Optimizing Information Leakage in Multicloud Storage Services

Hao Zhuang, *Member, IEEE,* Rameez Rahman,Pan Hui, *Member, IEEE,* and Karl Aberer, *Member, IEEE*

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, 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. 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. Furthermore, our analysis on *system attackability* demonstrates that our scheme makes attacks on information more complex.

Index Terms—Multicloud storage, information leakage, system attackability ,remote synchronization, distribution and optimization

1 Introduction

1.1 Motivation and Challenges

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 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 [1]. 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 [2], [3], [4], [5] 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 [6], 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 the eggs in one basket.

Yet, the situation is not so simple. CSPs such as Dropbox, among many others, employ *rsync*-like protocols [7] to synchronize the local file to remote file in their centralized clouds [8]. 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 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 [7]. 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

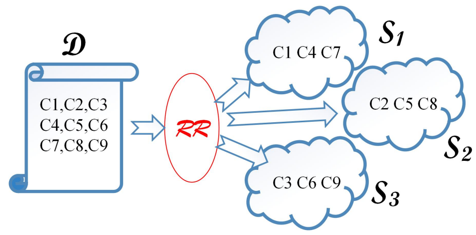

Fig. 1. The motivating example

• *H. Zhuang, R. Rahman and K. Aberer are with Distributed Information Systems Laboratory (LSIR), École Polytechnique Fédérale de Lausanne (EPFL), Switzerland*
E-mail: hao.zhuang@epfl.ch, rameez.rahman@epfl.ch, karl.aberer@epfl.ch
• *Pan Hui is with Department of Computer Science and Engineering, Hong Kong University of Science and Technology.*
E-mail: panhui@cse.ust.hk

Manuscript received Jan 19, 2016; revised Jan 26, 2017.

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 the data deduplication technique.

In fact, the data deduplication technique, which is widely adopted by current cloud storage services like Dropbox, is one example of exploiting the similarities among different data chunks to save disk space and avoid data retransmission [8], [9]. It identifies the same data chunks by their fingerprints which are generated by fingerprinting algorithms such as *SHA-1, MD5*. Any change to the data will produce a very different fingerprint with high probability [10]. However, these fingerprints can only detect whether or not the data nodes are duplicate, which is only good for *exact equality* testing. 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 the information leakage in the multicloud storage system.

1.2 Approach and Contributions

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. First we provide a novel algorithm for generating similarity preserving signatures for data chunks. Next based on this algorithm, we devise a chunk placement storage plan that efficiently synchronizes similar chunks together in a multicloud environment. Finally, we evaluate and validate our design using real datasets. Specifically, we make the following contributions in this paper:

- We present *StoreSim*, an information leakage aware multicloud storage system which incorporates three important distributed entities and we also formulate information leakage optimization problem in multicloud.
- 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.
- Based on the information leakage measured by BFSMinHash, we develop an efficient storage plan generation algorithm, *SPClustering*, for distributing users data to different clouds.
- 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. Furthermore, our analysis on the system attackability demonstrates that StoreSim makes attacks on information much more complex.

The rest of this paper is organized as follows. In section 2, we review existing literature related to our work and explain why our work is different. In section 3, we introduce multicloud storage services and discuss data synchronization mechanisms among three interacting entities in the multicloud. In section 4, we present the architecture, models and storage protocols of our StoreSim. Section 5 details the BFSMinHash algorithm to generate similarity-preserving signatures for data chunk while section 6 presents the SPClustering algorithm to distribute users data to different clouds with respect to optimizing information leakage. In section 7, we discuss our experiments results and limitations of StoreSim. Finally, section 8 concludes our work.

2 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: Depot [11] and SPORC [12] assumed that the storage clouds are unstrusted and fault-prone black boxes. However, both their work employed only a single cloud which has both compute and storage capacity. Our work is different since 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) [13] and Samsara [14] 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 [15] and optimizes data placement in a centralized way. This paper extends our work on StoreSim [16].

Multicloud storage services. Our work is not alone in storing data with the adoption of multiple CSPs, e.g., SPANStore [5], DepSky [2] and NCCloud [3]. However,

1. Most CSPs maintain revision history.

these work focused on different issues such as cost optimization [5], data consistency and availability [2] and service response time [17]. Other efforts [18] on the cloud orchestration provided deployment plans in terms of the tradeoff between price and performance. 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 d ata. S upplementary e fforts o n o vercoming vendor lock-in, DepSky [2] minimized 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 [4] employed the data replication at a higher storage cost. However, in StoreSim, we provide a user-specific w eight f or e ach cloud which not only coordinates the fraction of storage load for each cloud but also prevents the information leakage across the CSPs. Other studies have focused on measurement analysis of cloud storage services [8], [9]. Their work provided us with many insights on designing StoreSim. But their work failed to reveal optimization aspects of information leakages of the commercial CSPs they studied.

Cloud security. Many studies [19], [20], [21] focus on security and privacy aspects which are major obstacles of cloud adoption for both individuals and companies. Previous work [20] proposed a semantic framework based on crowd-sourcing to determine the sensitivity of items and diverse attitudes of users towards privacy. Bohli *et al.* [19] provided 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 f ragments o f t he d ata t o distinct clouds. Our work further implements the StoreSim system with new information leakage measures.

Near-duplicate detection. Li *et al.* [22] 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 [23], [24] and SimHash [23], [25] 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, this is the first w ork w hich applies near-duplicate techniques for preventing information leakage in multicloud storage services.

3 Multicloud Storage Services

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.

3.1 Distribution and Optimization

Cloud storage services such as Dropbox and Google Drive, in essence, are centralized repositories for vast aggregations of personal data which can be monetized to afford the low-cost (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 client-side encryption to encrypt all the data before uploading the data. However, this does not change the inherent nature of centralized architecture. As discussed previously, even with encryption, once the encryption key is exposed (e.g., by leveraging backdoors in the key-generation software [26], by compromising the insiders who know the key or by compromising the devices that stores the key), 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 [2], [3], [4], [5] and industry [27], [28], [29] 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. Unlike the previous work, we focus on the optimization of information leakage in multiple storage services (against single point of attack).

3.2 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:

1) **Client** is 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), delta encoding (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;

2) **Metadata servers** 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;

3) **Storage servers** store 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. Last but not least, cloud storage services will be in great

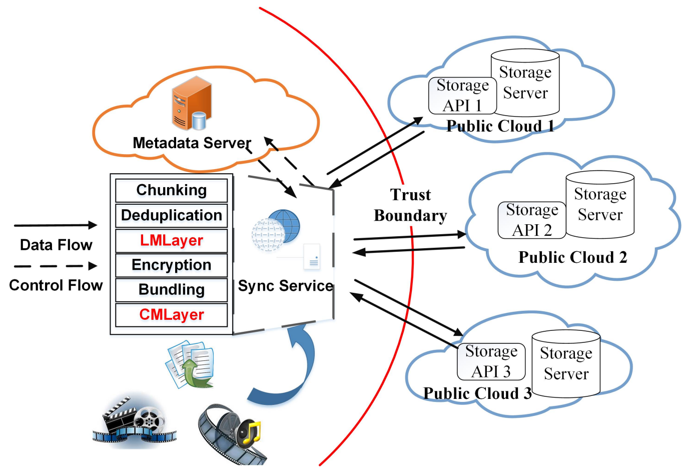

Fig. 2. Architecture of StoreSim

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.

Instead of using *diff-like* protocols, CSPs such as Dropbox, among many others, employ *rsync*-like protocols [7] 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.

4 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.

4.1 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 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 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.

4.2 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. As is shown in the table 1, we summarize all notations in this paper. 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. 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 $\mathcal{N} \subseteq \mathcal{V}$ 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 storage protocols.

4.3 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 to evaluate different CSPs in terms of cost, quota, network performance, etc. In this paper, we

Notation	Semantics
$\mathcal{G}$	a labeled data graph for metadata, *i.e.*, file system
$\mathcal{V}$	vertices set in a data graph
$\mathcal{E}$	edges set in a data graph
Ω	a set of labels
π	a function that assigns labels to edges
$\mathcal{N} \subseteq \mathcal{V}$	vertics that are data nodes
$\mathcal{L}$	pairwise information leakage function given two data nodes
$\mathcal{S}$	a set of storage clouds
$\mathcal{M}$	a storage plan that maps data nodes to storage clouds
n_i	a data node i
s_i	a storage cloud i

TABLE 1
Notations

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 modeled as *prior knowledge* of a CSP, i.e., 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]. To sum up, a CSP $s \in \mathcal{S}$ 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.

4.4 Storage Protocol

In essence, the storage protocol is a set of constraints or cost functions to reduce the information leakage on data distribution across multiple clouds. Motivated by our example in Section 1.1, 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 I nformation L eakage).* G iven a s et of data nodes $\mathcal{N}$ in data graph $\mathcal{G}$, we define $\mathcal{L}_p : \mathcal{N} \times \mathcal{N} \to \mathbb{R}$ 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 either syntactic or semantic way. For example, the information leakage can be measured as the dissimilar information (i.e., new information) based on similarity measures or as the information gain based on entropy measures. In this paper, we only model information leakage based on syntactic similarity. Specifically, we use delta Jaccard similarity of two sets, as our information leakage function [3].

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

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 1.0), there will be no leakage due to lack of new information. In the opposite case, the first data node stored in a cloud is a totally new node, which we define the information leakage of the first data node as a constant $\mathbf{1}$. It can be interpreted that all the information in the first data node at a CSP is leaked.

In addition, 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 $\mathcal{M} : \mathcal{N} \to \mathcal{S}$ 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 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 5 and then in Section 6 we propose a storage plan that places similar chunks together in a multicloud environment.

5 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 [31], 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 [23], [31], [32], 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.

5.1 MinHash Background

MinHash uses hashing to quickly estimate the Jaccard similarity of two sets, $J(S_1, S_2) = |\frac{S_1 \cap S_2}{S_1 \cup S_2}|$. It 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

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

3. Other appropriate similarity functions can also be employed depending on the data structure and application requirements

compute the minimum values of each hash function as MinHash signatures.

$$Sig_1 = \{min(h_i(S_1))|i = 1, \cdots, k\}$$
$$Sig_2 = \{min(h_i(S_2))|i = 1, \cdots, k\}$$

It follows that Jaccard similarity of two sets is approximated as $|Sig_1 \cap Sig_2|/k$. 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 $min_k(h(S))$. In this way, we have

$$Sig_1 = \{min_k(h(S_1))\}$$
$$Sig_2 = \{min_k(h(S_2))\}$$

Thus, a random sample of $S_1 \cup S_2$ can be represented as:

$$X = \{min_k(h(S_1 \cup S_2))\} = min_k(Sig_1 \cup Sig_2)$$

The Jaccard similarity is estimated as $|X \cap Sig_1 \cap Sig_2|/k$. 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 [24] 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 s ketching s cheme f or M inhash, w hich u ses a single hash function. BFSMinHash exploits the space efficient feature of Bloom filter, thus reducing the storage overhead.

5.2 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 s ize f or e ach d ata n ode. O ur 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 {ab, bc, cd, de}. From this perspective, we only consider the similarity in a syntactic way [33] 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 [34]. 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

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

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 7.

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: $maxHeap \leftarrow$ store k smallest values in a max heap
 3: for each shingle : shingles do
 4:     $fingerPrint$ = hashFunction(shingle);
 5:     $maxHeap \leftarrow fingerPrint$
 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
```

6 Generating Multicloud Storage Plan

Based on the pairwise information leakage measured by BFSMinhash algorithm, the next step is to generate the storage plan, as shown in Definition 2, 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.

6.1 Goodness of Storage Plan

The goodness function of storage plan is evaluated based on the pairwise information leakage, as it is defined in Definition 4. 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 t he p airs o f d ata n odes 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 1.1, 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 user-specific weights 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.

6.2 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.

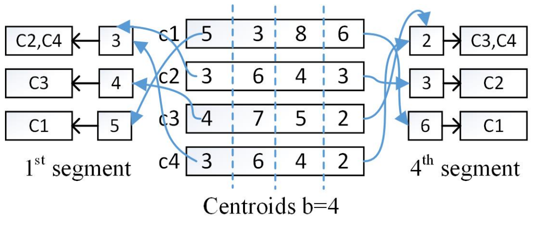

Fig. 3. ClusterIndex for Centroids with b=4 Segments

Inspired by clustering problems [35], 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 [36]. 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 5.2, 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).

Algorithm 2 shows three main steps of how to generate a storage plan. Firstly, in the initialization, the algorithm

Algorithm 2 Generating Storage Plan based on Clustering

```
Input: $\mathcal{N}$ : a set of data nodes, $\mathcal{S}$ : a set of CSPs
Output: $map \in \mathcal{M}$ storage plan
1:  Build ClusterIndex for all centroids
2:  for each $x : \mathcal{N}$ do
3:      for each $s : \mathcal{S}$ do
4:          $c$ = getCandidateSet($x$, $s$) //pruning
5:          $loss \leftarrow \frac{1}{|c|} \sum_{y \in c} \mathcal{L}_p(x, y)$
6:      end for
7:      $min\_loss \leftarrow$ 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
```

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 be indexed on the fly.

7 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.

7.1 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. All those components are optional, i.e., users can opt to use the LMLayer and encryption at the same time or they can disable any of them. StoreSim is also pluggable, i.e., StoreSim provides APIs for developers to add their modules for different encryption or information leakage measures. For example, StoreSim employs the BFSMinHash algorithm but developers can replace it with the SimHash algorithm based on Hamming distance [25]. 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 *rsync*-like protocols [7], which only synchronizes the new chunks (identified by SHA-1 signatures) between two copies. All the metadata, which is organized as data graph, are stored in a MySQL database. To deal with the heterogeneity in the different data models of different clouds, StoreSim employs CMLayer to enable cloud interoperability in a syntactic way. CMLayer provides the uniform APIs such as initialize, connect, upload, download and delete, for different CSPs. With this abstraction, the user can move their data across different CSPs in a transparent way, thereby alleviating vendor lock-in problems. We have implemented for three public storage clouds: Dropbox, Google Drive, and Amazon S3. All the communications between StoreSim and public CSPs are 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.

7.2 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[4] 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. The size of the dataset is controlled around 2GB since the free quota of personal storage service such as Dropbox is 2GB. Through these two datasets, we simulate two different use cases. 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).

7.3 BFSMinHash

In this part, we will evaluate the performance of our BFSMinHash algorithm and answer two questions:

- what's the approximation errors and effectiveness of our proposed BFSMinHash?
- Is the information leakage measured by BFSMinHash trustable?

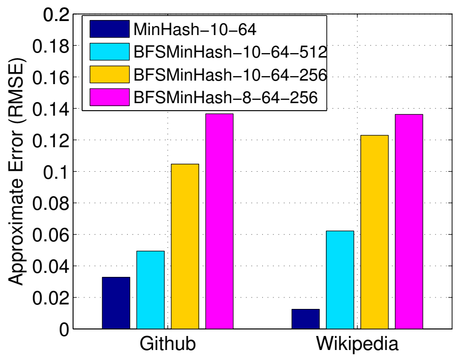

Fig. 4. Effect of parameters with MinHash-10-128 as baseline

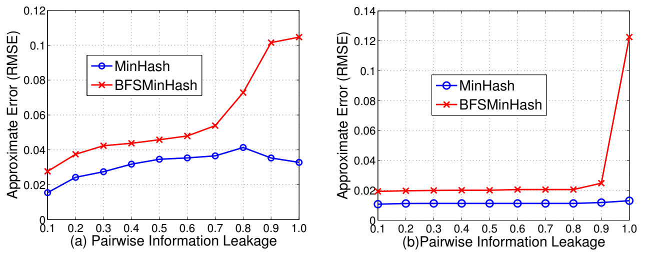

Fig. 5. Approximate Errors by groups for (a) Github and (b)Wikipedia

Approximation Errors. We implement BFSMinHash based on 64 bits *Murmur* hash function [37] and thus each 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. The approximation error of MinHash algorithm is caused only by sampling and shingle size while that of our BFSMinHash is due to all the parameters. The larger the size of sampling and Bloom filter, the closer that our algorithm approximates to the actual Jaccard similarity. However, we have to balance the tradeoff between approximation errors and storage cost. 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. In addition, to evaluate the performance of different algorithms, we define approximate error as the root mean square error (RMSE) between the result of MinHash-10-128 and that of the algorithm under comparison, i.e.,

$$RMSE = \sqrt{\frac{\sum_{t=1}^{n}(\hat{y}_t - y_t)^2}{n}}.$$

In this part, we randomly select two small sample datasets from Wikipedia and Github datasets due to huge number of pairs present in the original datasets. We select 400 pages from Wikipedia and 30 files from Github in which each page or file in these samples has 4 revisions. After chunking by StoreSim, the number of data chunks for both samples is around 1600. Thus, the total number of pairs is around 1,279,200 ($\binom{1600}{2}$). 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

4. https://github.com/divegeek/uscode

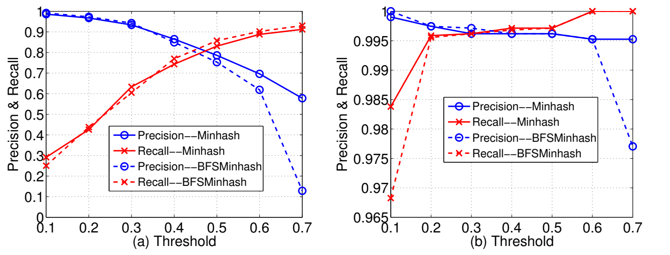

Fig. 6. Precision and recall by varying threshold for (a) Github and (b) Wikipedia

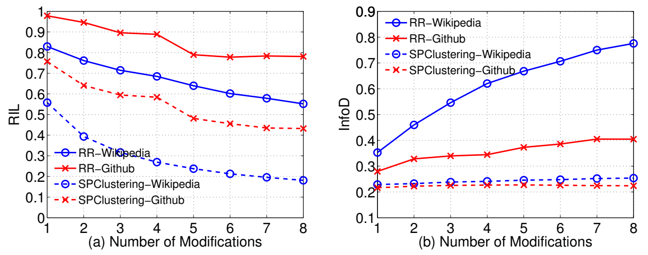

Fig. 7. Effect of modification numbers on (a) RIL and (b) InfoD

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.

Is the measured information leakage trustable? Given the pairwise information leakage computed based on our BFSMinhash algorithm, another question is that to what extent we can trust those values. To answer this question, we have to obtain the ground truth of pairwise information leakage. Instead of using Human evaluation [25], we compute pairwise similarities among all the data chunks using the Gensim library[5] for calculating cosine similarity between documents based on TF-IDF weights. We assume that the ground truth of pairwise information leakage is highly close to the result computed by Gensim algorithm[6]. Based on the Gensim result, we can generate the *relevant set*, which is denoted as R, as the set of pairs whose information leakage is less than 0.2. In the next step, we will query a set of pairs from the result generated by our BFSMinHash algorithm, denoted as *search set* S, with information leakage threshold ranging from 0.1 to 0.7. i.e., we query all the pairs with information leakage less than 0.1, all the pairs with information leakage less than 0.2, and so on. Therefore, we can calculate the precision, i.e., the fraction of searched pairs that are relevant to all the searched pairs, and recall, i.e., the fraction of the searched pairs that are relevant to all the relevant pairs, based on the search set S and relevant set R : $Prescion = \frac{|S \cap R|}{|S|}$, and $Recall = \frac{|S \cap R|}{|R|}$.

Figure 6 shows the results of both datasets. It clearly shows the tradeoff of precision and recall under different information leakage thresholds. For the Github dataset, choosing threshold as 0.4 achieves both good precision(0.85) and recall(0.75). To our surprise, for the Wikipedia dataset, both the precision and recall can be as high as 99.5% when we set the threshold to 0.3. That means the search set generated by our BFSMinhash algorithm is almost the same as the relevant set which is generated by Gensim algorithm. Therefore, we can safely reach the conclusion that the information leakage computed by BFSMinHash algorithm can be almost as good as that of GenSim by choosing an appropriate threshold. In this case, we can set the threshold as 0.4 for both datasets, which is the point in both Figure 6(a) and (b) where a good balance between precision and recall is achieved. Furthermore, we can also observe from Figure 6, that the performance of BFSMinhash and Minhash are similar for low thresholds. This further confirms the effectiveness of our BFSMinHash algorithm in identifying the data pairs with low information leakage.

7.4 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 6.1, 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 s_i^c and the InfoD of s_i is computed as:

$$InfoD_i = \frac{|v_i^c|}{\sum_{j=1}^{|S|} |v_j^c|} \tag{7}$$

where $\sum_{j=1}^{|S|} |v_j^c|$ denotes the total number of centroids in a dataset. From Equation 7, 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:

$$InfoD_{\mathcal{M}} = \sum_{i=1}^{|\mathcal{S}|} u_i * InfoD_i \tag{8}$$

For example, consider all CSPs with equal normalized weights 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 $InfoD = 1$ with every cloud obtaining all the information in the dataset, (i.e., $InfoD_{\mathcal{M}} = 1$). 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 in terms of both RIL and InfoD to answer three questions:

- What's the impact of user's modifications of data on the information leakage?

5. http://radimrehurek.com/gensim/
6. We note that we cannot apply Gensim to StoreSim since it is too time expensive. For the given data, it took us more than 24 hours to compute the information leakage for all pairs on a server machine.

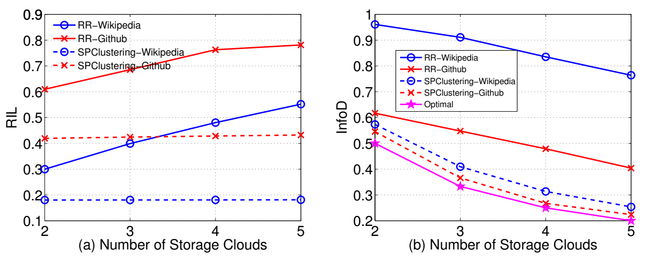

Fig. 8. Effect of CSP numbers on (a) RIL and (b) InfoD

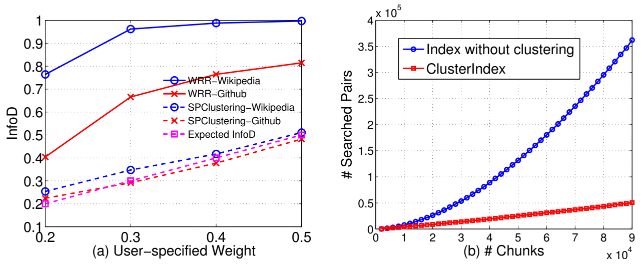

Fig. 9. (a) Effect of user's weight and (b) Pruning efficiency by ClusterIndex

- Is there any effect on the number of CSPs on which the users distribute their data?
- How does the weight specified to different CSPs by users influence the information loss?

Number of modifications. In this set of experiments, we have five CSPs with equal weights. 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 7 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 7 (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 7 (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 7. 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 weight. In Figure 8(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 8(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 8(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.

User-specific weight. In practice, users may have different weights for different CSPs. In StoreSim, the weight of each CSP not only coordinates the storage load but also the InfoD for preventing the information leaking. In this part, we do the experiments with 5 CSPs with normalized weights. We increase the weight of one CSP from 0.2 to 0.5 while the remaining is evenly distributed to the other four CSPs. Both datasets have 8 modifications and are synchronized to five CSPs for 9 times. Figure 9(a) shows the results for the CSP with the varying weight under SPClustering and *weighted round robin* (WRR). The bottom most line represents the expected InfoD in terms of user specified weights. Under SPClustering, it can be clearly observed that users can control the InfoD of a CSP by assigning different weights to it. However, the results also shows that information leakage of WRR for Wikipedia dataset can be as high as nearly 100% when it is assigned to obtain only 50% of the entire data. In other words, under WRR, the CSP ends up obtaining nearly all the information even though the user expected it to obtain only 50%.

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 9(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).

7.5 System Attackability Analysis

Attack model. We assume that the attackers maintain a network of resources to establish multiple *attack channels* to launch attacks to the target system. The attack channel can be either the communication links in computer networks (e.g., cyber attack) or the Human connections in social engineering (e.g., insiders attack). We define t he average cost of an attack channel as c which can be evaluated in terms of time or money. In the multicloud scenario, instead of a single point of attack to get everything, the attackers have to figure o ut m any t hings e .g., t he n umber o f CSPs that stores the data and the impact on the number of attack channels along with the increase in the number of CSPs, which makes the attack much more complex. To model this complexity, we assume that the attack channels has a polynomial growth rate $\mathcal{O}(n^m)$ with the number of CSPs adopted in a multilcoud system, where n is the number of CSPs and m is denoted as *multicloud effect* factor. Thus, given the average cost per attack channel c, we can define the cost function of an attack as $f(c, n, m) = c * n^m$, which captures that the cost of an attack increases with the number of attack channels. The c is mathematically a scale factor which can incorporate other extra costs due to the diversity factors in different systems. For example, the attacker may take more time and money on attacking an encrypted system.

System attackability. For different CSPs storing different proportion of information, $w_1, w_2, \ldots$ with probabilities of being compromised in an attack, $p_1, p_2, \ldots$, the expected benefit for attackers is $\sum_{i=1}^{n} p_i * w_i$. Based on this, We define the attackability of a system as a benefit-cost ratio to the attacker.

$$attackability = \frac{\sum_{i=1}^{n} p_i * w_i}{c * n^m}, 0 < p_i, w_i \leq 1 \qquad (9)$$

In other words, the attackability of a system is a measure of utility of potential attacks in terms of benefits gained and costs incurred. We can see that the high attackability of a system means that the attacker can gain high benefits at low costs. The probability p_i can be also interpreted as the risk of information leakage for each SDC. We note that p_i can equal 1 which can happen under the insider attack or coercion from the government. We also note that the attackability of a system is inversely proportional to n^m, which means that the more CSPs a multicloud system has and the higher multicloud effect is, the more arduous a system can be compromised.

Multicloud system settings. In this section, we further discuss four different settings of the mutlicloud system: *1) SingleCloud*, a single centralized cloud storage system with $n = 1, w_1 = 1$: in this case, multicloud effector factor m has no impact on the number of attack channels since it is not a multicloud system. The attackability of SingleCloud is $\frac{p_1}{c}$, which only depends on the average cost and the probability of CSP being compromised. *2) OptMulCloud*, an optimal multicloud storage system with $n > 1, \sum_{i=1}^{n} w_i = 1$: under this case, we can see that all the information are ideally distributed among the SDCs, corresponding to the weight of each SDC. *3) StoreSim*, our proposed system approximates to the optimal status by putting all the similar data nodes in the same cloud. *4) StoreRR*, the system with RR data distribution: the system without clustering similar data nodes in the same cloud may leak more information during the distribution. Both StoreSim and StoreRR have the setting that $n > 1, \sum_{i=1}^{n} w_i > 1$.

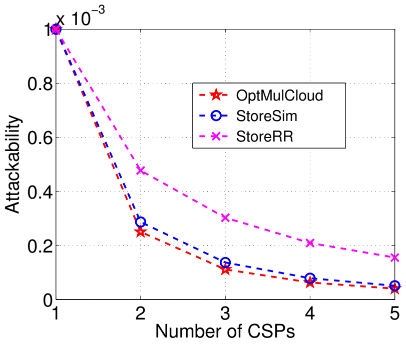

Fig. 10. Attackability of different systems with c=1, m=2, p=0.001

Proof of effectiveness. For the purpose of comparison, we keep three factors c as 1, m as 2 and p as 0.001 unchanged while varying the number of CSPs n from 1 to 5. We assume that all the CSPs are equally reliable with the same probability $p = 0.001$ of being compromised (i.e., the reliability of all the CSPs is 0.999) and all the CSPs are assigned the same weight to get the same amount of information. The values of w_i for StoreSim and StormRR are supplied by values of information density which are derived from our evaluation. Figure 10 shows the attackability of four proposed systems. It is clear that our proposed StoreSim, whose performance is near to that of OptMulCloud, is two times more complex than StoreRR to be attacked as the number of CSPs is 5. As for *SingleCloud* system ($n = 1$ in the Figure 10), all the information are stored in only one centralized cloud, which leads to a high value of attackability. Furthermore, it is worth mentioning that the attackability of StoreRR with 5 CSPs is higher than that of StoreSim with only 3 CSPs. This is an important result which shows that unplanned data distribution is worse than our scheme even under the presence of comparatively more CSPs.

From our attackability analysis, we can safely conclude that under our proposed StoreSim system, optimized data distribution across the multicloud reduces the risk that *wholesale* information is leaked and makes the attacks on *retail* information much more complex.

7.6 Discussion

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

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. 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 11 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 1 GB), the overhead of *LMLayer* is about 20% 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.

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 increase as shown in Figure 11.

Syntactic vs Semantic. In our paper, the information leakage function is designed based on syntactic similarity metric rather than semantic measures such as semantic similarity, semantic relatedness and semantic distance [39]. 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 measures is an orthogonal task to ours, since efficiently analyzing semantic similarity in users' data involves data curation and sophisticated machine learning techniques, which are not always time efficient for large datasets. Our future 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 modules.

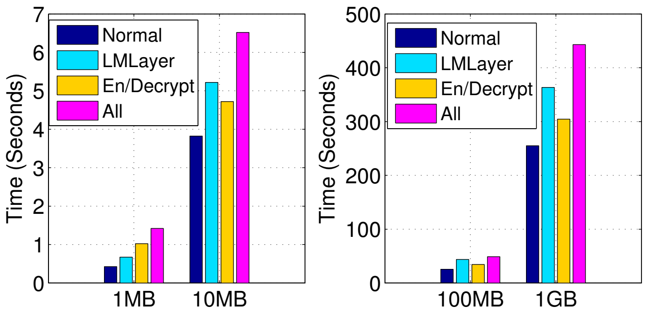

Fig. 11. Time cost with different configurations by varying file size

Encryption vs StoreSim. Encryption is the most strongest and effective way to prevent information leakage. However, if all the data are encrypted, users will not be able to enjoy many other services provided by storage service providers such as file sharing or collaboration. Most of these services cannot operate over the ciphertext. Though there are some works on homomorphic encryption (i.e., allows computations to be carried out on ciphertext) [40], resorting to encryption as the ultimate solution to prevent information leakage requires rewriting most of the cloud storage applications, which is evidently non-realistic. StoreSim provides an alternative approach to reducing information leakage to each CSP by optimizing data chunks distribution algorithms. 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. As shown in the Section 7.5, there is no single point of attack which can leak the wholesale information.

8 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. We show that distributing data chunks in a round robin way can leak user's data as high as 80% of the total information with the increase in the number of data synchronization. To optimize the information leakage, we presented the StoreSim, an information leakage aware storage system in the multicloud. 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 multicloud. We show that our StoreSim can achieve near-optimal performance and reduce information leakage up to 60% compared to unplanned placement. Finally, through our attackability 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] 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.

[2] 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.

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

[4] 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*. IEEE Computer Society Press, 2012, p. 20.

[5] 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.

[6] 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.

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

[8] 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*. ACM, 2013, pp. 205–212.

[9] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: understanding personal cloud storage services," in *Proceedings of the 2012 ACM conference on Internet measurement conference*. ACM, 2012, pp. 481–494.

[10] U. Manber *et al.*, "Finding similar files in a large file system." in *Usenix Winter*, vol. 94, 1994, pp. 1–10.

[11] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," *ACM Transactions on Computer Systems (TOCS)*, vol. 29, no. 4, p. 12, 2011.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] H. Zhuang, R. Rahman, P. Hui, and K. Aberer, "Storesim: Optimizing information leakage in multicloud storage services," in *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*. IEEE, 2015, pp. 379–386.

[17] 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.

[18] T. Zou, R. Le Bras, M. V. Salles, A. Demers, and J. Gehrke, "Cloudia: a deployment advisor for public clouds," in *Proceedings of the VLDB Endowment*, vol. 6, no. 2. VLDB Endowment, 2012, pp. 121–132.

[19] 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.

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

[21] 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 Seventh Symposium on Usable Privacy and Security*. ACM, 2011, p. 13.

[22] 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.

[23] 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.

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

[25] 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*. ACM, 2007, pp. 141–150.

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

[27] "Emc hybrid cloud computing," http://www.emc.com/cloud/hybrid-cloud-computing/index.htm.

[28] "Ibm multicloud toolkit," http://www.zurich.ibm.com/csc/security/toolkit/.

[29] "Microsoft hybrid clouds," http://www.microsoft.com/en-us/server-cloud/solutions/hybrid-cloud.aspx.

[30] J. W. Hunt and M. MacIlroy, *An algorithm for differential file comparison*. Bell Laboratories, 1976.

[31] 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.

[32] A. Rajaraman and J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2011.

[33] 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.

[34] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

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

[36] C.-Y. Chan and Y. E. Ioannidis, "Bitmap index design and evaluation," in *ACM SIGMOD Record*, vol. 27, no. 2. ACM, 1998, pp. 355–366.

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

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

[39] S. Harispe, S. Ranwez, S. Janaqi, and J. Montmain, "Semantic similarity from natural language and ontology analysis," *Synthesis Lectures on Human Language Technologies*, vol. 8, no. 1, pp. 1–254, 2015.

[40] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

Hao Zhuang received the BSc degree in software engineering from Northeastern University, China in 2009, and the MS degree from Peking University, China in 2012. He was awarded the Erasmus Mundus full scholoarship from the Europen Union for exchange study in security and mobile computing at Aalto University School of Science and Technology (TKK), Finland and Royal Institute of Technology (KTH), Sweden, respectively. He is currently working toward the PhD degree with a topic of decentralized cloud computing in Distributed Information Systems Laboratory (LSIR) at EPFL Lausanne, Switzerland. His major research interests include efficient resource allocation in decentralized cloud computing, optimization in the multicloud environment and data center analytics. He is a member of the IEEE.

Rameez Rahman received the PhD degree in "parallel and distributed systems" from Delft University of Technology, the Netherlands in 2011. He is currently serving as a research scientist at Bell Lab, Belgium. He has invested his research efforts in utilizing interdisciplinary ideas for building new, socially intelligent ICT systems. Such works have not only led to the development of more robust, efficient and fair protocols (results which have been published in many prestigious venues), but can also open up vistas for social scientists to experimentally and empirically test the feasibility of various social principles, using ICT systems. He envisions that much of the research impetus in the near future, in the sciences in general, and ICT in particular, is going to come from developing countries such as Pakistan. Toward that end, he intends to play his small part in fostering a research culture in Pakistan that can facilitate the realization of this vision.

Pan Hui received his Ph.D degree from Computer Laboratory, University of Cambridge, and earned his MPhil and BEng both from the Department of Electrical and Electronic Engineering, University of Hong Kong. He is currently a faculty member of the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology where he directs the HKUST-DT System and Media Lab. He also serves as a Distinguished Scientist of Telekom Innovation Laboratories (T-labs) Germany and an adjunct Professor of social computing and networking at Aalto University Finland. Before returning to Hong Kong, he has spent several years in T-labs and Intel Research Cambridge. He has published more than 150 research papers and has some granted and pending European patents. He has founded and chaired several IEEE/ACM conferences/workshops, and has been serving on the organising and technical program committee of numerous international conferences and workshops including ACM SIGCOMM, IEEE Infocom, ICNP, SECON, MASS, Globecom, WCNC, ITC, ICWSM and WWW. He is an associate editor for IEEE Transactions on Mobile Computing and IEEE Transactions on Cloud Computing, and an ACM Distinguished Scientist.

Karl Aberer received the Ph.D. degree in mathematics from the Eidgenossische Technische Hochschule Zurich (ETH), Zurich, Switzerland, in 1991. From 1991 to 1992, he was a Postdoctoral Fellow with the International Computer Science Institute (ICSI), University of California, Berkeley, CA, USA. In 1992, he joined the Integrated Publication and Information Systems Institute (IPSI), Forschungszentrum Informationstechnik GmbH (GMD), Darmstadt, Germany, where he was leading the research division Open Adaptive Information Management Systems. Since 2000, he was a Full Professor for Distributed Information Systems with the École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland. From 2004 to 2011, he was consulting for the Swiss government on research and science policy as a member of the Swiss Research and Technology Council (SWTR). From 2005 to 2012,he was the Director of the Swiss National Research Center for Mobile Information and Communication Systems (NCCR-MICS, www.mics.ch), Lausanne. Since September 2012, he has been the Vice-President of EPFL responsible for information systems. His research interests include decentralization and self-organization in information systems with applications in peer-to-peer search, semantic web, trust management, and mobile and sensor networks. Dr. Aberer is a Member of the editorial boards of the International Journal on Very Large Data Bases (VLDB), the ACM Transaction on Autonomous and Adaptive Systems, and the World Wide Web Journal.