Securing Files in the Cloud

Kheng Kok Mar
School of Information Technology, Nanyang Polytechnic, Singapore

Abstract

Public cloud storage has grown in popularity in recent years, both for personal and enterprise use. However, as with any 3rd party hosting solution, data security remains the major concern for most people. Secured Virtual Diffused File System (SVDFS) is a distributed file system which aims to address the security concerns by allowing users to transparently layer a secured virtual file system on top of existing public cloud infrastructure. SVDFS uses Information Dispersal Algorithm (IDA) to split the data into multiple unrecognizable slices and distribute them across multiple storage nodes in one or more cloud providers. With IDA, the adversary is required to compromise a threshold number of slices, in order to reconstruct the original information. IDA also provides availability of data without requiring full data replication. It helps to mitigate the loss of information in case a storage server or a specific storage medium is corrupted or compromised. Unlike other IDA-based storage systems, SDVFS supports file system operations such as streaming I/O and random read and write of files which have been dispersed using IDA.

1. Introduction

Public cloud storage has seen tremendous growth in the past years, both for personal and enterprise use. The convenience of access from anywhere and anytime has driven the usage of public cloud for personal storage. This is evidenced by the popularity of services such as Dropbox and SugarSync. In the enterprise space, especially the small and medium enterprises, the main driving force is cost. With the uncertainty of world economy, organizations are increasingly looking towards cloud as the means to reduce their total IT cost.

With off-premise storage, the risk of information theft is more pronounced as compared with on-premise storage. Organizations are concerned whether their company data are secured against theft from attackers both outside and inside the "cloud". It is highly possible that the data is stolen by the very people who are managing the cloud, regardless of stated security controls and compliance of cloud providers.

Furthermore, the integrity and availability of data in the cloud is a big concern. There is no guarantee that cloud service provider (CSP) will be in business forever and the risk of losing the data is very real, as witnessed in the high-profile case of Linkup [1], an online storage company which folded after losing substantial amount of customers' data. More recently, Amazon Web Service is not available for more than 24 hours in April last year [2], and the following month, Microsoft suffered a major disruption to its Business Productivity Online Services [3]. The sense of not having control over one's data is a major obstacle for mass adoption of cloud computing by enterprises. Even if customer has the opportunity to move the data to another cloud provider, the cost of migration would be prohibitive.

To address data confidentiality, most cloud storage security solutions depend on encrypting the data before storing them in the cloud. However, encryption entails complicated key management and there is broad implication if encryption keys are compromised. Furthermore, the security of most encryption algorithms depends on computational complexity of certain mathematical operations, such as factoring of a product of two large primes. However, with the rapid increase in computational power of modern computers, and the availability of massive computing power in the cloud, what is considered strong encryption currently may be broken in the near future.

To address data availability, we usually resort to data replication. However, each replicated data represents extra point of attack. Integrity of data is also difficult to guarantee in the cloud. CSP may move data to a lower-tiered storage than agreed-upon for economic reason or to hide data loss incidents [4] [5]. The traditional integrity checking of files are not applicable for outsourced data without having a local copy of the data.

This paper proposes a distributed file system for the cloud which is based on Information Dispersal [6] to better address the concerns of data confidentiality, integrity and availability. There are many other implementations of IDA-based storage systems. However, most of these systems are more suited for static file storage and retrieval. They are not designed to support file operations typically expected of file system such as streaming I/O and random read and write of files which have been dispersed by IDA.

2. Threat model and design considerations

2.1. Threat model

As the user's data resides off-premise, within the CSP network, the security threats can originate from two sources: internal and external attacks. In the case of internal attack, we assume CSP to be possibly self-serving, untrusted and malicious. It may move data to a lower-tier storage than what is agreed upon in the contract, or attempt to hide a data loss incident. It may also maliciously try to modify or delete the data or steal confidential information from the stored data for financial gain. In the case of external attacks, the attacker comes from outside the CSP domain and would try to penetrate the CSP network to gain access to some storage servers in order to modify or delete the data, or simply to steal confidential information.

In our model, we assume all the storage servers in the cloud are sufficiently protected, for example, with host-level firewall, so that adversary can compromise some but not all servers at different point in time. We also consider the adversary to be someone who is capable of monitoring and observing network traffic in and out of the cloud storage servers. It is further assumed that servers and the clients within the boundary of the enterprises are well-protected with existing security best practices, and the critical information is encrypted with the strongest possible encryption. In our following discussion, it is assumed that all communication between different components is secured using two-way SSL.

2.2. Design goals

Besides addressing the security concerns mentioned above, SVDFS is designed with the following goals in mind:

- The virtual file system can be transparently overlaid on any existing Infrastructure as a Service (IaaS) cloud, without the involvement of CSP or requiring the CSP to upgrade the existing infrastructure to support it.
- It is able to support multi-cloud configuration for added availability and security.
- It is able to support typical file system operations such as random read/write besides sequential read/write. It should efficiently support frequent updates of files without incurring too much overhead in terms of network traffic and computing resources.
- It supports single-writer, multiple-reader mode of file operation with efficient support for large number of concurrent reads.
- It allows clear separation of roles of data owner (the customer) and storage owner (the CSP)

3. Secured Virtual Diffused File System

3.1. System architecture

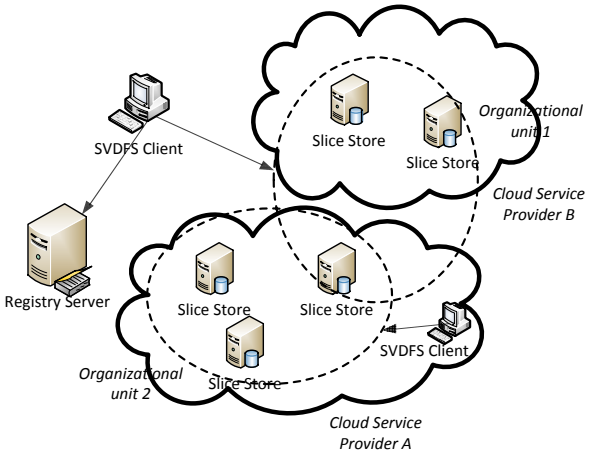

Figure 1. SVDFS System Architecture

Figure 1 shows the system architecture of Secured Virtual Diffused File System (SVDFS) and its deployment in a public cloud environment. Essentially SVDFS comprises of a registry server, one or more slice stores and SVDFS clients. Slice stores are data storage nodes and are used to store the file slices generated by IDA as described in Section IV. The slice stores reside with one or more CSPs and may be a mixture of different storage options, e.g. standalone VMs with attached storage or REST-based single-purpose storages. As depicted in the diagram, a slice store can logically belong to one or more administrative domain. The registry server hosts the master table which is a repository of SVDFS file metadata. It is similar to an inode table of UNIX or file allocation table (FAT) of Windows. Registry server is located outside the cloud, within the customer's premises, and is under the full control of the customer (the data owner). SVDFS client provides file system interface to the application, similar to a Network File

System (NFS) client. The SVDFS client can be used by application residing outside the public clouds (e.g. within the enterprise), or application residing within the cloud (e.g. a web application which runs on a Platform as a Service cloud or a compute instance of an IaaS cloud). We will describe each of these components in more detail in the following sections.

3.2. Registry Server

Registry Server consists of two server processes: master table server and slice store manager. Master table server maintains the metadata as shown in Figure 2 (list shown is not exhaustive), while slice store manager manages slice store registration and removal, as well as keeping track of the health and availability of the registered slice stores.

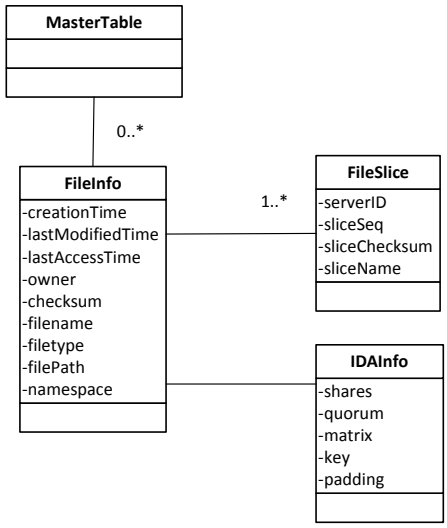

Figure 2. Metadata

Metadata contains information such as IDA matrix used to split the file, location of slices, namespaces of slice stores, secret keys, as well as usual file attributes such as file size, file creation time, owner and so on. One of the primary purposes of master table is to maintain the mapping of file to file slices, using *FileInfo* and *FileSlice* data structures as shown in Figure 2. *FileInfo* represents a single file object and is associated with one or more *FileSlice* data. The *FileSlice* corresponds to a slice of the file and contains information about the server which holds the slice, the sequence number of the slice (as determined during the splitting process of IDA), slice name and slice checksum which is used to check the integrity of each slice. Each file is associated with an *IDAInfo* data structure which contains information such as the IDA matrix used to split the file, the configured shares and quorum. The matrix used can be either system-wide or on a per-file basis. The *FileInfo* is segregated by namespace and is sorted by its unique *filePath* within each namespace. Metadata is stored in-memory to support fast access by large number of concurrent users. Since metadata required for each file is quite compact (in most cases, less than 1 KB), a typical server with 2 GB of memory can comfortably support millions of file entries.

The master table server, through its REST interface, handles non-data aspects of file operation such as file entry creation, deletion, renaming, slice store selection and allocation. SVDFS client connects directly to slice store servers for actual data transfer of file slices. The separation of data path from metadata operations ensures that the registry server remains responsive under heavy load.

Since the metadata is stored in-memory, we need to ensure metadata's integrity in case of a server crash. We employ a combination of transaction log and snapshot image files to achieve our purpose.

Transaction Log and Snapshot Image

Transaction log records every file operation which updates the metadata (e.g. create, update or delete file). Snapshot image is a persistent copy of metadata at a particular point in time. In the event of server crash, the transaction log is replayed to reconstruct the in-memory metadata when the server restarts. The transaction log can grow to a very large size over time, and it will take a long time to replay the log. As such, a *max_log_size* is configured such that when transaction log grows to the specified size, master table server will switch over to a new transaction log file. A separate thread is spawned to create a new snapshot image based on last snapshot image, if there is one, and all the transactions from the old transaction log are applied to the new snapshot image. This process is similar to what Google File System [7] or Hadoop File System [8] has implemented.

3.3. Slice Store

Slice store is the storage node for file slices. There are two types of slice store: *active slice store* and *passive slice store*.

Active Slice Store

For active slice store, a REST-based slice store server process runs on a server (either physical or virtual) with direct attached storage, such as an Amazon EC2 instance with attached Elastic Block Store (EBS). Each slice store is configured with unique slice store id and the namespaces it belongs to. The

slice store id uniquely identifies the slice store. Since the location of any file slice is tied to it, the slice store id is immutable. However, its IP address can be dynamic and the slice store manager maintains a mapping table to map slice store id to its actual public IP address. This design is well-suited for a cloud environment such as Amazon EC2 where an EC2 instance is assigned dynamic IP address when it restarts.

When the server process starts up, it registers with slice store manager its slice store id, namespaces, management endpoint address and port, type of cloud interface supported and its loading factor. The loading factor reflects the storage utilization of a slice store. Registry Server will take into account the loading factor when it allocates slice stores to client in order to better balance the load across all the slice stores. The slice store also periodically sends "hello" message to slice store manager to report its status and loading factor. If slice store manager does not hear the "hello" message from a slice store, for a configured timeout period, it will mark the slice store as unavailable from its allocation list. All communications between registry server and slice stores are done through two-way SSL.

Passive Slice Store

Passive slice store refers to the special-purpose Blob services such as Amazon S3 or Windows Azure Blob. Typically these services allow user to create "bucket" or "container", which acts like a folder. Each bucket or container is accessible through a REST interface using a globally unique URL. The bucket or container will act as slice store. Slice store manager will be configured manually with the information of the bucket or container such as service URL, credentials required to access the bucket or container. The slice store manager acts as a proxy to the bucket or container. It sends "dummy" requests to the blob service URLs to check the health and availability of the blob containers or buckets.

File Slice naming

Name mangling is used for slice names to make it difficult for adversary to correlate file slices to a file. For example, we use GUID to generate the slice name. Other scheme supported is by filename hashing. To make each slice to have a unique slice name, we generate the slice name as a secured hash of concatenated string of namespace, full file path and slice sequence number.

Access Control for Passive Slice Store

Access to Amazon S3 or Windows Azure Blob services normally requires the use of an access key and a secret key associated with the cloud storage accounts. Since SVDFS client connects directly to the slice store for data transfer, the client would need to have the proper credentials. However, it is risky to share the access key/secret key pair with a large number of clients. To minimize the risk of exposing the cloud account credentials, we have employed Shared Access Signature (SAS) [9] for Windows Azure blob and Query String Authentication (QSA) [10] for Amazon S3. For example, in the case of Windows Azure blob, instead of sending account access key/secret key pair to the SVDFS client, the registry server generates a one-time SAS URL which incorporates information required for access to the blob such as the target resource, the validity period, the permission and so on. The SAS is tied to container policy so that in case we need to revoke a Shared Access Signature, we can just remove the associated container policy. Depending on the file size, the time validity granted may not be enough, and the client may have to request a new SAS from the registry if the SAS expires before it finishes reading or writing the slices.

3.4. SVDFS client

Figure 3 shows the SVDFS client interface and its major functional blocks. The SVDFS client provides file system interface to application. It is implemented as a Java library as well as Windows File System Driver.

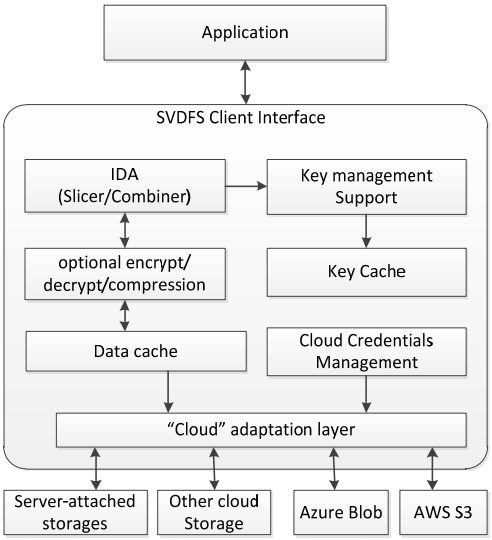

Figure 3. SVDFS Client

SVDFS Java library provides file I/O classes similar to standard I/O classes provided in JDK. This eases the

porting of Java application to use SVDFS. SVDFS client library is suitable for applications deployed in a PaaS environment, as it is not possible to install a kernel based file system in a PaaS environment. Application hosted on PaaS runs in a sandbox environment and typically has no access to the local file system. SVDFS client library allows the hosted application to directly use SVDFS for file I/O, instead of being constrained by storage options provided by the PaaS. For example, a hosted Google App Engine application can make use of the library to transparently access files in SVDFS and not constrained to use Google Datastore.

When a file is open for writing, SVDFS client requests the Registry Server to allocate a set of slice stores to use. The slicer then calls the key management module to retrieve the key necessary to generate the IDA transform matrix. When data is being written to the file, the slicer uses the matrix to transform the data stream into multiple slices which are sent to various slice stores through the cloud adaptation layer. When a file is closed, the client will update the registry server with the file metadata such as location of slices, IDA matrix, file size, slice size and so on. Similarly, when a file is open for reading, the client retrieves the corresponding metadata from the Registry Server such as the locations of the slices and the IDA transform matrix. It then retrieves the slices directly from slice stores using the cloud adaptation layer. The combiner uses IDA transform matrix to recombine the slices into the original data.

The cloud adaptation layer is responsible for using the correct interface as well as the appropriate credentials to interact with different types of slice store. It supports a pluggable architecture and new plugin can be developed to support new cloud storage option. Presently, the adaptation layer supports Windows Azure Blob Service, Amazon S3 and active slice store.

4. Information Dispersal

The security of SVDFS system is based on "paper shredder" approach, where a confidential document is shredded into multiple unreadable strips. We employ Information Dispersal algorithm (IDA) to split a file into multiple unrecognizable file slices. The compromise of one or more slices is inconsequential as they would not yield any useful information about the original file. To ensure added security, we may also want to scatter the paper strips into multiple secret locations. Similarly, in SVDFS, we do so by "diffusing" the slices into multiple storage nodes within the cloud to make it difficult for attackers to locate all the slices required.

IDA has been employed in various implementation of secure storage for files [11], [12], [14] [15]. In our scheme, we have used IDA to implement a secured virtual file system which supports typical file system operations such as streaming and random file I/O.

4.1. Overview of information dispersal

In IDA, we choose two integers n and m, where $n > m$. A file is split into n slices and a minimum of m slices are required for reconstruction of the original file. A transform matrix of n rows and m columns is used to transform the original file into n slices.

Let A be the $n \times m$ Cauchy matrix:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \cdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}$$

Each subset of m rows is linearly independent vector.

Let F be the original file of size N and b_i be the byte array of F. The bytes in F can be chunked into blocks of m bytes:

$$F = (b_1, b_3, \ldots b_m), (b_{m+1}, b_{m+2}, \ldots . b_{2m}) .. (b_{N-m+1,} \ldots b_N)$$

Let B be the input matrix of F and C the output matrix. Thus, we can write the following matrix equation:

$$A \cdot \begin{bmatrix} b_1 & b_{m+1} & \cdots & b_{N-m+1} \\ b_2 & b_{m+2} & \cdots & b_{N-m+2} \\ \vdots & \vdots & \ddots & \vdots \\ b_m & b_{2m} & \cdots & b_N \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1N/m} \\ c_{21} & c_{22} & \cdots & c_{2N/m} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nN/m} \end{bmatrix}$$

where

$c_{ik} = a_{i1} b_{(k-1)m+1} + a_{i2} b_{(k-1)m+2} + \ldots + a_{im} b_{km}$,

$1 \le i \le n, 1 \le k \le N/m$

Each row of C corresponds to a file slice, and n slices are produced.

Any m out of n slices can be used for combination. For simplicity of discussion, assume we use slices 1 to m for recombination.

Let A' be a subset of A of rows 1 to m. Let A^{-1} be the inverse matrix of A':

$$A^{-1} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1m} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{m1} & \alpha_{n2} & \cdots & \alpha_{mm} \end{bmatrix}$$

For reconstruction, the inverse matrix A^{-1} is applied to the m (out of n) slices to obtain the original data:

$$A^{-1}.\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1N/m} \\ c_{21} & c_{22} & \cdots & c_{2N/m} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{n2} & \cdots. & c_{mN/m} \end{bmatrix} = \begin{bmatrix} b_1 & b_{m+1} & \cdots & b_{N-m+1} \\ b_2 & b_{m+2} & \cdots & b_{N-m+2} \\ \vdots & \vdots & \ddots & \vdots \\ b_m & b_{2m} & \cdots & b_N \end{bmatrix}$$

where

$b_j = \alpha_{i1}c_{ik} + a_{i2}c_{2k} + ... + a_{im}c_{mk}$, $1 \le j \le N$,

$i = j \bmod m, k = \lceil j/m \rceil$

4.2. Streaming read/write operations

In streaming mode, data written to a file is treated as byte output stream which will be transformed and streamed to the slice store immediately. Similarly, when file slices are being read from the slice stores, they are transformed and delivered to the application immediately as byte input stream without having to wait for the complete slice to be available. This has the advantage of reducing the memory footprint required to support large file at the SVDFS client since the large file is not buffered locally. It is also more suited for applications which require streaming capability such as audio and video streaming.

We note from the equation above that each row of matrix *C* corresponds to a file slice, where c_{11} corresponds to 1st byte of the file slice 1, c_{21} corresponds to the 1st byte of file slice 2 and so on. We further note that c_{11} can be computed immediately when first *m* bytes are available in the input matrix. When *m* bytes of data are available from the file input stream, e.g. $b_1,\ldots,b_m$, the vector of *m* bytes is multiplied with transform matrix *A* to get 1st byte of data for the *n* slices, $c_{11}, c_{21},\ldots, c_{n1}$. When the next *m* bytes of data are available, e.g. $b_{m+1},\ldots,b_{2m}$, the vector of the new *m* bytes is multiplied with *A* to get the 2nd byte of data for the *n* slices, c_{12}, c_{22} .. c_{n2}. We can choose two system parameters *j* and *k* where *j* is multiple of *m* and $k \ge 1$. Whenever *j* bytes of data are written to a file, the transform can be immediately carried out to generate *n* slices, and whenever *k* bytes of each slice is obtained, the slices can be flushed to the slice stores. For efficiency and to reduce network traffic, *k* should not be too small. For file size which is not multiple of *m*, padding is required.

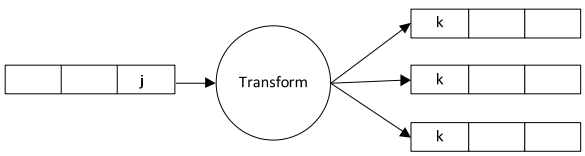

Figure 4. Transform process

Random File Read

Let *i* be the starting index of the file to be read. We calculate the byte "column" or offset into the file slices that is to be retrieved as:

$$s = \left\lfloor \frac{i}{m} \right\rfloor$$

Bytes starting from offset *s* are read from *m* slices, i.e. $c_{i(s+1)}, c_{i(s+2)}, c_{i(s+3)}, \ldots (i \in \{1,..,m\})$. Here first *m* slices are used for ease of explanation. In actual operation, any *m* of *n* slices can be used. Inverse transform matrix A^{-1} is then multiplied with these byte arrays to obtain the original data $b_{sm+1}, b_{sm+2}, b_{sm+3}$ and so on.

$$A^{-1} \cdot \begin{bmatrix} c_{1(s+1)} & c_{1(s+2)} & \cdots \\ c_{2(s+1)} & c_{2(s+2)} & \cdots \\ \vdots & \vdots & \ddots \\ c_{m(s+1)} & c_{m(s+2)} & \cdots \end{bmatrix} = \begin{bmatrix} b_{sm+1} & b_{(s+1)m+1} & \cdots \\ b_{sm+2} & b_{(s+1)m+2} & \cdots \\ \vdots & \vdots & \ddots \\ b_{sm+m} & b_{(s+1)m+m} & \cdots \end{bmatrix}$$

Figure 5 shows the interactions between the client, registry server and slice stores for *Random File Read* operation.

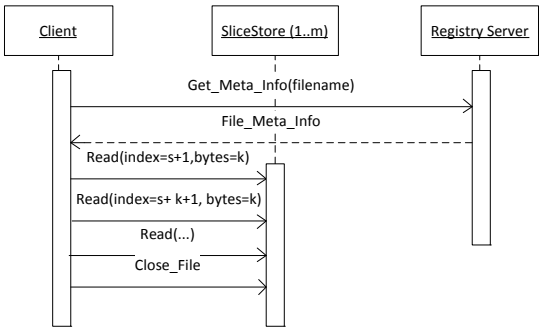

Figure 5. Random file read

Random File Write

For random write of a file at offset *i*, the affected offset of each slice can be calculated as follows:

$$s = \left\lfloor \frac{i}{m} \right\rfloor$$

If *c* is the number of bytes updated, and assume $c < N$ (the file size). Then we calculate

$$e = \left\lfloor \frac{i + (c-1)}{m} \right\rfloor$$

Bytes $c_{a(b+1)}$ $(a \in \{1,..,m\}, b \in \{1,..,e\})$ are read from m slices and multiplied with the inverse transform matrix A^{-1} to obtain $b_{sm+1}, b_{sm+2}, \ldots b_{sm+m}, \ldots b_{em+1}, b_{em+2}, \ldots b_{em+m}$ The elements starting from *x* where $b_{sm+x} = b_i$ are then updated with the new data. The updated data is then multiplied with matrix A to obtain the update slices.

$$A \cdot \begin{bmatrix} b_{sm+1} & b_{(s+1)m+1} & \cdots & b_{em+1} \\ b_{sm+2} & b_{(s+1)m+2} & \cdots & b_{em+2} \\ \vdots & \vdots & \ddots & \vdots \\ b_{sm+m} & b_{(s+1)m+m} & \cdots & b_{em+m} \end{bmatrix} =$$

$$\begin{bmatrix} c_{1(s+1)} & c_{1(s+2)} & \cdots & c_{1(e+1)} \\ c_{2(s+1)} & c_{2(s+2)} & \cdots & c_{2(e+1)} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n(s+1)} & c_{n(s+2)} & \cdots & c_{n(e+1)} \end{bmatrix}$$

Figure 6 shows the interactions between client, registry server and slice stores for Random File Write.

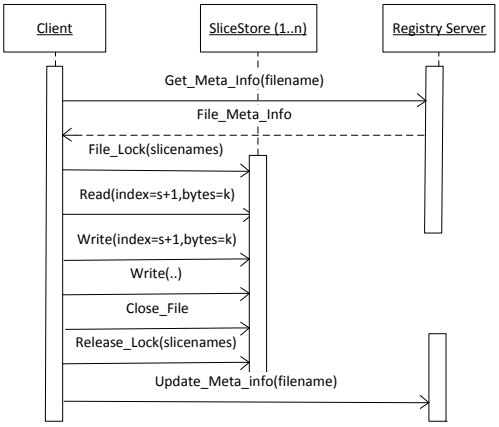

Figure 6. Random file write

5. Error detection & data integrity

To prevent the slices stored on the slice stores from being tempered with, a keyed hash is generated for each slice. We use the standard HMAC SHA256 for computing the keyed hash. The computed hash values h_i ($i \in \{1,\ldots,n\}$) are stored together with the slices on the slice stores and the secret key used to compute the HMAC, s_k, is stored with the registry server. During reconstruction of file, the client retrieves both the slices, C_i ($i \in \{1,\ldots,m\}$) and the corresponding keyed hashes h_i ($i \in \{1,\ldots,m\}$) from the slice stores. From the slice data, it computes the keyed hash, H_i ($i \in \{1,\ldots,m\}$) using s_k and check if $h_i = H_i$ ($i \in \{1,\ldots,m\}$). If $h_i \neq H_i$, then the particular slice C_i is discarded and another slice from the remaining $(n - m)$ slices will be chosen instead. The adversary cannot modify the data and generate a new keyed hash on the slice stores without the knowledge of secret key, s_k.

Streaming I/O support poses a challenge as we cannot compute a complete hash of the slice while the slice is being generated and streamed to the slice store. Furthermore, in a random write operation of a file, the slices on the slice store will be updated randomly as well, which requires a corresponding update in the stored key hash values. The keyed hash of the slice cannot be updated without having to first download the entire slice and re-compute the hash. It is obviously inefficient and goes against the intent of providing streaming random I/O support. To address this problem, each file slice is treated as a stream of fixed size blocks (e.g. by setting k to 512 or 1024 bytes) and the keyed hash is computed for each block as the slice is being generated by IDA encoding process. A concatenated list of *block hashes* is stored on the slice store together with the slice. Whenever a slice is updated, the keyed hash only needs to be re-computed for the affected blocks within the slice where the data is changed. For a block size of 1024 bytes, the storage overhead for the hashes is only about 3%.

However, it should be noted that adversary can maliciously corrupt the file slice by re-ordering or deleting one or more blocks and the corresponding *block hash* without being detected. The attack can be prevented by including a block sequence number as part of the hash computation.

To support "proof of retrievability" as described in [4], the registry server can randomly select the slice block to verify and download the corresponding slice block and its corresponding block hash. It then computes the keyed hash over the block, using the secret key and check if the computed key hash matches the block hash that is downloaded. Although this incurs more bandwidth than those described in [13] and [16], it does not require the generation of pre-computed verification token which is not suitable for streaming random I/O of the file.

6. Error recovery

Error recovery becomes more complicated in supporting streaming write operation. Since the file is not buffered locally, retransmission is not possible in case of failure of some slice stores during a write operation. To maintain the minimum availability requirement for the file, a new slice store is immediately allocated by the registry server for the client to continue writing the remaining data of the slice to. The registry server will maintain a link from the original slice to the slice fragments in the new slice store, as show in Figure 7. When the original slice store is restored, the registry server will perform background recovery process to consolidate the fragments of slices to a single slice on the original slice store.

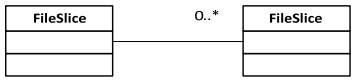

Figure 7. Linked segments of file slice

7. Security & performance analysis

7.1. Security

IDA-based scheme provides better cost-performance than a replication-based scheme in terms of data availability. For comparison, let us assume that a storage node has a failure rate of λ, and let R be the recovery rate, which is the possibility of recovering data when one or more storage nodes fail, and O be the storage overhead, which is the extra storage space required to maintain a certain recovery rate. For a replication based scheme, the storage overhead O is $100(n-1)\%$ and the recovery rate is $R = 1 - \lambda^n$, for a replication factor of n. For an IDA-based scheme, the storage overhead is $(n-m)/n$ and the recovery rate $R = 1 - \lambda^{n-m+1}$. We can see that the recovery rate increases when $(n-m)$ increases. For example, if $n = 10$, $m = 7$, and assume $\lambda = 0.1$, the IDA scheme will provide 99.99% recovery rate with an overhead of only 30%. A replication-based scheme will require 300% overhead to achieve similar level of recovery rate.

Traditionally, encryption is employed to ensure data confidentiality. However, SVDFS enhances data confidentiality with the use of IDA. To read a file, the adversary needs to compromise a minimum of m slice stores or eavesdrops on m slices. Furthermore, the adversary also has to determine which slices logically belong to a file. The adversary will also need to guess the transform matrix used and apply the matrix with the correct ordering of the slices. To achieve all these will be difficult in practical terms. To further enhance the security of SVDFS, the client can optionally encrypt the file slices before writing to the slice stores. The system can be configured to use a randomly generated matrix for every file, instead of using a global matrix for all the files. The client can also send "dummy" slices which contain random and useless data to the slice stores to confuse the adversary.

For added security, SVDFS allows slices to be spread across multiple cloud providers. As an example, we choose $n = 9$, $m = 6$, and 3 different CSPs, with each one holding 3 slices. The adversary will need to compromise at least 2 cloud providers to be able to reconstruct the data. Furthermore, even if one CSP fails, we can still access the remaining 6 slices from the other 2 CSPs. We also need not worry about erasing the data in the cloud if we discontinue the service with any one of the CSP.

It should be noted that security of the SVDFS relies heavily on the registry server. Furthermore, the registry server seems to be a single point of failure. Access to files would not be possible without the registry server. However, it should be pointed out that since registry server resides within company's premises, it can be adequately protected using well-proven techniques such as strong authentication, server hardening, firewall, IDS and so on. To ensure the continuous operation of the registry server, the registry should be configured with a HA cluster.

7.2. Performance

We have collected some performance data based on our SVDFS implementations. In our test setup, the registry server, the slice stores, NFS server and clients are running on virtual machines hosted on VMware ESX4i hosts. Each VM is configured with 1 GB of memory and 1 virtual CPU with no resource reservation. The NFS client and server used are based on CentOS 5.5 distribution. The time shown below are taken over multiple runs.

Benchmarking with NFS

We have measured the time taken for SVDFS read and write operations on files of different sizes and compared with that of NFS for benchmarking purpose. The SVDFS used is based on a configuration with n/m of 10/7.

Table 1. NFS vs SVDFS

	NFS		**SVDFS**	
	write	***read***	***write***	***read***
1 MB	317ms	2ms	485ms	259ms
50 MB	8435ms	38ms	7176ms	6945ms
200 MB	31763ms	145ms	27198ms	27171ms

From the performance data, it can be seen that the throughput of write operation of SVDFS is comparable with that of NFS write. However, NFS read performs significantly faster than SVDFS read. In this respect, it is worth noting that performance optimization techniques in the read operation, such as using a larger read block, or read-ahead caching have not been attempted. This will be future scope of work. We also did a profiling of the codes and found that majority of the time are spent on network I/O and buffer copy. The time used for IDA transform and calculating keyed hashes is a small percentage of total time spent.

Quorum and I/O Performance

We have also done some performance comparison for different n and m, for files of different sizes.

Table 2. Write Throughput

	***n*/*m*=20/17**	***n*/*m*=15/12**	***n*/*m*=10/7**	***n*/*m*=5/3**

1 MB	470	458	485	452
50 MB	9656	8792	7176	7836
200 MB	37993	33528	27198	29372

Table 3. Read Throughput

	***n*/*m*=20/17**	***n*/*m*=15/12**	***n*/*m*=10/7**	***n*/*m*=5/3**
1 MB	274	246	259	271
50 MB	9830	8071	6945	7873
200 MB	39830	33755	27171	29277

We observed that for small file, the performances are similar for different *n*/*m* configurations. However, for medium and large files, *n*/*m* of 10/7 seems to yield the best performance. Theoretically, we can achieve more parallelism in data fetching with higher *n*. However, in practice, the overhead of having too many concurrent threads offsets the advantage of parallel data transfer.

For *n*/*m* of 10/7, we can achieve a write throughput of about 7 MB/s for larger files, and about 2 MB/s for smaller files. The throughput is lower for smaller files due to a larger percentage of overhead (e.g. querying and updating registry server) compared with the file size.

Similarly, we achieve the best read throughput for medium and large files for *n*/*m* of 10/7. The read throughput is about 7MB/s for larger files, similar to the write throughput. However, for smaller files it is about 3.86Mb/s.

"Slow Slice Effect"

In the current implementation, it is sometimes observed that the read I/O performance is significantly degraded if one or more slice is slow to arrive. As the combiner cannot combine the slices until certain number of bytes from all the m slices of data is available in the buffer, a slow read thread will hold up the rest of the threads. One possible solution to this is to read more than the quorum of slices and whenever first m read threads have read the required number of bytes, the combination process starts and the slice data from remaining read threads discarded.

8. Related works

Bian and Seker [14] described a distributed file system which uses IDA to recursively slice and distributes the file segments onto different P2P nodes and uses a hashed-key chain algorithm to implement layered encryption to encrypt data stored in the nodes. The main design aim is to protect the privacy of user. It does not address the data integrity issue in an untrusted cloud provider network. It also does not address how the file system handles dynamic and random read write of data. The performance of file I/O can also be an issue as slices need to propagate up and down the parent-child hierarchy of nodes. Furthermore, the use of P2P, while providing a degree of fault tolerance against node failures, can pose a management challenge in an enterprise setting. Tahoe [15] is another file system based on IDA and is designed to provide secure, long term storage such as backup applications.

For storage integrity checking, Juels and Kaliski [4] first described a formal "proof of retrievability" model which combines spot-checking and error-correcting codes to ensure data possession and retrievability. Bowers *et al.* [13] applied POR to a distributed set of servers. Wang, *et al.* [16] and Ateniese *et al*. [17] use a set of pre-computed tokens to challenge the cloud storage servers to prove data possession. However, the pre-computation of verification token treats the file as relatively static object and hence is not suitable for use in SVDFS context where it needs to support streaming and random file I/O. Ateniese *et al.,* in an earlier work [18], described the use of a homomorphic verifiable tag based on RSA which allows unlimited challenges without having to pre-compute challenge tokens. However the computation overhead of such token can be quite expensive. Wang *et al.* [16] also describes a mechanism to update the parity blocks when a data block is updated. It does so by computing an update matrix *F* based on the delta of new and old values.

9. Conclusions

The Secured Virtual Diffused File System (SVDFS) provides data security, integrity and availability suited for deployment in a public cloud environment. The system provides a clear separation of storage owner and information owner. It is designed to be scalable and fault tolerant and is well-suited for multi-cloud architecture. Its support of streaming I/O and random read write enables support of wide range of applications.

Our performance test shows that SVDFS is able to achieve reasonably good write throughput. It performs relatively well for large files. We believe that the read throughput can be further improved with larger read block, read-ahead caching and other optimization techniques.

10. References

[1] J. Kincaid, "MediaMax/TheLinkup Closes Its Doors," [Online]. Available: http://www.techcrunch.com/2008/07/10/mediamaxthelinkup-closes-its-doors/, July 2008.

[2] "Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region", [Online]. Available: http://aws.amazon.com/message/65648/

[3] "Microsoft BPOS Outage Shows Danger of Trusting the Cloud," [Online]. Available: http://www.pcworld.com/businesscenter/article/227951/

[4] A. Juels and J. Burton S. Kaliski, "Pors: proofs of retrievability for large files", in *Proc. of CCS'07*, Alexandra, VA, October 2007, pp. 584-597.

[5] M.A. Shah, M. Baker, J.C. Mogul, and R. Swaminathan, "Auditing to Keep Online Storage Services Honest," in *Proc. of HotOS'07*, Berkeley, CA, USA, 2007, pp. 1–6.

[6] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *J. ACM*, vol 36, no 2, pp. 335-348, 1989.

[7] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system," in *Proc. of the 19th ACM symposium on Operating systems principles*, New York, NY, USA, 2003, pp. 29-43.

[8] Hadoop. [Online]. Available: http://hadoop.apache.org/

[9] "Shared Access Signature," [Online]. Available: http://msdn.microsoft.com/en-us/library/windowsazure/hh508996.aspx

[10] "Signing and Authenticating REST Requests," [Online]. Available: http://docs.amazonwebservices.com/AmazonS3/latest/dev/RESTAuthentication.html

[11] A. Iyenger, R. Cahn, J. Garay, and C. Jutla, "Design and Implementation of a Secure Distributed Data Repository," in *Proc. of the 14th IFIP Internat. Information Security Conf* , 1998, pp. 123-135.

[12] C. Cachin and S. Tessaro, "Asynchronous verifiable information dispersal," in *Proc. of the 24 th IEEE Symposium on Reliable Distributed Systems*, 2005, pp. 191-202.

[13] K. D. Bowers, A. Juels, and A. Oprea, "Hail: A high-availability and integrity layer for cloud storage," in *Proc. of CCS'09*, 2009, pp. 187–198.

[14] J. Bian and R. Seker, "Jigdfs: A secure distributed file system," in *Proc. of 2009 IEEE Symposium on Computational Intelligence in Cyber Security*, 2009, pp. 76-82.

[15] Z. Wilcox-O'Hearn and B. Warner, "Tahoe: the least-authority filesystem," in *StorageSS '08: Proc. of the 4th ACM international workshop on storage security and survivability*, New York, NY, USA, 2008, pp. 21–26.

[16] C. Wang, Q. Wang, K. Ren, and W. Lou, "Ensuring data storage security in cloud computing," in *Proc. of IWQoS'09*, 2009, pp. 1-9.

[17] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," in *Proc. of SecureComm'08*, 2008, pp. 1-10.

[18] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proc. of CCS'07*, Alexandria, VA, October 2007, pp. 598–609.

11. Acknowledgements

This work is supported by the Singapore Ministry of Education, under the Innovation Fund Grant MOE2009-IF-1-009.