ISSN2394-3777 (Print)
ISSN2394-3785 (Online)
Available online atwww.ijartet.com
International Journal of Advanced Research Trends in Engineering and Technology
(IJARTET) Vol. 5, Special Issue 14, April 2018

OPTIMIZING THE OUTFLOW OF DATA IN STORAGE SERVICES

ASHIKA R[#1]
Dept of Computer Science and Engineering
C.Byregowda Institute of Technology
Kolar, India
MAHESH RAJ P N[#3]
Dept of Computer Science and Engineering
C.Byregowda Institute of Technology
Kolar, India
MOIN SHARIFF M[#4]
Dept of Computer Science and Engineering
C.Byregowda Institute of Technology
Kolar, India
NAZIYA SULTANA[#5]
Dept of Computer Science and Engineering
C.Byregowda Institute of Technology
Kolar, India
CHARANYA B[#2]
Dept of Computer Science and Engineering
C.Byregowda Institute of Technology
Kolar, India

ABSTRACT

Multi-cloud is being widely considered as the future of cloud computing. Distributing data over different Cloud Storage Providers (CSPs) automatically provides users with a certain degree of information leakage control, for no single point of attack can leak all the information. However, unplanned distribution of data chunks can lead to high information disclosure even while using multiple clouds. In this paper, we study an important information leakage problem caused by unplanned data distribution in multicloud storage services. Then, we present StoreSim, an information leakage aware storage system in multicloud. We design an approximate algorithm to efficiently generate similarity-preserving signatures for data chunks based on MinHash and Bloom filter. Next, we present an effective storage plan generation algorithm based on clustering for distributing data chunks with minimal information leakage across multiple clouds. We show that our scheme can reduce the information leakage by up to 60% compared to unplanned placement. Furthermore, our analysis on system attackability demonstrates that our scheme makes attacks on information more complex.

Keywords - StoreSim, MinHash, Bloom filter, System attackability, Multicloud storage, information leakage, ,remote synchronization and optimization.

I. INTRODUCTION

As the usage of devices such as laptops, cellphones and tablets is increasing rapidly, users wish for an ubiquitousand massive network storage to handle their ever-growing digital lives. To meet these demands, many cloud-based storage and file sharing services such as Dropbox, Google Drive and Amazon S3, have gained popularity due to the easy-to-use interface and low storage cost. However, these centralized cloud storage services are criticized for grabbing the control of users' data, which allows storage providers to run analytics for marketing and advertising . Also, the information in users' data can be leaked e.g., by means of malicious insiders, backdoors, bribe and coercion. One possible solution to reduce the risk of information leakage is to employ multicloud storage systems in which no single point of attack can leak all the information.CSPs such as Dropbox, among many others, employ rsync-like protocols to synchronize the local file to remote file in their centralized clouds. Every local file is partitioned into small chunks and these chunks are hashed with fingerprinting algorithms such as SHA-1, MD5. Thus, a file's contents can be uniquely identified by this list of hashes. For each update of local file, only chunks with changed hashes will be uploaded to the cloud. This synchronization based o on hashes is different from diff -like protocols that are based on comparing two versions of the same file line by line and can detect the exact updates and only upload these updates in a patch style. Instead, the hash-based synchronization model needs to upload the whole chunks with changed hashes to the cloud. Thus, in the multicloud environment, two chunks differing only very slightly can be distributed to two different clouds. The problem does not exist in a single storage cloud such as Dropbox since users have no other choice but to give all their information to only one cloud. When the storage is in the multicloud, we have the opportunity to minimize the total information that is leaked to each CSP. The optimal case is that each CSP obtains the same amount of information.

Therefore, we need more sophisticated techniques to detect the near duplicate (or similar) data chunks to reduce the information leakage in the multicloud storage system.

II. RELATED WORK

In this section, we will review some of the literature related to the four distinct pillars of our work, which are as follows:

Untrusted storage cloud:

We consider a mutlicloud in which each storage cloud is only served as storage without the ability to compute. The earlier previous work such as Cooperative File System (CFS) designed their storage system with a peer-to-peer network comprised of potentially untrusted nodes. Our work targets to use storage cloud without using decentralized P2P protocol and optimizes data placement in a centralized way.
This paper extends our work on StoreSim.

Multicloud storage services:

Our work is not in storing data with the adoption of multiple CSPs. Ourwork focuses on the information leakage optimization for storage service in a multicloud environment by exploiting information similarity caused by the synchronization of modified data. In StoreSim, we provide a user-specific weight for each cloud which not only coordinates the fraction of storage load for each cloud but also prevents the information leakage across the CSPs.

Cloud security:

Many studies focus on security and privacy aspects which are major obstacles of cloud adoption for both individuals and companies. Provided a survey for four different multicloud architectureswith various security and privacy-enhancing designs. The architecture of StoreSim is one of them, which allowsdistributing fine-grained fragments of the data to distinct clouds. Our work further implements the StoreSim system with new information leakage measures.

Near-duplicate detection:

We design our information leakage function based on similarity. To compute the information leakage, we need to compute the pairwise similarities. MinHash was designed for detecting the near-duplicate web pages based on. However, that work cannot apply to our work directly dueto heavy computation and high storage overhead.

III. STORAGE SERVICES OF MULTICLOUD

In this section, we first introduce multicloud storage services from the perspectives of both distribution and optimization. Then we discuss data synchronization mechanisms among three distributed entities in multicloud storage services.

A. Distribution and Optimization

Cloud storage services such as Dropbox and Google Drive, are centralized repositories for vast aggregations of personal data which can be monetized to afford the lowcost (or free) storage services for their users. While the users enjoy these storage services, they also lose their control on the data. Recent news about PRISM [6] shows that these CSPs can be compromised under coercion. Some other cloud storage services such as Wuala, SpiderOak employ clientside encryption to encrypt all the data before uploading the data. However, this does not change the inherent nature of centralized architecture. Even with encryption, once the encryption key is exposed , a user's entire data can be easily divulged. The situation can be somewhat alleviated by using multiple clouds services so that no single CSP has access to the user's entire data (encrypted or otherwise). Many works have been proposed in both academia and industry for using multiple CSPs for storing data. These works show that data distribution over multiple CSPs can avoid single point of failure, thereby improving the service availability and fault-tolerance. In addition, adopting multiple CSPs offers the opportunities for optimization on different metrics such as cost, network latency, service response time and vendor lock-in.

B. Data Synchronization Mechanism of Cloud Storage Services

In multicloud storage system, there are three distributed entities which synchronize users' data from the remote client to the cloud:

a) Client :

Usually clientis in charge of pre-processing the users' data for the purpose of optimization, such as chunking (i.e., dividing files into individual chunks of a maximum size data unit), deduplication (i.e., avoiding storing and re-transmitting the same content already available on the remote servers), deltaencoding (i.e., transmission of only modified portions of a file), bundling (i.e., the transmission of multiple small files as a single object) and encryption/decryption.

b) Metadata servers :

These are used to store the metadata database about the information of files, CSPs and users, which usually are structured data representing the whole cloud file system.

c) Storage servers :

Itstores the raw data blocks which can be both structured and unstructured data. The most essential step of data synchronization is to detect updates. One solution is diff-like protocols [30] which are based on comparing two versions of the same file line by line and can detect the exact updates. Only these updates will be uploaded to the cloud in a patch file which describes the difference between the old and the new version. However, diff-like protocols are not suitable for cloud storage services for three reasons. First, to compute the patch file, the client needs more storage overhead to store old versions, leading to the loss of users. Second, cloud storage services usually synchronize users' files across different clients and devices. If a file is modified in one client, then all other clients need to update both the old and the new version of this file, which results in high communication overhead.

The Cloud Storage Services will be in great danger if the client bears the burden of maintaining revision histories. For example, a mistake of deleting old versions made by users can result in synchronization errors.

In order to overcome the problems faced in diff-like protocols, employ rsync-like protocols [7] instead of using diff-like protocols, CSPs such as Dropbox, among many others. In order to synchronize the local file to remote file in their centralized clouds [8]. rsync-like protocols only require each client only storing the newest version and use signature based approach to detect updates. Specifically, every local file in the client is partitioned into small chunks and these chunks are hashed with fingerprinting algorithms such as SHA-1, MD5. In this way, a file's contents can be uniquely identified by this list of hashes and we call these hashes as signatures. To synchronize the updates to the cloud, the client will firstly send the signatures of current file to the metadata server. Then, the metadata server will detect these modified chunks by comparing current signatures with the signatures of last version and only returns the signatures of these changed chunks to the client. Finally, the client will only upload these chunks with changed signatures to the storage server. In this paper, we design our system based on rsync-like protocols and further optimize the system in terms of information leakage.

In the next section, we present StoreSim, an information leakage aware system for multicloud storage service with considering three distributed entities and their interactions.

IV. STORESIM

In this section, we firstly describe the architecture of StoreSim. Then we introduce StoreSim in terms of metadata and CSP models. Finally, we formulate the information leakage optimization problem in the multicloud.

A. Architecture

The architecture of StoreSim is shown in Figure. It can be observed that there is a trust boundary between the metadata and storage servers. We assume that clients and metadata servers, which are situated inside the trust boundary, are trustable by users while remote servers outside the boundary are untrustworthy.

For example, the metadata can be stored in private database servers while storage servers can be located public CSPs such as Amazon S3, Dropbox and Google Drive. Storage servers can be accessed through standard APIs (Application Programming Interfaces). As is shown in Figure 2, all control flows are inside the trust boundary while data flows can cross the trust boundary. In order to optimize the information leakage, we design two components in StoreSim. The first component is the Leakage Measure layer (LMLayer) that is used to evaluate the information leakage and further to generate the storage plan which maps data chunks to different clouds. The other component is the Cloud Manager layer (CMLayer) that provides cloud interoperability in a syntactic way. In the following, we will first present how we model metadata and storage cloud.

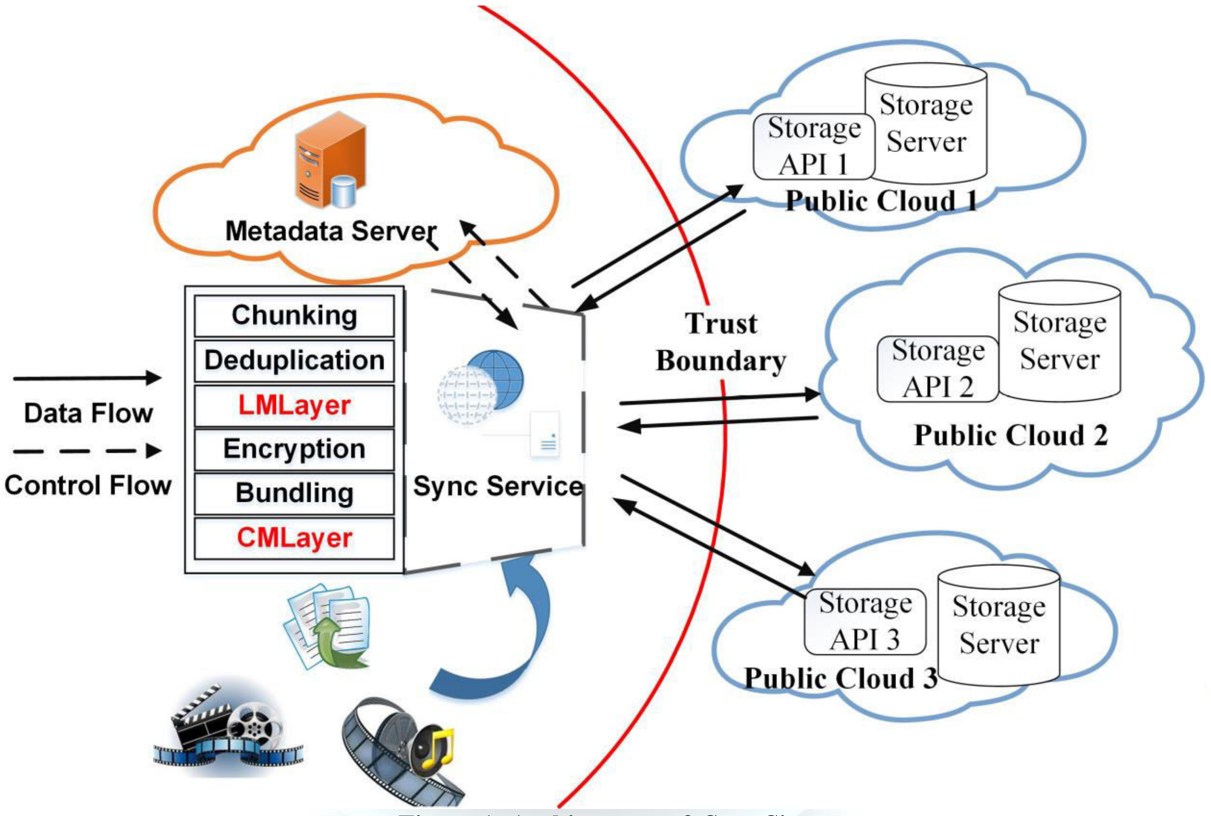

Figure 1: Architecture of StoreSim

B. MetaData Model

The data model we discuss in this section is for the metadata that represents the file system of StoreSim. Within the data graph, the vertices V represent different objects in a file system such as users, folders, files and data chunks. The edges E indicate a variety of relationships among different objects which can be distinguished by a set of labels . The labels also facilitate the process of path-oriented search, e.g., to find all data chunks of one file, or to find all the files of one user. In this paper, we do not focus on optimizing query performance of different data models. In practice, we may apply more techinques such as indexing, caching to improve query performance with the growth of the data volume. Furthermore, we define N _ V as the set of data nodes which store the raw data in G. We aim to distribute data nodes N to different CSPs in terms of storage protocols.

C. CSP Model

The cloud storage provider (CSP) model in our paper includes both user and system specific weights. User-specific weight to each cloud can be assigned either by StoreSim (the default) or by users in terms of their preferences, e.g., the fraction of data they want to store on a particular cloud, the trust that the user has in a CSP or the general reputation of the provider. Meanwhile, the system specific weight can be assigned by StoreSim toevaluate different CSPs in terms of cost, quota, network performance, etc. In this paper, we model user-specific weight as the storage load, i.e., the ratio of the total size of data stored on a cloud to the size of entire data of the user, while the system-specific weight is modelled as prior knowledge of a CSP, i.e., the set of data nodes whichhave been stored on it. Thus, the amount of prior knowledge of a CSP increases with the number of data nodes stored on it. We assume that the knowledge is unforgettable, i.e., the knowledge of a data node will not be removed even when the data node is removed from the cloud2. To sum up, a CSPs 2 S in StoreSim is parameterized by two factors < u; v > where u is a storage load factor while v indicates the prior knowledge of the CSP.

V. EFFICIENT MEASUREMENT OF PAIRWISE INFORMATION LEAKAGE

We define the pairwise information leakage as delta Jaccard similarity, as is shown in Equation 1. For each pair of data nodes (chunks), we convert the data nodes as sets of words and compute the Jaccard similarity. However, the set operations for measuring pairwise similarity can be quite expensive [15], even assuming small-sized chunks, given that the number of pairs increases quadratically as the number of chunks increases. Thus, we need an efficient algorithm to compute the Jaccard similarity with less computation and storage overhead. In the following, we first introduce the background of MinHash algorithm [8], [15], [16], which provides a fast way to compute Jaccard similarity, and explain why we cannot apply the existing approaches directly. Next we present BFSMinHash, a Bloom filter sketch for MinHash in order to reduce storage overhead.

A. MinHash Background

MinHash uses hashing to quickly estimate the Jaccard similarity of two sets, $J(S1, S2) = \left|\frac{S1 \cap S2}{S1 \cup S2}\right|$. It can be also interpreted as "the probability that a random element from the union of two sets is also in their intersection":$Prob\left[\min\left(h(S1)\right) = \min\left(h(S2)\right)\right] = \left|\frac{S1 \cap S2}{S1 \cup S2}\right|$where h is the independent hash function and min(h(S1)) gives the minimum value of h(x); $x \in S1$. Therefore, we can choose a sequence of hash functions h1, h2, _ _ _ , hk andMinHash signatures.

$$Sig1 = \{\min(h_i(S1)) \mid i = 1, \_\,\_\,\_, k\}$$
$$Sig2 = \{\min(h_i(S2)) \mid i = 1, \_\,\_\,\_, k\}$$

It follows that Jaccard similarity of two sets is approximated as |Sig1 ∩ Sig2| / k. However, MinHash with many hash functions needs to compute the results of multiple hash functions for every member of every set, which is computationally expensive. In our paper, we adopt a variant of Minhash which avoids the heavy computation by using only a single hash function. Instead of selecting only a single minimum value per hash function, the signature of Min-Hash with single hash function h will select the k smallest values from the set h(S), which is denoted as mink(h(S)). In this way, we have

$$Sig1 = \{\min_k(h(S1))\}$$
$$Sig2 = \{\min_k(h(S2))\}$$

Thus, a random sample of S1 ∪ S2 can be represented as:
$X = \{\min_k(h(S1 \cup S2))\} = \min_k(Sig1 \cup Sig2)$

The Jaccard similarity is estimated as $|X \cap Sig1 \cap Sig2|/k$. For Minhash algorithm, to compute the similarity for a pair of data nodes, we only need to store an array of MinHash signatures rather than storing the whole data. Although it reduces the storage cost greatly, it can still be heavy given the huge number of data nodes. Suppose that each hash function generates a signature of 64 bits and k is 64, the storage cost of each data node is about 512 bytes. If we have about two million chunks, the overhead of storing the signatures is 1 Gigabyte. Thus, we need a compact representation of these MinHash signatures to reduce the storage overhead. Previous work [2] proposed b-bit MinHash which only stores b lowest bits of each signature computed by different hash functions to reduce the storage space. However, this approach does not work for the MinHash with a single hash function since all the signatures are computed by the same hash function. Instead, we design BFSMinHash, a Bloomfilter sketching scheme for Minhash, which uses a single hash function. BFSMinHash exploits the space efficient feature of Bloom filter, thus reducing the storage overhead.

B. Bloom-Filter Sketch for MinHash

Similar to the fingerprints in data deduplication, we expect an algorithm to generate the signature with a relatively small and fixed size for each data node. Our proposed BFSMinHash algorithm employs a Bloom-filter with a single hash function to sketch MinHash signatures. Algorithm 1 shows three steps in BFSMinHash: shingling (line 1), fingerprinting (line 2-6) and sketching (line 7-11). The input is a byte stream of a data chunk and the output is a fix-sized similarity-preserving signature of this chunk.

Firstly, we convert each data chunk to a set of shingles which are contiguous subsequences of tokens. The process of shingling is to tokenize the byte stream into a set of shingles. For example, if the input is "abcde" and the size of a shingle is 2, the set of shingles is fab, bc, cd, deg. From this perspective, we only consider the similarity in a syntactic way [17] rather than in a semantic way. In other words , we do not consider the difference between the fruit apple and the company Apple. Then, for each shingle, we will compute its fingerprints by MinHash. We use a maximum heap with the fixed-size of k to save k smallest MinHash fingerprints for each data node. It only takes O(1) to get the maximum value of all k values in a maximum heap. Only when a new fingerprint is less than the maximum value stored in the heap, it will be added to the heap and the current maximum in the heap will be removed. From the shingling and fingerprinting steps, we can see that the time complexity of our algorithm is linear in the total length of data chunks. Finally, sketching based on Bloomfilter will convert the MinHash fingerprints into a fixed size signature. The Bloom filter is a space efficient data structure which can be used to test whether an element is in a set. However, when we adopt Bloom filter, we have to tolerate its effect of false positives. The rate of false positives is computed as $(1 - e^{-\frac{nk}{s}})^n$, where s is the size of Bloom filter, k is expected number of elements that will be added in Bloom filter and n is the number of hash functions [18]. For example, if we implement a Bloom filter with size of 512 bits and k is 64, the optimal number of hash functions is 1 with a false positive rate of 11.7%. In our case, we aim to keep the size of Bloom filter as small as possible and therefore the Bloom filter in our BFSMinHash algorithm always employs a single hash function. The final output of Algorithm 1 is a signature with the same size as the Bloom filter. In this way, computing similarity of two data nodes is converted to compute the similarity of two bloom filters. Given two signatures x; y, the Jaccard similarity is

$$J(x,y) = \frac{\sum_i (xi \wedge yi)}{\sum_i (xi \vee yi)} \qquad (1)$$

where xi; yi is the ith bit of x, y, and $\wedge, \vee$, are bitwise and, or operators respectively. Later we will evaluate approximate errors of BFSMinHash, which are caused by both MinHash and Bloom filter.

Algorithm 1 Bloom-Filter Sketch for MinHash

```
Input: byte[] chunk: byte stream of a data chunk
Output: byte[] signature
1: List<byte[]> shingles = ByteSegment(chunk,size);
2: maxHeap ← store k smallest values in a max heap
3: for each shingle : shingles do
4:         fingerPrint = hashFunction(shingle);
5:         maxHeap ← fingerPrint
6: end for
7: BloomFilter bf; //implement with a single hash function
8: for each fingerPrint : maxHeap do
9:         bf.add(fingerPrint);
10: end for
11: byte[] signature = bf.toByteArray();
12: return signature
```

VI. GENERATING MULTICLOUD STORAGE PLAN

Based on the pairwise information leakage measured by BFSMinhash algorithm, the next step is to generate the storage plan, with respect to the information leakage. Before we present our storage plan generation algorithm, we need to introduce a goodness function to quantify the quality of a storage plan.

A. Clustering for Storage Plan Generation

In Equation 3, Lmin needs to find the pairs with the minimal information leakage. This search problem is challenging when the number of pairs increases quadratically. Suppose we have 100,000 data nodes, the number of pairs will be as high as 5 billion. Thus, we need to design an efficient search algorithm to find data pairs with minimal information leakage.

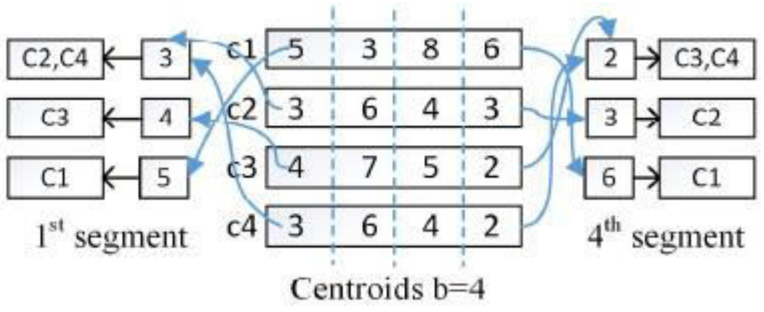

Figure. 2 :ClusterIndex for Centroids with b=4 Segments

Inspired by clustering problems, we propose a storage plan generation algorithm, SPClustering, to group similardata nodes. We define a data node as the centroid when no existing data node has low pair wise information leakage with it. In practice, we define a leakage threshold, according to which a data node becomes a centroid if all its pair wise information leakage with other nodes are greater than this threshold. In other words, a centroid represents all data nodes which are similar to it. Given any new data node, we only compute its pair wise similarities with a set of centroids, which largely reduces the number of pairs. Moreover, we build the ClusterIndex among the centroids to further prune the search space. A single index entry in Cluster Index points to a set of similar centroids, which is similar to the Bitmap index in traditional databases.

Algorithm 2 Generating Storage Plan based on Clustering

```
Input: N : a set of data nodes, S : a set of CSPs
Output: map ∈Mstorage plan
1: Build ClusterIndex for all centroids
2: for each x : N do
3:         for each s : S do
4:                 c = getCandidateSet(x, s) //pruning
5:                 $loss \leftarrow \frac{1}{|c|}\sum_{y\in c} Lp(x,y)$
6:         end for
7:          min_loss ← find s with minimal loss
8:         if min_loss > threshold then
9:                 assign x based on weights of CSPs
10:                add x as a centroid and build ClusterIndex for x
11:        end if
12:         map.put(x,s)
13: end for
14: return map
```

Specifically, suppose the size of signature generated by BFSMinHash algorithm is s bits, we divide the signature into b segments with the length of each segment as s/b. We will use each segment as the key in hash function and therefore, all the signatures with the same key will be hashed together.

For example, as is shown in Figure 2, when the key is the value of first segment, c2 and c4 are hashed to the same index entry for they share the same value of first segment. Those signatures are more likely to be similar to each other since they already share one same segment. Recall from Section, the number of elements sampled by BFSMinHash is k, which means its signature based on Bloom filter is at most with k bits set to one. If we cannot search any similar node from the ClusterIndex with b segments for a given node, that means there are at least b bits different from the given node with all the centroids. Based on Equation 1, it implies that there is no centroid that has Jaccard similarity with the given node larger than $(k-b)/(k+b)$. For example, if k is 64 and we divide the signature into 8 segments, the ClusterIndex can efficiently search all the similar centroids with similarity higher than 77.8%. Thus, in order to find centroids with less or more similarity, we need to respectively increase anddecrease the value of b (the number of segments). Algorithm 2 shows three main steps of how to generate a storage plan. Firstly, in the initialization, the algorithm. builds the ClusterIndex for a set of centroids online. We do not persist the ClusterIndex to reduce the storage overhead. The cost of building ClusterIndex is acceptable, which takes about 400 milliseconds for 100 thousand centroids. Then, we will find the cloud with the minimal information leakage based on candidate set for each new data node. The candidate set is queried based on ClusterIndex. Finally, if the minimal information leakage is still larger than the threshold,we will assign this node only based on the weights of CSPs. Also, the node will be labeled as the centroid and beindexed on the fly.

VII. DISCUSSION

In this part, we will discuss limitations of our StoreSim from four perspectives:

A. Syntactic vs Semantic

The information leakage function is designed based on syntactic similarity metric rather than semantic measures.Thus, our system is incapable of detecting the private data such as financial documents and compromising photos in a semantic manner. In future our work will focus on developing efficient algorithms of optimizing privacy in multicloud storage based on semantics. These algorithms can be incorporated in our StoreSim as plugin module.

B. Encryption vs StoreSim

Encryption is a process of converting information into a code to prevent from unauthorized access . However, if all the data are encrypted, users will not be ableto enjoy many other services provided by storage service providers such as file sharing or collaboration. Most of these services cannot operate over the ciphertext. In addition, StoreSim can also incorporate encryption after detecting near duplicate chunks and placing them together. Then, the information leakage can be reduced even if the encryption key is exposed.

C. CPU overhead

It is clear that the client in our system performs more additional work which introduces more computation. To reduce the CPU overhead, we choose MinHash algorithm with a single hash function and propose SPClustering with ClusterIndex as discussed before. In addition, we design BFSMinHash, which combines Bloom filters and MinHash algorithm, to further reduce the size of similarity-preserving fingerprints by 1/16. In StoreSim, there are four main components in the client: deduplication based on SHA-1 signature, LMLayer based on BFSMinhash and SPClustering, encryption/decryption based on AES-256 (same with that employed by SpiderOak [38]) and bundling based on ZIP.

D. Storage overhead.

The storage overhead depends on the Bloom filter size in BFSMinHash algorithm. In pratice, we set the Bloom filter size as 256 bits such that each chunk has the storage overhead of 32 Bytes. In other words, the storage overhead of 1GB data is around 64 KB if the chunk size is 512KB. Thus, the storage overhead is very low and constant (about 0.006% in our case). However, with the increase in the size of total data, the CPU overhead will also increase.

VIII. CONCLUSION

Certain degree of information may lead to leakage by distributing data on multiple cloud. However, information leakage is avoidable by unplanned distribution of data chunks. To optimize the information leakage, we presented the StoreSim, an information leakage aware storage system in the multicloud.. we demonstrate that StoreSim is both effective and efficient (in terms of time and storage space) in minimizing information leakage during the process of synchronization in multicloud. We show that our StoreSim can reduce information leakage upto 60%compared to unplanned placement and can achieve optimal performance. Finally, through our analysis, we further demonstrate that StoreSim not only reduces the risk of wholesale information leakage but also makes attacks on retail information much more complex.

REFERENCES

[1] G. Greenwald and E. MacAskill, “Nsa prism program taps in to user data of apple, google and others,” The Guardian, vol. 7, no. 6, pp. 1–43, 2013.
[2] P. Li and C. K¨onig, “b-bit minwise hashing,” in Proceedings of the 19th international conference on World wide web. ACM, 2010, pp. 671–680.
[3] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, “Sporc: Group collaboration using untrusted cloud resources.” in OSDI, vol. 10, 2010, pp. 337–350
[4] L. P. Cox and B. D. Noble, “Samsara: Honor among thieves in peer-to-peer storage,” ACM SIGOPS Operating Systems Review, vol. 37, no. 5, pp. 120–132, 2003..
[5] J. Crowcroft, “On the duality of resilience and privacy,” in Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, vol. 471, no. 2175. The Royal Society, 2015, p. 20140862.
[6] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, “Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services,” in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013, pp. 292–308.
[7] T. Li and N. Li, “On the tradeoff between privacy and utility in data publishing,” in Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2009, pp. 517–526
[8] M. Henzinger, “Finding near-duplicate web pages: a large-scale evaluation of algorithms,” in Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval. ACM, 2006,pp. 284–291.
[9] S. Choy, B. Wong, G. Simon, and C. Rosenberg, “A hybrid edge-cloud architecture for reducing on-demand gaming latency,” Multimedia Systems, pp. 1–17, 2014.
[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, “Wide-area cooperative storage with cfs,” in ACM SIGOPS Operating Systems Review, vol. 35, no. 5. ACM, 2001, pp. 202–215.
[11] H. Harkous, R. Rahman, and K. Aberer, “C3p: Context-aware crowdsourced cloud privacy,” in 14th Privacy Enhancing Technologies Symposium (PETS 2014), 2014.
[12] A. Bessani, M. Correia, B. Quaresma, F. Andr´e, and P. Sousa, “Depsky: dependable and secure storage in a cloud-of-clouds,” ACM Transactions on Storage (TOS), vol. 9, no. 4, p. 12, 2013.
[13] C. Gentry, “A fully homomorphic encryption scheme,” Ph.D. dissertation, Stanford University, 2009
[14] H. Zhuang, R. Rahman, and K. Aberer, “Decentralizing the cloud: How can small data centers cooperate?” in Peer-to-Peer Computing (P2P), 14-th IEEE International Conference on. Ieee, 2014, pp. 1–10.
[15] M. S. Charikar, “Similarity estimation techniques from rounding algorithms,”in Proceedings of the thiry-fourth annual ACM symposium on Theory of computing.
ACM, 2002, pp. 380–388.
[16] A. Rajaraman and J. D. Ullman, Mining of massive datasets. Cambridge University Press, 2011.
[17] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig, “Syntactic clustering of the web,” Computer Networks and ISDN Systems, vol. 29, no. 8, pp. 1157–1166, 1997.
[18] A. Broder and M. Mitzenmacher, “Network applications of bloom filters: A survey,” Internet mathematics, vol. 1, no. 4, pp. 485–509, 2004.