing algorithm on the stored data in the cloud provider. An intruder needs to retrieve at least three values to be able to find out the real value that we want to hide from the intruder. Hence, we provide a fake object scheme in which an alert through the fake object creation will be given to the admin in order to maintain data integrity. This paper also surveys recent research related to single and multi-cloud security and addresses possible solutions. It is found that the research into the use of multi cloud providers to maintain security has received less attention from the research community than has the use of single clouds. This work aims to promote the use of multi-clouds due to its ability to reduce security risks that affect the cloud computing user.

General Terms: - *Data Protection, Data Sharing.*

Keywords:- *Map Reduce, Fake Object Scheme, Key Management, Secure Storage.*

I.INTRODUCTION

The abilities to use multiple Clouds and to migrate, at design or at run-time, applications from one Cloud to another could mitigate the risk of Cloud adoption and would allow building high performance and reliable applications. The business world now demands a mix of many best-of-breed cloud services to form the optimal solution. The answer is proving to be a concept called "multicloud". Its more complex than a hybrid cloud, which is typically a paired private and private cloud. Multi cloud add more clouds to the mix, perhaps two or more public iaas providers, a private paas, on-demand management and security systems form public clouds, private used accounting. Multi clouds requires more thinking around security and governance, given their complexity and distribution, it may develop resiliency issues, considering the number of moving parts, and these have value only if you select the right providers, whether on-demand or private.

Fake Object Schema:

Cloud computing holds the promise of revolutionizing the manner in which enterprises manage, distribute, and share information. The data owner (client) can out-source almost all its information processing tasks to a "cloud". The cloud can be seen as a collection of servers (we shall sometimes refer to it as the server) which caters the data storage, processing and

maintenance needs of the client. Needless to say this new concept of computing has already brought significant savings in terms of costs for the data owner. Among others, an important service provided by a cloud is Database as a Service (DAS). In this service the client delegates the duty of storage and maintenance of his/her data to a third party (an un-trusted server). This model has gained lot of popularity in the recent times. The DAS model allows the client to perform operations like create, modify and retrieve from databases in a remote location [9]. These operations are performed by the server on behalf of the client. However, delegating the duty of storage and maintenance of data to a third party brings in some new security challenges. The two main security goals of cryptography are privacy and authentication. These security issues are relevant to the outsourced data also. The client who keeps the data with an untrusted server has two main concerns. The first one being that the data may be sensitive and the client may not want to reveal the data to the server and the second one is the data whose storage and maintenance has been delegated to the server would be used by the client. The typical usage of the data would be that the client should be able to query the database and the answers to the client's queries would be provided by the server. It is natural for the client to be concerned about a malicious server who does not provide correct answers to the client queries. In this work we are interested in this problem. We aim to devise a scheme in which the client would be able to verify whether the server is responding correctly to its queries.

The problem of interest is of data authentication, and there are well known cryptographic solutions to the basic data authentication problem. In the symmetric key setting this has been addressed by the use of message authentication codes and in the asymmetric key setting signature schemes provide this functionality.

We are interested in the problem of authenticated query processing in the context of relational databases. We consider the scenario where a client delegates a relational database to an untrusted server. When the client queries its outsourced data, it expects in return a set of records (query reply) satisfying the query's predicates. As the server is not trusted, so it must be capable of proving the correctness of its responses. In other words, a malicious server may attempt to insert fake records into the database, modify existing records or simply skip some of them from the query response. Hence there must exist a mechanism, which can protect the client from such malicious server behavior. We describe the intricacies of the problem with the help of an example. Consider the relational database of employees data shown in Table 1.

EmpId	Name	Gender Level
TRW	Tom	M L2
MST	Mary	F L1
JOH	John	M L2
LCT	Lucy	F L1
ASY	Anne	F L1
RZT	Rosy	F L2

Table 1. Relation R1 (This relation would serve as a running example).

We consider that this relation has been delegated by a client to a server, and the client poses the following query

SELECT * FROM R1 WHERE Gender = 'M' OR Level = 'L2'.

The correct response to this query is the set Res consisting of three tuples

Res = {(TRW, Tom, M,L2), (JOH, John, M,L2), (RZT, Rosy, F,L2)}.

In answering the query the server can act maliciously in various ways. In the context of authentication, we are concerned with two properties of the response namely correctness and complete- ness. Correctness and completeness denote two different malicious activities of the server, we explain these notions with an example below:

1. Incorrect result: The server responds with three tuples, but changes the tuple (TRW, Tom, M,L2), with (TRW, Tom, F,L2). Moreover, it can be the case that the server responds with Res ∪ {(BRW, Bob, M,L2)}, i.e., it responds with an extra tuple which is not a part of the original relation.

2. Incomplete result: The server may not respond with the complete result, i.e., it can delete some valid results from the response, i.e., instead of responding with Res it responds with Res − {(TRW, Tom, M,L2)}.

It is to be noted that incomplete results are also incorrect, but we differentiate the two scenarios (as it has been previously done in the literature) by the fact that in an incorrect result the server inserts something which is not in the database, and in case of an incomplete answer the answer is correct but is not complete, in the sense that the server drops some valid tuples from the correct response. A client must be able to verify both correctness and completeness of a response. The problem of correctness can be easily handled in the symmetric setting by adding a message authentication code to each tuple. A secure message authentication code is difficult to forge, and thus this property would not allow the server to add fake entries in its response. The completeness problem is more difficult and its solution is achieved through more involved schemes.

The problem of query completeness has been largely addressed by some interesting use of authenticated data structures. The basic idea involved is to store the information already present in the relation in a different form using some special data structures. This redundancy along with some special structural properties of the used data structures help in verifying completeness. A large part of the literature uses tree based authentication structures like the Merkeley hash tree or its variants. Some notable works in this direction are reported in. These techniques involve using a special data structure along with some cryptographic authentication mechanism like hash functions and/or signatures schemes. The tree based structures yield reasonable communication and verification costs. But, in general they require huge storage at server side, moreover the query completeness problem is largely addressed with respect to range queries and such queries may not be relevant in certain scenarios, say in case of databases with discrete attributes which do not have any natural metric relationship among them. Signature schemes have also been used in a novel manner for solving the problem. One line of research has focused on aggregated signatures. Signature aggregation helps in reducing the communication cost to some extent and in some cases can function with constant extra communication overhead. A related line of research uses chain signatures. If one uses chain signatures as in , the use of specialized data structures may no longer be required. Though there has been considerable amount of work on authenticated query processing on relational databases, but it has been acknowledged that the problem of query authentication largely remains open. An unified cryptographic treatment of the problem is missing in the literature. In most existing schemes cryptographic objects have been used in an ad-hoc manner, and the security guarantees that the existing schemes provide are not very clear. In this work we initiate a formal cryptographic study of the problem of query authentication in a distinct direction. We propose a new scheme which does not use any specialized data structure to address the completeness problem. Our solution involves usage of bitmap indices for this purpose. Bitmap indices have gained lot of popularity in the current days for their use in accelerated query processing [2], and many commercially available databases like Oracle, IBM DB2, Sybase IQ now implement some form of bitmap index scheme in addition to the more traditional B-tree based schemes, thus it may be easy to incorporate a bitmap based scheme in a modern database without significant extra cost. To our knowledge, bitmaps have not been used till date for a security goal. In addition to bitmap indices we use a secure message authentication code (MAC) as the only cryptographic object. We show that by the use of these simple objects one can design a query authentication scheme which allows verification of both correctness and completeness of query results. As the basic cryptographic object is a symmetric key primitive, thus our scheme does not provide public verifiability. Moreover, in this work, we restrict ourself to static databases only. We see private verifiability of our scheme more as a design goal than a limitation, as we believe that

there exist scenarios where public verifiability may not be required, and in such scenarios it is better not to use the heavy machinery of public key signatures which uses computationally intensive number theoretic operations, whereas computational overheads of symmetric key message authentication schemes are minimal. Extension of our scheme to dynamic scenarios may be possible, but in this current work we do not further deal with such possibilities, we plan to discuss the extension of our scheme to dynamic scenarios in a separate work. There exist many static transactional databases, say databases related to data warehousing applications, where efficient authenticated query processing may be required.

II. MAP REDUCE

MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster. A Map Reduce program is composed of a Map() procedure that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a Reduce()procedure that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The "Map Reduce System" (also called "infrastructure" or "framework") orchestrates by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

The model is inspired by the map and reduce functions commonly used in functional programming, although their purpose in the Map Reduce framework is not the same as in their original forms. Furthermore, the key contributions of the MapReduce framework are not the actual map and reduce functions, but the scalability and fault-tolerance achieved for a variety of applications by optimizing the execution engine once. Map Reduce libraries have been written in many programming languages, with different levels of optimization. A popular open-source

implementation is cloud computing. The name MapReduce originally referred to the proprietary Google technology but has since been genericized.

'MapReduce' is a framework for processing parallelizable problems across huge datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes are on the same local network and use similar hardware) or a grid (if the nodes are shared across geographically and administratively distributed systems, and use more heterogenous hardware). Computational processing can occur on data stored either in a filesystem (unstructured) or in a database (structured). MapReduce can take advantage of locality of data, processing data on or near the storage assets to decrease transmission of data. "Map" step: The master node takes the input, divides it into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes the smaller problem, and passes the answer back to its master node.

"Reduce" step: The master node then collects the answers to all the sub-problems and combines them in some way to form the output – the answer to the problem it was originally trying to solve. MapReduce allows for distributed processing of the map and reduction operations. Provided that each mapping operation is independent of the others, all maps can be performed in parallel – though in practice this is limited by the number of independent data sources and/or the number of CPUs near each source. Similarly, a set of 'reducers' can perform the reduction phase, provided that all outputs of the map operation that share the same key are presented to the same reducer at the same time, or that the reduction function is associative. While this process can often appear inefficient compared to algorithms that are more sequential, MapReduce can be

applied to significantly larger datasets than "commodity" servers can handle – a large server farm can use MapReduce to sort a petabyte of data in only a few hours. The parallelism also offers some possibility of recovering from partial failure of servers or storage during the operation: if one mapper or reducer fails, the work can be rescheduled – assuming the input data is still available.

Another way to look at MapReduce is as a 5-step parallel and distributed computation:

1. Prepare the Map() input – the "MapReduce system" designates Map processors, assigns the K1 input key value each processor would work on, and provides that processor with all the input data associated with that key value.
2. Run the user-provided Map() code – Map() is run exactly once for each K1 key value, generating output organized by key values K2.
3. "Shuffle" the Map output to the Reduce processors – the MapReduce system designates Reduce processors, assigns the K2 key value each processor would work on, and provides that processor with all the Map-generated data associated with that key value.
4. Run the user-provided Reduce() code – Reduce() is run exactly once for each K2 key value produced by the Map step.
5. Produce the final output – the MapReduce system collects all the Reduce output, and sorts it by K2 to produce the final outcome.

Logically these 5 steps can be thought of as running in sequence – each step starts only after the previous step is completed – though in practice, of course, they can be intertwined, as long as the final result is not affected. In many situations the input data might already be distributed ("sharded") among many different servers, in which case step 1 could sometimes be greatly simplified by assigning Map servers that would process the locally present input data. Similarly, step 3 could sometimes be sped up by assigning Reduce processors that are as much as possible local to the Map-generated data they need to process.

III.LOGICAL VIEW

The *Map* and *Reduce* functions of *MapReduce* are both defined with respect to data structured in (key, value) pairs. *Map* takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

Map(k1,v1) → list(k2,v2)

The *Map* function is applied in parallel to every pair in the input dataset. This produces a list of pairs for each call. After that, the MapReduce framework collects all pairs with the same key from all lists and groups them together, creating one group for each key. The *Reduce* function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

Reduce(k2, list (v2)) → list(v3)

Each *Reduce* call typically produces either one value v3 or an empty return, though one call is allowed to return more than one value. The returns of all calls are collected as the desired result list. Thus the MapReduce framework transforms a list of (key, value) pairs into a list of values. This behavior is different from the typical functional programming map and reduce combination, which accepts a list of arbitrary values and returns one single value that combines *all* the values returned by map. It is necessary but not sufficient to have implementations of the map and reduce abstractions in order to implement MapReduce.

Examples

IPHV7I10005X
International Journal Of Advanced Research and Innovation -Vol.7, Issue .I
ISSN Online: 2319 – 9253
Print: 2319 – 9245

```
function map(String name, String document):
  // name: document name
  // document: document contents
  for each word w in document:
    emit (w, 1)

function reduce(String word, Iterator partialCounts):
  // word: a word
  // partialCounts: a list of aggregated partial counts
  sum = 0
  for each pc in partialCounts:
    sum += ParseInt(pc)
  emit (word, sum)
```

Here, each document is split into words, and each word is counted by the *map* function, using the word as the result key. The framework puts together all the pairs with the same key and feeds them to the same call to *reduce*, thus this function just needs to sum all of its input values to find the total appearances of that word.

```
 SELECT age, AVG(contacts)
   FROM social.person
GROUP BY age
ORDER BY age
```

Using MapReduce, the K1 key values could be the integers 1 through 1,100, each representing a batch of 1 million records, the K2 key value could be a person's age in years, and this computation could be achieved using the following functions:

```
function Map is
    input: integer K1 between 1 and 1100, representing a batch of 1 million social.person records
    for each social.person record in the K1 batch do
        let Y be the person's age
        let N be the number of contacts the person has
        produce one output record (Y,(N,1))
    repeat
end function
```

```
    input: age (in years) Y
    for each input record (Y,(N,C)) do
        Accumulate in S the sum of N*C
        Accumulate in C_new the sum of C
    repeat
    let A be S/C_new
    produce one output record (Y,(A,C_new))
end function
```

The MapReduce System would line up the 1,100 Map processors, and would provide each with its corresponding 1 million input records. The Map step would produce 1.1 billion (Y,(N,1))records, with Y values ranging between, say, 8 and 103. The MapReduce System would then line up the 96 Reduce processors by performing shuffling operation of the key/value pairs due to the fact that we need average per age, and provide each with its millions of corresponding input records. The Reduce step would result in the much reduced set of only 96 output records (Y,A), which would be put in the final result file, sorted by Y. The count info in the record is important if the processing is reduced more than one time. If we don't add the count of the records, the computed average would be wrong, for example:

```
-- map output #1: age, quantity of contacts
10, 9
10, 9
10, 9
-- map output #2: age, quantity of contacts
10, 9
10, 9
-- map output #3: age, quantity of contacts
10, 10
```

IPHV7I10005X
International Journal Of Advanced Research and Innovation -Vol.7, Issue .I
ISSN Online: 2319 – 9253
Print: 2319 – 9245

If we reduce files #1 and #2, we will have a new file with an average of 9 contacts for a 10 year old person ((9+9+9+9+9)/5):

```
-- reduce step #1: age, average of contacts
10, 9
```

If we reduce it with file #3, we lost the count of how many records we've already seen, so we would end up with an average of 9.5 contacts for a 10 year old person ((9+10)/2), which is wrong. The correct answer is 9.17 ((9+9+9+9+9+10)/6).

Dataflow:

The frozen part of the MapReduce framework is a large distributed sort. The hot spots, which the application defines, are:

- an *input reader*
- a *Map* function
- a *partition* function
- a *compare* function
- a *Reduce* function
- an *output writer*

3.1 Input reader:

The *input reader* divides the input into appropriate size 'splits' (in practice typically 16 MB to 128 MB) and the framework assigns one split to each *Map* function. The *input reader* reads data from stable storage (typically a distributed file system) and generates key/value pairs.

A common example will read a directory full of text files and return each line as a record.

3.2 Map function

The *Map* function takes a series of key/value pairs, processes each, and generates zero or more output key/value pairs. The input and output types of the map can be (and often are) different from each other. If the application is doing a word count, the map function would break the line into words and output a key/value pair for each word. Each output pair would contain the word as the key and the number of instances of that word in the line as the value.

3.3Partition function

Each *Map* function output is allocated to a particular *reducer* by the application's *partition* function for sharding purposes. The *partition* function is given the key and the number of reducers and returns the index of the desired *reducer*. A typical default is to hash the key and use the hash value modulo the number of *reducers*. It is important to pick a partition function that gives an approximately uniform distribution of data per shard for load-balancing purposes, otherwise the MapReduce operation can be held up waiting for slow reducers (reducers assigned more than their share of data) to finish. Between the map and reduce stages, the data is *shuffled* (parallel-sorted / exchanged between nodes) in order to move the data from the map node that produced it to the shard in which it will be reduced. The shuffle can sometimes take longer than the computation time depending on network bandwidth, CPU speeds, data produced and time taken by map and reduce computations.

Comparison function

The input for each *Reduce* is pulled from the machine where the *Map* ran and sorted using the application's *comparison* function.

Reduce function

The framework calls the application's *Reduce* function once for each unique key in the sorted order. The *Reduce* can iterate through the values that are associated with that key and produce zero or more outputs. In the word count example, the *Reduce* function takes the input values, sums them and generates a single output of the word and the final sum.

Output writer

The *Output Writer* writes the output of the *Reduce* to the stable storage, usually a distributed file system.

3.4 Distribution and reliability

MapReduce achieves reliability by parceling out a number of operations on the set of data to each node in the network. Each node is expected to report back periodically with completed work and status updates. If a node falls silent for longer than that interval, the master node (similar to the master server in the Google File System) records the node as dead and sends out the node's assigned work to other nodes. Individual operations use atomic operations for naming file outputs as a check to ensure that there are not parallel conflicting threads running. When files are renamed, it is possible to also copy them to another name in addition to the name of the task (allowing for side-effects). The reduce operations operate much the same way. Because of their inferior properties with regard to parallel operations, the master node attempts to schedule reduce operations on the same node, or in the same rack as the node holding the data being operated on. This property is desirable as it conserves bandwidth across the backbone network of the datacenter.

Implementations are not necessarily highly reliable. For example, in older versions of cloud the *NameNode* was a single point of failure for the distributed filesystem. Later versions of cloud have high availability with an active/passive failover for the "NameNode."

Uses:

MapReduce is useful in a wide range of applications, including distributed pattern-based searching, distributed sorting, web link-graph reversal, term-vector per host, web access log stats, inverted index construction, document clustering, machine learning, and statistical machine translation. Moreover, the MapReduce model has been adapted to several computing environments like multi-core and many-core systems, desktop grids, volunteer computing environments, dynamic cloud environments, and mobile environments.

At Google, MapReduce was used to completely regenerate Google's index of the World Wide Web. It replaced the old *ad hoc* programs that updated the index and ran the various analyses.

MapReduce's stable inputs and outputs are usually stored in a distributed file system. The transient data is usually stored on local disk and fetched remotely by the reducers.

IV.CRITICISM

Lack of novelty

David DeWitt and Michael Stonebraker, computer scientists specializing in parallel databases and shared-nothing architectures, have been critical of the breadth of problems that MapReduce can be used for. They called its interface too low-level and questioned whether it really represents the paradigm shift its proponents have claimed it is. They challenged the MapReduce proponents' claims of novelty, citing Teradata as an example of prior art that has existed for over two decades. They also compared MapReduce programmers to Codasyl programmers, noting both are "writing in a low-level language performing low-level record manipulation." MapReduce's use of input files and lack of schema support prevents the performance improvements enabled by common database system features such as B-trees and hash partitioning, though projects such as Pig (or PigLatin), Sawzall, ApacheHive, YSmart, HBase and BigTable are addressing some of these problems. Greg Jorgensen wrote an article rejecting these views. Jorgensen asserts that DeWitt and Stonebraker's entire analysis is groundless as MapReduce was never designed nor intended to be used as a database. DeWitt and Stonebraker have subsequently published a detailed benchmark study in 2009 comparing performance of Clouds MapReduce and RDBMS approaches on several specific problems. They concluded that relational databases offer real advantages for many kinds of data use, especially on complex processing or where the data is used across an enterprise, but

that MapReduce may be easier for users to adopt for simple or one-time processing tasks.

Google has been granted a patent on MapReduce. However, there have been claims that this patent should not have been granted because MapReduce is too similar to existing products. For example, map and reduce functionality can be very easily implemented in Oracle's PL/SQL database oriented language.

Restricted programming framework

MapReduce tasks must be written as acyclic dataflow programs, i.e. a stateless mapper followed by a stateless reducer, that are executed by a batch job scheduler. This paradigm makes repeated querying of datasets difficult and imposes limitations that are felt in fields such as machine learning, where iterative algorithms that revisit a single working set multiple times are the norm.

V.ALGORITHM

In cloud computing, we have problem like security of date, files system, backups, network traffic, and host security. Here we are proposing a data security using encryption description with des algorithm while we are transferring it over the network.

The data encryption standard(DES) is the name of the federal information processing standard(FIPS),which describes the data encryption algorithm(DEA).the DES has been extensively studied since its publication and is the most widely used symmetric algorithm in the world. The DES has a 64-bit block size key during execution. DES is a symmetric cryptosystem, specifically a 16-round feistel cipher. When used for communication, both sender and receiver must know the same secret key, which can be used for communication, both sender and receiver must knokw the same secret key, which can be used to encrypt and decrypt the message, or to generate and verify a message authentication cod(MAC). The des can also be used for single- user encryption, such as to store files on a hard disk in encrypted form. The DES has a 64-bit block size and used a 56 bit key during execution.

In Cipher block chaining mode of operation of DES, each block of ECB encrypted ciphertext is XORed with the next plain text block to be encrypted, thus making all the blocks dependent on all the previous blocks, this means that in order ot find the plaintext of a particular blocks dependent on all the previous blocks, you need to know the ciphertext, the key and the cipher text for the previous block. The first block to be encrypted has no previous cipher text, so the plaintext is XORed with a 64-bit numb er called the initialization vector. so if data is transmitted over transmission error, the error will be carried forward to all the subsequent blocks since each block is dependent upon the last. This mode of operation is more secure than ECB(electronic code book) because the extra XOR step adds one more layer to the encryption process.

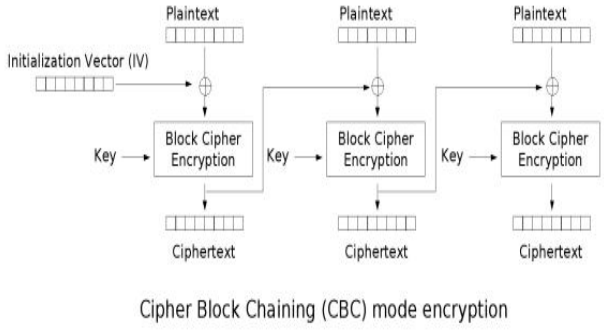

Figure 2.1

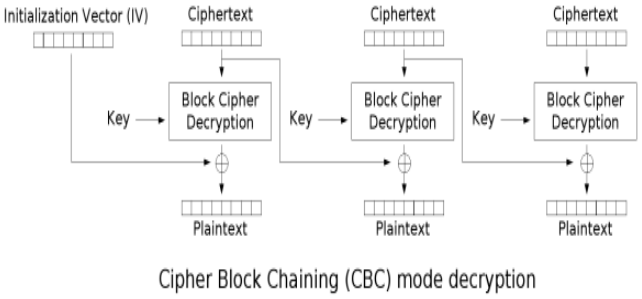

Figure 2.2

Compositions of Encryption and Decryption :

Encryption E = eL1 o eL2 …… o eL16

Decryption D = dL16 o dL15 o ……o dL1

Leader L is derived from the Password. Here we have 16 rotations. So we need 16 Leaders (L1 to L16) from Password.

L1 = First two bits of Password.

L2 = Second two bits of Password

L3 = Third two bits of Password and so on

Steps:

Get the Plaintext.

Get the Password.

Convert the Characters into binary form.

Derive the Leaders (L1 to L16) from the Password.

Apply the Formula to get the encrypted and decrypted message.

Encryption

x1 x2 x3

L y1 y2 y3

Decryption

y1 y2 y3

L x1 x2 x3

step. The meaning of the concepts is also taken into consideration, , by applying any linguistic analysis. Thus, it is important to note that input ontologies might have different label domains for node naming, without reducing the efficiency of the proposed methodology. The main advantages of the proposed framework can be summarized in terms of extensibility and flexibility. Other important future works are the possibility of modeling the antecedent and the consequent of an association rule as ontology concepts in order to express constraints on the association rules structure. Furthermore we could improve the system by integrating the constraints evaluation directly in the mining algorithm.

VI. CONCLUSION

In this paper we have analyzed the challenges and viability of deploying a computing cluster on top of a multi-cloud infrastructure. We also proposed a different data fragmentation schemes for multi cloud storage in cloud computing, which seeks to provide each customer with reliability, availability and better cloud data storage decisions. we proposed a secured cost-effective multicloud storage (SCMCS) in cloud computing, which seeks to provide each customer with a better cloud data storage decision, taking into consideration the user budget as well as providing him with the best quality of service (Security and availability of data) offered by available cloud service providers. Cloud computing security is still considered the major issue in the cloud computing environment. Customers do not want to lose their private information as a result of malicious insiders in the cloud. In addition, the loss of service availability has caused many problems for a large number of customers recently.

REFERENCES

1. Multi-Cloud Deployment of Computing Clusters for Loosely-Coupled MTC Applications Rafael Moreno Vozmediano, Ruben S. Montero, Ignacio M. Llorente (special issue on many task computing) july 2010.
2. B.AmarNadh Reddy, P.Raja Sekhar Reddy / International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622 www.ijera.com Vol. 2, Issue 5, September-October 2012, pp.1130-1134
3. Yashaswi Singh, Farah Kandah, Weiyi Zhang Department of Computer Science, North Dakota State University, Fargo, ND 58105 workshop on 2011
4. International Journal of Emerging Technology and Advanced Engineering Website: www.ijetae.com (ISSN 2250-2459, Volume 2, Issue 10, October 2012)
5. Harinder Kaur M.tech (C.S.E)(Persuing), B.Tech (C.S.E),Seth Jai Parkash Mukand Lal Institute Of Engineering and Technology, Radaur. Kurukshetra University, Kurukshetra, Haryana, India
6. http://en.wikipedia.org/wiki/Cloud_computing Cloud computing, Wikipedia.

7. 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. on Computer and communications security, 2007, pp. 598-609.

8. A. Bessani, M. Correia, B. Quaresma, F. André and P. Sousa, "DepSky: dependable and secure

storage in a cloud-of-clouds", EuroSys'11:Proc. 6thConf. on Computer systems, 2011, pp. 31-46.
9. K. Birman, G. Chockler and R. van Renesse,"Toward a cloud computing research agenda", SIGACT News, 40, 2009, pp. 68-80.
10 K.D. Bowers, A. Juels and A. Oprea, "HAIL: A high-availability and integrity layer for cloud storage", CCS'09: Proc. 16th ACM Conf. on Computer and communications security, 2009, pp. 187-198.
11. C. Cachin, R. Haas and M. Vukolic, "Dependable storage in the Intercloud", Research Report RZ, 3783, 2010.
12. C. Cachin, I. Keidar and A. Shraer, "Trusting the cloud", ACM SIGACT News, 40, 2009, pp. 81-86.

AUTHOR'S PROFILE:

1. M. Kruthika Reddy working as an iOS developer in Kofax India Pvt. Ltd, pursuing her Master of Technology(compter science and Technology) from GITAM UNIVERSITY, Rudraram. Her Research area includes Cloud Computing.

2. A Srinivas Reddy working as Asst. Professor in CSE dept. at GITAM University. I have completed my Master of Technology from National Institute of Technology Tiruchirappalli(NIT-Trichy) in Computer Science Engineering. My research area including Digital Electronics, image processing.