StoreSim: Optimizing Information Leakage in Multicloud Storage Services

Hao Zhuang*, Rameez Rahman*, Pan Hui†, Karl Aberer*
* LSIR, École Polytechnique Fédérale de Lausanne (EPFL)
† Hong Kong University of Science and Technology
hao.zhuang@epfl.ch, rameez.rahman@epfl.ch, panhui@cse.ust.hk, karl.aberer@epfl.ch

***Abstract*—Many schemes have been recently advanced for storing data on multiple clouds. Distributing data over different cloud storage providers (CSPs) automatically provides users with a certain degree of information leakage control, as no single point of attack can leak all user's information. However, unplanned distribution of data chunks can lead to high information disclosure even while using multiple clouds. In this paper, to address this problem we present *StoreSim*, an information leakage aware storage system in multicloud. StoreSim aims to store syntactically similar data on the same cloud, thus minimizing the user's information leakage across multiple clouds. We design an approximate algorithm to efficiently generate similarity-preserving signatures for data chunks based on MinHash and Bloom filter, and also design a function to compute the information leakage based on these signatures. Next, we present an effective storage plan generation algorithm based on clustering for distributing data chunks with minimal information leakage across multiple clouds. Finally, we evaluate our scheme using two real datasets from *Wikipedia* and *GitHub*. We show that our scheme can reduce the information leakage by up to 60% compared to unplanned placement.**

I. Introduction

With the increasingly rapid uptake of devices such as laptops, cellphones and tablets, users require an ubiquitous and 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 their 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 them to run analytics for marketing and advertising [1]. Also, the information in users' data can be leaked e.g., by means of malicious insiders, backdoors [2], bribes and coercion [3]. One possible solution to reduce the risk of information leakage is to employ multicloud storage systems [4, 5, 6, 7] in which no single point of attack can leak all the information. A malicious entity, such as the one revealed in recent attacks on privacy [3], would be required to coerce all the different CSPs on which a user might place her data, in order to get a complete picture of her data. Put simply, as the saying goes, do not put all your eggs in one basket.

Yet, the situation is not so simple. CSPs such as Dropbox, among many others, employ *rsync*-like protocols [8] 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* [9]. 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 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 [8]. 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 following motivating example will show that if chunks of a user's data are assigned to different CSPs in an unplanned manner, the information leaked to each CSP can be higher than expected. Suppose that we have a storage service with three CSPs S_1, S_2, S_3 and a user's dataset D. All the user's data will be firstly chunked and then uploaded to different clouds. The dataset D is represented as a set of hashes generated by each data chunk. This scenario is shown in Figure 1. In addition, we consider that the data chunks are distributed to different clouds in a round robin (RR) way. Apparently, RR is good for balancing the storage load and each cloud thus obtains the same amount of data. However, *the same amount of data does not necessarily mean the same amount of information*. For example, if we find that the set of chunks $\{C3, C6, C9\}$ are almost same, it means S_3 actually obtains the information equivalent to that in only one chunk. If all other chunks are different, S_1 and S_2 obtain three times as much information as S_3, even though all of them obtain the same amount of data. 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. In our example, data distribution based on RR can achieve the optimal result only if all the chunks are different. However this is not the case in cloud storage service due to two reasons: 1) Frequent modifications of files by users result in large amount of similar chunks[1]; and 2) Similar chunks across files, due to which existing CSPs use

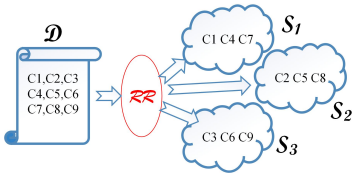

Fig. 1: The motivating example

[1]Most CSPs maintain revision history.

the data deduplication technique.

Determining identical chunks is relatively straightforward but efficiently determining similarity between chunks is an intricate task due to the lack of similarity preserving fingerprints (or signatures). At the same time, similarity is of paramount importance if one wants to limit information disclosure. Put simply, two paragraphs of text with one word different would lead to two different chunks. If one were to only consider identity, the two chunks would be considered different and placed separately; however both of them contain almost entirely the same information, hence they should ideally be placed together. We note here that the above problem is relevant even with encryption because once the encryption key is exposed (as in coercion of the CSP by some third party such as the National Security Agency or due to the maliciousness of the CSP itself), the entire data of the user can be easily leaked. If encryption is performed after detecting near duplicate chunks and placing them together, then the information leakage can be reduced even if the encryption key is exposed. Therefore, we need more sophisticated techniques to detect the near-duplicate (or similar) data chunks to reduce information leakage in the multicloud storage system.

Through the above example, we can see that storing the data in a multicloud system without proper optimization on the data distribution can lead to avoidable information leakage. In this paper, we focus on reducing information leakage to each individual CSP in a multicloud storage system and provide mechanisms for distributing users data over multiple CSPs in a leakage aware manner. Specifically, we make the following contributions in this paper:

- We present *StoreSim*, an information leakage aware multicloud storage system and formulate information leakage optimization problem in multicloud (Section II).
- We propose an approximate algorithm, *BFSMinHash*, based on Minhash and Bloom filter to generate similarity-preserving signatures for data chunks. We also design a pairwise information leakage function based on Jaccard similarity (Section III).
- Based on the information leakage measured by BFS-MinHash, we develop an efficient storage plan generation algorithm, *SPClustering*, for distributing users data to different clouds (Section IV).
- Finally, we use two datasets crawled from *Wikipedia* and *GitHub*, containing files with multiple revisions, to evaluate our framework. Through extensive experiments, we show the effectiveness and efficiency of our proposed scheme for reducing information leakage across multiple clouds (Section V).

II. 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 2. 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

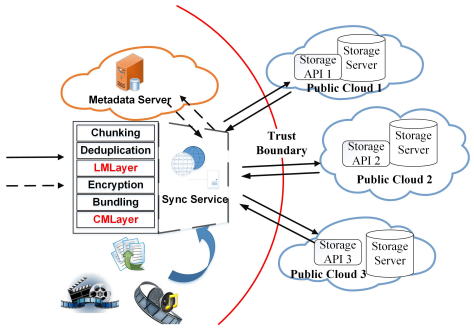

Fig. 2: Architecture of StoreSim

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 in 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 storage plan which maps data chunks to different clouds. The other component is Cloud Manager layer (CM-Layer) that provides cloud interoperability in a syntactic way.

B. Models

MetaData model. The data model we discuss in this section is for the metadata that represents the file system of StoreSim. We model users' data as a labeled graph $\mathcal{G} =< \mathcal{V}, \mathcal{E}, \Omega, \pi >$ where $\mathcal{V}$ is a set of vertices, $\mathcal{E}$ is a set of edges, Ω is a set of labels, and $\pi : \mathcal{V} \cup \mathcal{E} \rightarrow \Omega$ is a function that assigns labels to vertices and edges. Within the data graph, the vertices $\mathcal{V}$ represent different objects in a file system such as users, folders, files and data chunks. The edges $\mathcal{E}$ indicate a variety of relationships among different objects which can be distinguished by a set of labels Ω. 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. Furthermore, we define $\mathcal{N} \subseteq \mathcal{V}$ as the set of data nodes which store the raw data in $\mathcal{G}$. We aim to distribute data nodes $\mathcal{N}$ to different CSPs in terms of the storage protocol defined in Section II-C.

CSP model. A cloud storage provider (CSP) $s \in \mathcal{S}$ is parameterized by two factors $< u, v >$ where u is a *storage load* factor while v indicates the *prior knowledge* of a CSP. 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, can be assigned either by StoreSim (the default) or by users in terms of their preferences. The prior knowledge of a CSP is modeled as the set of data nodes which have 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 cloud[2].

C. Storage Protocol

In essence, the storage protocol is a set of constraints or cost functions to reduce the information leakage on data

[2]This is a necessary assumption since providers such as Dropbox also do not actually delete user's data immediately.

distribution across multiple clouds. The protocol in StoreSim is to store similar chunks on the same cloud, thereby reducing information leakage to each individual CSP. In the following, we firstly define information leakage for a pair of data nodes.

Definition 1: (Pairwise Information Leakage). Given a set of data nodes $\mathcal{N}$ in data graph $\mathcal{G}$, we define $\mathcal{L}_p : \mathcal{N} \times \mathcal{N} \rightarrow \mathbb{R}$ as the pairwise information leakage function. For any pair of data nodes $n_i, n_j \in \mathcal{N}, \mathcal{L}_p(n_i, n_j)$ computes the pairwise information leakage of two nodes.

$\mathcal{L}_p$ can measure the information leakage in terms of syntactic or semantic way. In this paper, we only compute the information leakage based on syntactic similarity. In this paper, we use delta Jaccard similarity of two sets, as our information leakage function (other appropriate similarity functions can also be employed depending on the data structure and application requirements).

$$\mathcal{L}_p(n_i, n_j) = J_\Delta(\sigma(n_i), \sigma(n_j)) = 1 - |\frac{\sigma(n_i) \cap \sigma(n_j)}{\sigma(n_i) \cup \sigma(n_j)}| \quad (1)$$

where the function $\sigma(\cdot)$ will convert a data node into a set (i.e., representing a data node as a set of words). It follows immediately that if a CSP gets a new data node that is a duplicate of an existing data node on that cloud (i.e., Jaccard similarity between them is 1), there will be no leakage due to lack of new information. In addition, we define the information leakage of the first data node stored in a cloud as a constant $\mathbf{1}$. It can be interpreted that all the information in the first data node at a CSP is leaked.

Furthermore, we also define a storage plan as a mapping from data nodes to different CSPs, which is defined as:

Definition 2: (Storage Plan). Given a set of data nodes $\mathcal{N}$ in the data graph $\mathcal{G}$ and a set of CSPs $\mathcal{S}$, a storage plan $\mathcal{M} : \mathcal{N} \rightarrow \mathcal{S}$ is a mapping of each data node $n \in \mathcal{N}$ to a CSP $s \in \mathcal{S}$.

The storage plan can be generated in terms of users' preference and QoS factors. For example, the storage plan based on round robin makes for a good balance of the storage load among different CSPs. In our paper, we will evaluate the goodness of storage plan with respect to the information leakage. The goodness of storage plan is defined as:

Definition 3: (Goodness of Storage Plan). Given a set of data nodes $\mathcal{N}$ in the data graph, a pairwise information leakage function $\mathcal{L}_p$ and a storage plan $\mathcal{M}$, we define $G_\mathcal{M}(\mathcal{N}, \mathcal{M}, \mathcal{L}_p) \in \mathbb{R}$ as the goodness function of storage plan $\mathcal{M}$.

From the Definition 3, we can see that the goodness of storage plan depends on the pairwise information leakage function $\mathcal{L}_p$ and the storage plan $\mathcal{M}$. Thus, an interesting question is whether there exists an optimal storage plan with respect to a given information leakage measure. We can formulate this information leakage optimization problem as:

Definition 4: (Information Leakage Optimization Problem). Given a set of data nodes $\mathcal{N}$ in the data graph, a pairwise information leakage function $\mathcal{L}_p$ and a storage plan $\mathcal{M}$, the information leakage optimization problem is to find the optimal storage plan with minimal information leakage $\mathcal{M}^* = \underset{\mathcal{M}}{\operatorname{argmin}}\, G_\mathcal{M}(\mathcal{N}, \mathcal{M}, \mathcal{L}_p)$

In this paper, we provide an approximate algorithm for addressing this problem. We first discuss how to efficiently measure pairwise information leakage in Section III and then in Section IV we propose a storage plan that places similar chunks together in a multicloud environment.

III. 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 [10], 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, 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 [10, 11] uses hashing to quickly estimate the Jaccard similarity of two sets which can be also interpreted as "the probability that a random element from the union of two sets is also in their intersection", $Prob[min(h(S_1)) = min(h(S_2))] = |\frac{S_1 \cap S_2}{S_1 \cup S_2}| = J(S_1, S_2)$ where h is the independent hash function and $min(h(S_1))$ gives the minimum value of $h(x), x \in S_1$. Therefore, we can choose a sequence of hash functions $h_1, h_2, \cdots, h_k$ and compute the minimum values of each hash function as MinHash signatures, i.e., $Sig(S) = \{min(h_i(S))|i = 1, \cdots, k\}$. It follows that Jaccard similarity of two sets is approximated as $|Sig(S_1) \cap Sig(S_2)|/k$. 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 MinHash with single hash function h will select the k smallest values from the set $h(S)$, which is denoted as $Sig(S) = \{min_k(h(S))\}$. Thus, a random sample of $S_1 \cup S_2$ can be represented as $X = \{min_k(h(S_1 \cup S_2))\} = min_k(Sig(S_1) \cup Sig(S_2))$. The Jaccard similarity is estimated as $|X \cap Sig(S_1) \cap Sig(S_2)|/k$.

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 [12] 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 Bloom-filter 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. There are three steps in BFSMinHash: *shingling*, *fingerprinting* and *sketching*.

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 {ab, bc, cd, de}. From this perspective, we only consider the similarity in a syntactic way [13] 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 Bloom-filter 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 $(1 - e^{-nk/s})^n$, 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 [14]. 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 BFSMinHash algorithm 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

$$J(x,y) = \frac{\sum_i (x_i \wedge y_i)}{\sum_i (x_i \vee y_i)} \tag{2}$$

where x_i, y_i 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, in Section V.

IV. Generating Multicloud Storage Plan

Based on the pairwise information leakage measured by BFSMinhash algorithm, the next step is to generate the storage plan with the minimal 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. Goodness of Storage Plan

The goodness function of storage plan is evaluated based on the pairwise information leakage, as it is defined in Definition 1. Recall from Equation 1, the pairwise information leakage measures how much new information will be leaked when a pair of data nodes are stored in the same cloud. Thus, it is essential to find the pairs of data nodes with minimal information leakage. In order to measure the goodness of a storage plan, we introduce a metric called *relative information leakage* (RIL), which is defined as the average of minimal pairwise information leakage among all the data nodes in a storage plan. For example, in our motivating example in Section I, cloud S_2 stores three data nodes for a total of $\binom{3}{2}$ pairs, $\{(C2, C5), (C5, C8), (C2, C8)\}$. Suppose $\{\mathcal{L}_p(C2, C5) = 0.25,\ \mathcal{L}_p(C5, C8) = 0.15, \mathcal{L}_p(C2, C8) = 0.1\}$, we have the information leakage of first data node $C2$ as constant $\mathbf{1}$ while the minimal pairwise information leakage for $C5, C8$ is 0.15 and 0.1, respectively. Thus, the RIL of data nodes stored in S_2 is the average minimal pairwise information leakage $(1 + 0.15 + 0.1)/3 = 0.416$. Formally, given an individual CSP $s_i = (u_i, v_i) \in S$ in a storage plan $\mathcal{M}$, the RIL of all data nodes stored in s_i is formulated as:

$$RIL_i = \frac{1}{|v_i|}(\mathbf{1} + \sum_{l=2}^{|v_i|} \mathcal{L}_{min}(n_l, n_k)), \tag{3}$$

$$s.t. \quad v_i = \{n \in \mathcal{N} | \mathcal{M}(n) = s_i\}, \tag{4}$$

$$l \neq k, n_l, n_k \in v_i \tag{5}$$

where $\mathbf{1}$ is the information leakage for the first data node and $\mathcal{L}_{min}(n_l, n_k)$ returns the minimal pairwise information leakage, $\mathcal{L}_p(n_l, n_k)$, for n_l by searching the node $n_k, l \neq k$, which is most similar to it. v_i in Equation 4 represents prior knowledge and is modeled as the set of data nodes stored in s_i. Since we have $\mathcal{L}_p \in [0, 1]$, it follows that $RIL_i \in [\frac{1}{|v_i|}, 1]$. In the extreme case where all the data nodes stored in a CSP are the same, the RIL is $\frac{1}{|v_i|}$, which means the actual information it has obtained equals to the information of one node. In other words, a good storage plan, which can effectively detect the similar chunks and distribute them to the same cloud, has a low RIL value. Based on this, we can compute the RIL for a storage plan $\mathcal{M}$ as the weighted average of the RILs of all CSPs:

$$RIL_{\mathcal{M}} = \sum_{i=1}^{|\mathcal{S}|} u_i * RIL_i \tag{6}$$

where u_i is the normalized storage loads of CSPs such that $\sum_{i=1}^{|\mathcal{S}|} u_i = 1$. In this way, the information leakage optimization problem with respect to RIL is to find an optimal storage plan with minimal relative information leakage to each CSP.

B. Clustering for Storage Plan Generation

In Equation 3, $\mathcal{L}_{min}$ 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 $\binom{100,000}{2}$. Thus, we need to design an efficient

search algorithm to find data pairs with minimal information leakage.

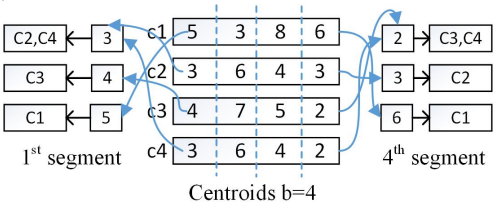

Fig. 3: ClusterIndex for Centroids with b=4 Segments

Inspired by clustering problems [15], we propose a storage plan generation algorithm, *SPClustering*, to group similar data nodes. We define a data node as the *centroid* when no existing data node has low pairwise information leakage with it. In practice, we define a leakage threshold, according to which a data node becomes a centroid if all its pairwise 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 pairwise 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 ClusterIndex points to a set of similar centroids, which is similar to the Bitmap index in traditional databases [16]. 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 3, 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 III-B, 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 2, 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 and decrease the value of b (the number of segments).

To further generate a storage plan, we firstly builds the ClusterIndex for a set of centroids on the fly. 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, for each new data node, we will find the cloud with the minimal information leakage based on the candidate set which is queried based on ClusterIndex. Finally, if the minimal information leakage of new node is still larger than the leakage threshold, we will assign this node only based on the storage loads of CSPs. Meanwhile, this node will be labeled as the centroid and be indexed on the fly.

V. Experimental Evaluation

In this section, we first introduce the implementation of StoreSim and the two datasets used for evaluation. Then we evaluate the performance of two algorithms, BFSMinHash and SPClustering. Finally, we analyze the time cost introduced by the leakage measure layer in StoreSim.

A. Implementation

We have implemented the StoreSim prototype using Java, and it includes both basic components (such as chunking, data deduplication, bundling and encryption/decryption), and featured components including LMLayer and CMLayer. In the LMLayer, we implement the algorithms described in the previous sections, while the CMLayer enables StoreSim to communicate with multiple CSPs. StoreSim employs the common fixed-size chunking with a maximum chunk size of 512 KB. The chunk is identified by *SHA-1* signature, which is also used for data deduplication. The small chunks can be bundled as a ZIP file to minimize the network transmission overhead. Succinctly, before the chunk is synchronized, it can be measured for leakage optimization, encrypted, and bundled for better network transmissions. The synchronization of StoreSim is based on the delta encoding [8], which only synchronizes changed chunks (identified by SHA-1 signatures) between two copies. All the metadata, which is organized as data graph, are stored in a MySQL database. We have implemented for three public storage clouds: Dropbox, Google Drive, and Amazon S3. All the communications between StoreSim and public CSPs occur using APIs supplied by those CSPs. We also support the synchronization of files to the local FTP servers. The metadata server is deployed on our local server machine and the evaluation is conducted on a personal client machine with Intel i7-2640M CPU and 4GB memory.

B. Dataset

For the evaluation, we aim to find such data which has undergone several modifications, and thus results in many similar chunks. This can serve as a model for the modifications that users make in the cloud storage services. Wikipedia and Github are two such data sources that contain web pages and files which are reviewed and modified multiple times. Thus, we crawled two datasets from Wikipedia and Github, respectively. The Wikipedia dataset contains a total of 2197 web pages and each web page has a maximum 49 revisions. For each web page, the crawler only stores the text that is extracted from HTML files. The total size of the dataset is 1.2 GB. The size of each webpage is relatively small, which ranges from 29 Bytes to 118 KB with an average size of 11KB. The Github dataset contains the United States code[3] spanning 56 files. The files in this dataset are much larger than those in the Wikipedia dataset, in the range of 47.7KB to 50MB with an average size of 5.3 MB. The files in this dataset have a maximum of 8 modifications and the total dataset size is 2.1 GB. Thus, we observe that the data chunks generated by Wikipedia dataset are small in size with maximum chunk size of 118 KB, but great in number (91,929) while those generated by Github dataset are bigger in size with maximum size of 512KB but are less in number (4,274).

C. BFSMinHash

In this part, we will evaluate the performance of BFSMinHash algorithm in terms of approximation errors and effectiveness.

Approximation Errors. We implement BFSMinHash based on 64 bits *Murmur* hash function [17] and thus each

[3]https://github.com/divegeek/uscode

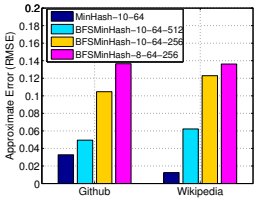

Fig. 4: Approximate Errors (MinHash-10-128 as baseline)

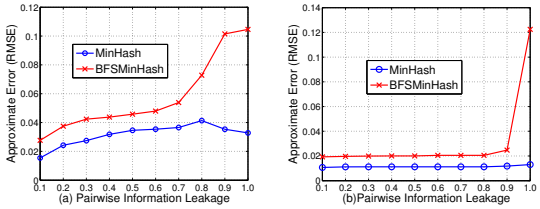

Fig. 5: Approximate Errors by groups for (a) Github and (b)Wikipedia

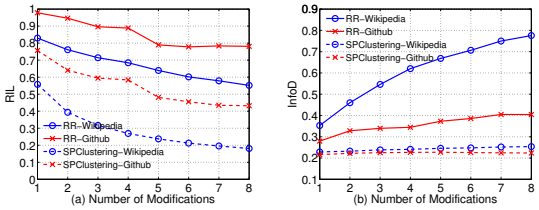

Fig. 6: Effect of modification numbers on (a) RIL and (b) InfoD

MinHash signature is 64 bits. In BFSMinHash, we have to further fix three parameters: the shingle size l, the sampling size of MinHash k and the Bloom filter size s. We seek to determine the suitable parameters for BFSMinHash for our subsequent experiments, by first comparing the performance among five settings of our algorithm: 1) MinHash-10-128; 2) MinHash-10-64; 3) BFSMinHash-10-64-512; 4) BFSMinHash-10-64-256; 5) BFSMinHash-8-64-256, where the numbers in the name correspond to the value of l, k and s (only for BFSMinHash). Among different settings, MinHash-10-128 theoretically has the best performance since it has the largest sampling and shingle size and no sketch. Given our goal is to evaluate the approximation error between our BFSMinHash and MinHash algorithm, we select MinHash-10-128 as the baseline and compare its performance with that of the other four algorithms. From Figure 4, we can see that MinHash-10-64 without sketching outperforms the other three for both datasets. The sampling size of MinHash-10-64 is 64, i.e., it will select 64 smallest MinHash signatures. The storage cost for each chunk is 64*64 bits = 512 Bytes. We can observe that BFSMinhash algorithms with the shingle size of 10 are better than that with the shingle size of 8. This is because a longer shingle size decreases the probability of a given shingle appearing in any document. In addition, we also observe that approximate error is influenced by the ratio of the sampling size to the Bloom filter size. As is shown in Figure 4, the performance of BFSMinhash-10-64-512 is better than BFSMinHash-10-64-256 by about 6%. However, the storage cost of BFSMinHash-10-64-256 (32 Bytes per chunk) is only half of BFSMinHash-10-64-512 (64 Bytes per chunk).

Considering the tradeoff between storage cost and approximate errors, in StoreSim we adopt the setting of BFSMinHash-10-64-256. We observe that overall approximate errors of BFSMinHash-10-64-256 are about 10.4% for Wikipedia and 12.2% for Github. Thus, an immediate question is that are these approximation errors of BFSMinHash-10-64-256 tolerable to find the pairs with the minimal information leakage? We will answer this question in the next group of experiments. In the following, we use BFSMinHash to refer to BFSMinHash-10-64-256 while MinHash refers to MinHash-10-64.

Effectiveness of BFSMinhash. In the last experiment, we evaluated approximate errors based on all the pairs. In fact, the primary goal of our algorithm is to identify those pairs which have minimal information leakage and put them in the same cloud. Thus, we are more interested in approximate errors of the pairs with the minimal information leakage. Put bluntly, we are interested in those pairs of nodes whose information leakage is low, say 0.3, rather than those whose information leakage is very high, say 0.8, since these do not serve our needs of placing similar nodes on the same CSPs.

Therefore, in this set of experiments, we divide pairs into ten groups in terms of their pairwise information leakage, where the first group is all the pairs with information leakage less than 0.1 while the second group is set of pairs with information leakage less than 0.2, and so on. The approximate errors of different groups are shown in Figure 5. It is interesting to discover that the performance of BFSMinhash is highly close to the MinHash algorithm for groups 1-7 of Github dataset and groups 1-9 of Wikipedia dataset. For the Github dataset, the performance of BFSMinhash degraded dramatically after the information leakage is larger than 0.7 while for the Wikipedia dataset, the performance of BFSMinhash remains stable till the information leakage is 0.9. The dramatic increase in approximation errors of the pairs with large information leakage means that our algorithm is not very accurate for the pairs with low similarities. However, as stated earlier, in practice, we are targeted to identify the pairs with low information leakage (or high similarity). Therefore, we can safely state that our BFSMinhash algorithm is effective enough to meet our demands since it identifies pairs with information leakage as high as 0.7 with low error. To conclude and to answer the question raised by the last set of experiments, the results clearly show that our BFSMinHash is almost as effective as MinHash in identifying the pairs with the minimal information leakage while it can reduce the storage cost to *1/16* of MinHash.

D. SPClustering

In this part, we will evaluate the performance of our storage plan generation algorithm SPClustering. Besides the metric of RIL as defined in Section IV-A, we further define a new metric, *information density* (InfoD) from the perspective of entire dataset. The InfoD of a CSP is defined as the ratio of the information it has stored to the entire information in the whole dataset. Given a CSP $s_i = (u_i, v_i) \in S$, we further denote the set of data nodes which are also centroids stored in s_i as v_i^c and the InfoD of s_i is computed as $InfoD_i = \frac{|v_i^c|}{\sum_{j=1}^{|S|} |v_j^c|}$ where $\sum_{j=1}^{|S|} |v_j^c|$ denotes the total number of centroids in a dataset. We approximate the total information in a dataset to that information in its centroids since the centroid represents all data nodes which are similar to it. Base on this, the InfoD of a storage plan $\mathcal{M}$ for a dataset is computed as the weighted average InfoD of each CSP, $InfoD_{\mathcal{M}} = \sum_{i=1}^{|\mathcal{S}|} u_i * InfoD_i$. For example, consider all CSPs with equal normalized storage loads of $\frac{1}{|\mathcal{S}|}$. Here the optimal case of storage plan ensures that $InfoD = \frac{1}{|\mathcal{S}|}$, with every cloud obtaining $\frac{1}{|\mathcal{S}|}$ of total information, (i.e., $InfoD_{\mathcal{M}} = \frac{1}{|\mathcal{S}|}$); while the worst case is

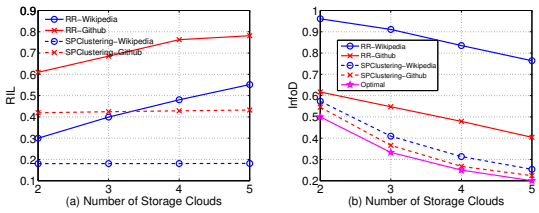

Fig. 7: Effect of CSP numbers on (a) RIL and (b) InfoD

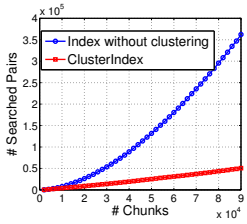

Fig. 8: Pruning efficiency by ClusterIndex

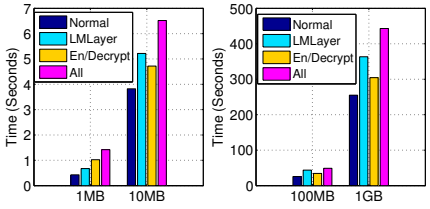

Fig. 9: Time cost with different configurations by varying file size

$InfoD_{\mathcal{M}} = 1$ with every cloud obtaining all the information in dataset. Thus, we can see that the higher the InfoD is, the more information are leaked to each SDC. In the following, we will evaluate the goodness of storage plan generated by our SPClustering to answer two questions: 1) What's the impact of user's modifications of data on the information leakage? 2) Is there any effect on the number of CSPs on which the users distribute their data?

Number of modifications. In this set of experiments, we have five CSPs with equal storage loads. After each modification, the dataset will be synchronized to clouds using delta encoding. The more modifications a dataset has undergone, the more resultant similar chunks. Figure 6 shows the influence of number of modifications on information leakage of storage plans generated by both SPClustering and Round Robin (RR) algorithms. It clearly shows that SPClustering outperforms RR greatly for both the Wikipedia and Github datasets. From Figure 6 (a), we can observe that with the increase in number of modifications, the RIL of SPClustering decreases, by about 30% (from the first modification to the last), much more quickly than RR, which decreases by only about 16%. The decrease of RILs implies that modifications on a dataset brings about more similar chunks. Our SPClustering algorithm is much more effective than RR to place those similar data chunks with the minimal information leakage in the same cloud. In Figure 6 (b), we can observe that RR without clustering the similar data nodes leaks the information in the dataset quickly, for the infoDs of RR increase to about 80% and 40% for Wikipedia and Github, respectively. Recall that the number of data chunks in Wikipedia dataset is much larger than that in Github, which also brings about much more similar data chunks. Thus, under RR without optimization on data chunks distribution, we can observe that Wikipedia leaks information much quicker than Github. On the other hand, InfoDs of our approach almost remains stable (around 22%, 25% for Github and Wikipedia, respectively), which indicates that our approach prevents the information leakage effectively. The reader may recall that the files in Wikipedia dataset have undergone a maximum of 49 modifications while that of Github a maximum of 8 modifications. Thus, we only compare the first 9 versions of Wikipedia dataset to that of whole Github dataset in Figure 6. If we evaluate the whole Wikipedia dataset with 49 modifications, the final RILs of Wikipedia decreases to 9.7% while InfoDs of Wikipedia increase to 31%. Thus, we can conclude that our approach greatly prevents information leaked in the process of data synchronization.

Number of CSPs. In this set of experiments, we fix the number of modifications to 8 and vary the number of CSPs from 2 to 5. All CSPs in the experiments have the same storage load. In Figure 7(a), we can see that the RILs of SPClustering are almost stable for both datasets while those of RR increase steadily. The stable RIL implies that SPClustering algorithm is effective to prevent information leakage by putting the pairs with the minimal information leakage together regardless of the number of CSPs. As for RR, with the increase in number of CSPs, the probability of putting data pairs with minimal information leakage in the same cloud decreases, thereby leading to increased RIL. In Figure 7(b), we can observe that InfoDs of our approach decreases as the number of CSPs increases. This is the benefit of multicloud environment and the more CSPs there are, the less the data obtained by each cloud. However, we can observe from Figure 7(b) that a CSP under RR without considering information leakage can obtain more than 70% of entire Wikipedia dataset and 40% of Github dataset even with a total of 5 CSPs while SPClustering can achieve near-optimal value with respect to InfoD. For all cases, we observe that SPClustering improves the InfoD by about 60% and 45% for Wikipedia and Github respectively, compared to RR, which means SPClustering prevents about 60% of total information in Wikipedia from being leaked.

ClusterIndex. Finally, we evaluate the pruning efficiency of ClusterIndex employed by SPClustering algorithm. We vary the number of data nodes in ClusterIndex from 2000 to 90,000 in Wikipedia dataset and compare the performance with that of indexing without considering clustering. We only evaluate based on Wikipedia dataset since the number of data chunks in Github is limited. As is shown in Figure 8(b), ClusterIndex can reduce the number of searched pairs by 86% without much tradeoff on the precision (about 2.6%, not shown in the figure).

E. Discussion

CPU overhead. It is clear that the client in our system performs more additional work which introduces more computation. 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[18]) and bundling based on ZIP. We evaluate the overhead introduced by LMLayer in terms of four configurations: 1) *Normal*: deduplication and bundling; 2) *LMLayer*: deduplication, LMLayer and bundling; 3) *En/Decrypt*: deduplication, encryption/decryption and bundling; 4) *All*: all together. We compare the time cost by varying the size of files from 1MB to 1GB and Figure 9 shows the results. The time cost starts from dividing input files into small chunks and ends with assembling chunks to the original file. The *En/Decrypt* mode has an additional overhead since it has to decrypt the chunks before assembling. We discover that for small files of size less than 10MB, the overhead introduced by LMLayer is almost the same as the *En/Decrypt* mode. Especially in the case of 1MB, the performance of *LMLayer* is better than that of *En/Decrypt* mode. We conjecture this is because compared to *En/Decrypt* mode which needs key setup, there is no initialization overhead

for measuring information leakage. For the large files (both 100MB and 1GB), the overhead of *LMLayer* is higher than that of *En/Decrypt*. In all cases, we notice that even in the *All* mode with all components running, the time cost is still tolerable for cloud storage services.

Syntactic vs Semantic. In our paper, the information leakage function is designed based on syntactic similarity metric rather than semantic. Thus, our system is incapable of detecting the private data such as financial documents and compromising photos in a semantic manner. Distributing data based on semantic privacy measures is an orthogonal task to ours, since efficiently analyzing semantic similarity in users' data involves data curation and machine learning techniques.

VI. Related Work

Multicloud storage services. Our work is not alone in storing data with the adoption of multiple CSPs, e.g., SPANStore [7], DepSky [4] and NCCloud[5]. However, these works focus on different issues such as cost optimization [7], data consistency and availability [4]. Unlike these works, our work focuses on the information leakage optimization for storage service in a multicloud environment by exploiting information similarity caused by the synchronization of modified data. Supplementary works invest efforts in overcoming vendor lock-in. DepSky [4] aims to minimize the cost of data transfer from one cloud to another by storing only a fraction of the total amount of data in each cloud, while Scalia [6] employs data replication to minimize transfer cost albeit at a higher storage cost. Other studies [9, 19] have focused on measurement analysis of cloud storage services. Their work provided us with many insights on designing and implementing StoreSim. However, they did not focus on the optimization aspects of information leakages of the commercial CSPs they studied.

Cloud security. Many studies [20, 21, 22] focus on security and privacy aspects which are major obstacles in cloud adoption for both individuals and companies. Previous work [21] proposed a semantic framework based on crowd-sourcing to determine the sensitivity of items and diverse attitudes of users towards privacy. Bohli *et al.* [20] provide a survey for four different multicloud architectures with various security and privacy-enhancing designs. The architecture of StoreSim is one of them, which allows distributing fine-grained fragments of the data to distinct clouds. Our work also implements the StoreSim system with new information leakage measures.

Near-duplicate detection. Li *et al.* [23] proposed a privacy loss measure based on the JS-divergence distance which is a method of measuring the similarity between two probability distributions. Inspired by their work, we design our information leakage function based on similarity. To compute the information leakage, we need to compute the pairwise similarities. MinHash [11, 12] and SimHash [11, 24] were designed for detecting the near-duplicate web pages based on Jaccard and Hamming distance, respectively. However, their work cannot apply to our work directly due to heavy computation and high storage overhead. To the best of our knowledge, ours is the first work which applies near-duplicate techniques for preventing information leakage in multicloud storage services.

VII. Conclusion

Distributing data on multiple clouds provides users with a certain degree of information leakage control in that no single cloud provider is privy to all the user's data. However, unplanned distribution of data chunks can lead to avoidable information leakage. In this paper, we presented StoreSim, an information leakage aware storage system, to optimize the information leakage in the multicloud environment. StoreSim achieves this goal by using novel algorithms, BFSMinHash and SPClustering, which place the data with minimal information leakage (based on similarity) on the same cloud. Through an extensive evaluation based on two real datasets, 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 a multicloud environment.

Acknowledgment

This research is funded by the EU project CloudSpaces: Open Service Platform for the Next Generation of Personal clouds (FP7-317555).

References

[1] 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, p. 20140862, The Royal Society, 2015.

[2] "Prism surveillance program by nsa." http://en.wikipedia.org/wiki/Edward_Snowden#Disclosure.

[3] G. Greenwald and E. MacAskill, "Nsa prism program taps in to user data of apple, google and others," *The Guardian*, 2013.

[4] A. Bessani, M. Correia, B. Quaresma, F. André, 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.

[5] H. Chen, Y. Hu, P. Lee, and Y. Tang, "Nccloud: A network-coding-based storage system in a cloud-of-clouds," 2013.

[6] T. G. Papaioannou, N. Bonvin, and K. Aberer, "Scalia: an adaptive scheme for efficient multi-cloud storage," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 20, IEEE Computer Society Press, 2012.

[7] 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*, pp. 292–308, ACM, 2013.

[8] T. Suel and N. Memon, "Algorithms for delta compression and remote file synchronization," 2002.

[9] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking personal cloud storage," in *Proceedings of the 2013 conference on Internet measurement conference*, pp. 205–212, ACM, 2013.

[10] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pp. 380–388, ACM, 2002.

[11] 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*, pp. 284–291, ACM, 2006.

[12] P. Li and C. König, "b-bit minwise hashing," in *Proceedings of the 19th international conference on World wide web*, pp. 671–680, ACM, 2010.

[13] 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.

[14] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, 2004.

[15] P. Berkhin, "A survey of clustering data mining techniques," in *Grouping multidimensional data*, pp. 25–71, Springer, 2006.

[16] C.-Y. Chan and Y. E. Ioannidis, "Bitmap index design and evaluation," in *ACM SIGMOD Record*, vol. 27, pp. 355–366, ACM, 1998.

[17] "Murmur hash function." https://sites.google.com/site/murmurhash/.

[18] "Spideroak encryption specification." https://spideroak.com/engineering_matters#encryption.

[19] Z. Ou, H. Zhuang, A. Lukyanenko, J. K. Nurminen, P. Hui, V. Mazalov, and A. Yla-Jaaski, "Is the same instance type created equal? exploiting heterogeneity of public clouds," *Cloud Computing, IEEE Transactions on*, vol. 1, no. 2, pp. 201–214, 2013.

[20] J.-M. Bohli, N. Gruschka, M. Jensen, L. L. Iacono, and N. Marnau, "Security and privacy-enhancing multicloud architectures," *Dependable and Secure Computing, IEEE Transactions on*, vol. 10, no. 4, pp. 212–224, 2013.

[21] H. Harkous, R. Rahman, and K. Aberer, "C3p: Context-aware crowdsourced cloud privacy," in *14th Privacy Enhancing Technologies Symposium (PETS 2014)*, 2014.

[22] I. Ion, N. Sachdeva, P. Kumaraguru, and S. Čapkun, "Home is safer than the cloud!: privacy concerns for consumer cloud storage," in *Proceedings of the 7th Symposium on Usable Privacy and Security*, ACM, 2011.

[23] 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.

[24] G. S. Manku, A. Jain, and A. Das Sarma, "Detecting near-duplicates for web crawling," in *Proceedings of the 16th international conference on World Wide Web*, pp. 141–150, ACM, 2007.