A Comparative Study of Existing Cloud Security System Models as against an Implementation of the CDDI Model Dubbed SecureMyFiles System

Frimpong Twum
Kwame Nkrumah University of Science and Technology
Department of Computer Science
Kumasi - Ghana

J. B. Hayfron-Acquah
Kwame Nkrumah University of Science and Technology
Department of Computer Science
Kumasi - Ghana

J. K. Panford
Kwame Nkrumah University of Science and Technology
Department of Computer Science
Kumasi - Ghana

ABSTRACT
Prior studies have established the security of resources outsourced for cloud storage heavily preys on subscribers' minds. Existing cloud security frameworks classified as direct architectures (such as provided by DropBox, Box, Google, BackBlaze B2) and indirect architectures provided by the Cloud Access Security Brokers (CASB)/Security as a Service (SECaaS) providers have been found to be inadequate in assuring the cloud subscriber of the security of resources in terms data corruption, privacy, and performance in recovering data. This study employed an experimental lab set-up using JAVA, SQL, and PHP to develop the Cloud Data Distribution Intermediary (CDDI) framework into software system dubbed Secure My Files (SMF). The SMF system provides users a choice of selecting one of four data priority levels (Low, Normal, Important, or Critical) at the time of uploading their data resource for cloud storage. The priority level selected determines: the uploading and downloading process the system uses, the amount of data that can be recovered in the event of data corruption, and the performance during data recovery. The security strength of the SMF system in relation to assuring of confidentiality, Integrity, and Availability of cloud data was found to be much stronger than the existing models and systems provided by DropBox, Box, Google Drive, BackBlaze B2, and CASB/SECaaS. This is because with the SMF System the cloud subscriber data is distributed across different Cloud Service Providers (CSP's) distributed storage infrastructures as against the existing frameworks and systems where the data reside with one single provider.

General Terms
Cloud Computing, Cloud Security Framework, Cloud Subscriber, Cloud Service Providers, Security, Privacy, Cloud Computing Framework

Keywords
Cloud Computing, Cloud Security Framework, Cloud Subscriber, Cloud Service Providers, Security, Privacy, Cloud Computing Framework

1. INTRODUCTION
Cloud computing come with numerous benefits but also faces several security issues especially in terms of data privacy, data integrity, and data availability. Cloud computing characteristics of resource pooling, multi-tenancy, on-demand self-service, broad-network access, and rapid elasticity introduces new security threats in terms of data accessibility, data ownership and data accuracy and hence demand new approaches for dealing with them. Traditional counter security measures have been found to be inadequate for dealing with cloud security issues, an example been that encrypting data before sending it to a single CSP does not protect the data from been decrypted, deleted, or altered (O'Reilly, 2017). A survey conducted by the Cloud Security Alliance CSA in 2016 identifies twelve security concerns of cloud computing including data breaches, data loss, malicious insiders and Denial of Service among others (The Treacherous 12, 2017). Other cloud security issues include: the cloud provider profiting from using the subscriber's data entrusted in their care for advertising, or using the data to learn more about the subscriber for their own interest.

Although research suggests that cloud security threats from multi-tenancy architecture have been reduced by major CSPs such as Amazon and Microsoft, the threats are still real especially for smaller CSP's (Shapland, 2017). In addition, although different countries have different privacy and security laws, acts, and regulations that govern the protection of data for example, the Asia Pacific Economic Cooperation (APEC) privacy framework, the Organisation for Economic Corporation and Development (OECD) privacy framework and the European Economic Area (EEA) data protection laws, the actual responsibility of ensuring that data and other resources outsourced to the cloud are secured and protected against data loss, damage, misuse usually rest with the custodian of the data - the CSP (CSA, 2011; OpenCirrus, 2017). However, given that data is the life blood of every serious organisation and that with cloud computing the subscriber's vital data asset is outsourced to third party organisation, it is critical the cloud subscriber take key interest in ensuring the safety of their data been outsourced for cloud storage. Hence, this study is of the same view with Fahmida (2016) that with cloud computing, the data owner (cloud tenant) even bears paramount responsibility in ensuring security of its data than the custodian of the data. Especially where critical business data such as trade secrets, financial data, employee data, or health data are been transferred for cloud storage. This assertion is more critical because when it comes to cloud computing service provision there is a chain of inter-dependency of services provisioning and hence tracing data leakage(s) could be extremely difficult. This study proposes secure data security architecture and system that ensures data is useful only to the owner.

Aim and Objectives - This paper implements a secured cloud data security solution in a defense-in-depth design based on the CDDI framework (Twum et. al. 2019) that alleviates the cloud subscriber's fear of their data security in respect of the data accessibility, data usage, data location, data accuracy, and data ownership. The objective of the study is to implement and test a total robust cloud security solution framework dubbed Cloud

Data Distribution Intermediary (CDDI) in terms of strength and performance against existing cloud security frameworks.

2. LITERATURE REVIEW

2.1 Review of Existing Cloud Storage Security Architectures

Subscriber → Provider (Direct) Model

The prevalent model among users of cloud storage is the direct link to the Cloud Storage Provider (CSP) via the web interface or the desktop/mobile client as shown in Figure 1. CSPs such as Google Drive, Dropbox and Box all provide an interface through which the client can upload or download files, and in the case of desktop or mobile interfaces, synchronize files and folders on their desktop or mobile device. In the direct model, data security while the file is being transferred and while it is residing in the cloud storage space, is the responsibility of the CSP (TipTopSecurity, 2016). Different CSPs implement security in a different way but the most common approach is to split the file into chunks on the subscriber's end, encrypt the chunks, then transfer the chunks individually to the provider's infrastructure over the internet.

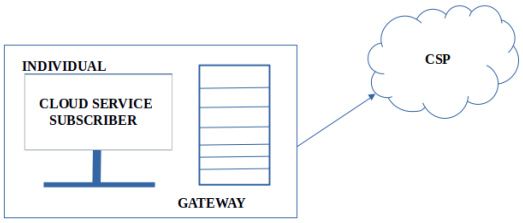


Figure 1 - Direct Model of Subscriber-CSP interaction

Subscriber -> Cloud Access Security Broker (CASB) -> Cloud Storage Provider (Indirect) Model

In this model, the subscriber adds to data security by involving a Security-As-A-Service (SECaaS) system in the cloud storage setup (Figure 2). Cloud Access Security Broker (CASB) systems are SECaaS implementations that function as a software guard for the data that moves around within and out of an organization. A CASB acts as an independent intermediary between a cloud subscriber and cloud provider. CASB's are on-premises or cloud-hosted software that sits between cloud subscribers and CSP's to enforce security, compliance, and governance policies for cloud usage (SkyHigh, 2017). By employing a CASB introduces an extra layer of security in the cloud environment which gives the subscriber security assurance and install some level of trust (DoubleHorn, 2017). CASB systems enforce the regulations on what data can be transferred in and out of the organization, especially to Cloud Servers (Rubens, 2017).

CASBs support a varying set of functions including, but not limited to, visibility into cloud usage within the organization, enforcement of compliance with the organization's regulations for cloud interaction, protection from external malware and a way to ensure that data is stored in the cloud securely. Notable CASBs are Forcepoint CASB, Skyhigh Networks, Cisco Cloudlock and Microsoft Cloud App Security. By way of securing data sent to the cloud, CASBs often encrypt the organization's data before upload. In cases where the data in the file is marked as too important to risk its contents being disclosed, the CASB protects the organization by preventing the upload of the file to the CSP. Figure 3 is an example implementation of CASB via Cloud Proxy.

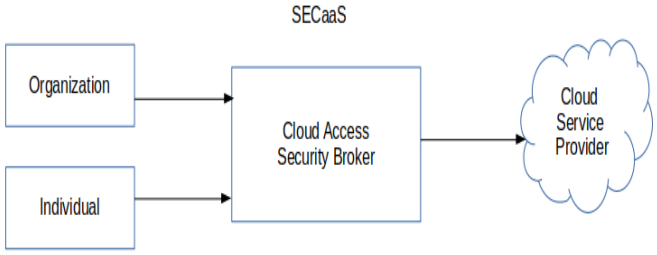


Figure 2 - Indirect Model of Subscriber-CSP Interaction Using Cloud Access Security Broker

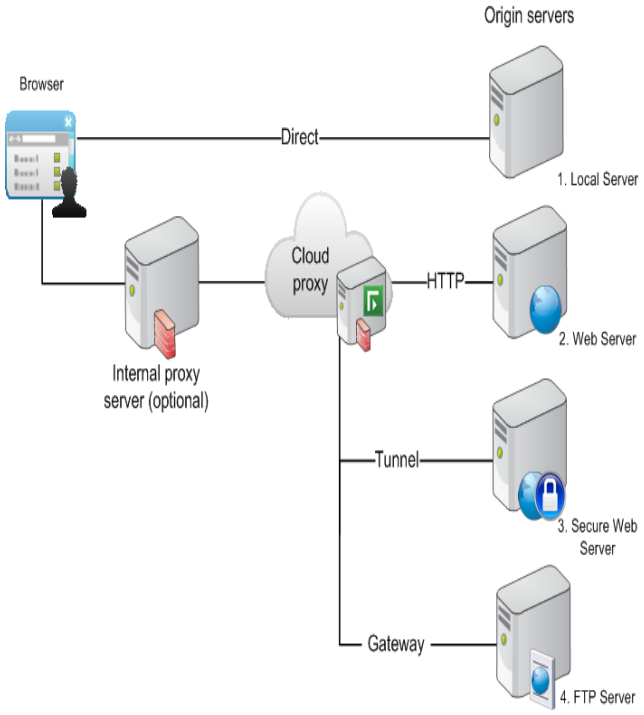


Figure 3 - Example Implementation of Cloud Access Security Broker via Cloud Proxy (Forcepont, 2017)

3. METHODOLOGY

The Cloud Data Distribution Intermediary (CDDI) framework (appendix 2) is implemented into software using the Java, SQL and PHP programming languages. The software is named SecureMyFiles (SMF) System.

3.1 The SecureMyFiles (SMF) System

The SecureMyFiles (SMF) system has modules as follows:

3.1.1 File Upload Module: The file upload module depicted by Figure 4 is sub-components SMF system. The user first selects the file to upload regardless of the size, format or type and whether zipped or unzipped. It should be noted though that the Java Virtual Machine (JVM) allocates memory for the program based on the hardware Random Access Memory (RAM). The implication here is that files with a substantially large size can cause an *OutOfMemory* Exception during runtime.

The file name is hashed using SHA-1 as the ***first layer*** of system security to obscure the identity of the file being uploaded. The system uses Secure Hash Algorithm 1 (SHA-1) from Java's inbuilt Security package. Though the Message

Digest 5 (MD5) was considered as an alternative hashing method but the MD5 algorithm is now widely regarded as cryptographically broken (Kessler, 2017). This means that the hash from an MD5 function can be reversed to obtain the original data which was hashed. As such, the MD5 algorithm was discarded in favor of SHA-1 which is still a one-way hashing scheme. Further, SHA-1 produces a longer string than MD5. This is useful in that it is more difficult to break a longer string than a shorter one.

As a ***second layer*** of security, the contents of the file are obfuscated (**transposed)** using the **encryption function** of **transposition cipher** algorithm based on the motions of the Rubik's cube to generate a cipher text (an encrypted output). The system uses a hash value based on the user's credentials as the encryption key. The system applies a custom algorithm to transform the hash value into a sequence of rotations. During a decryption, the rotation sequence is read from the last to the first and each rotation is carried out in the counter-clockwise direction. This undoes the clockwise rotations done during the encryption (Twum et.al., 2019a).

As a ***third layer*** of security, the encrypted output file is split into shards using a File Splitting and Erasure Protection Modules for example the Reed-Solomon coding technique or the Checksum Data Recovery technique (CDR) (Twum et.al., 2019b). To make use of Reed Solomon Coding, first an encoding polynomial is generated after which it is used to encode the data in a file (Twum et. al, 2016a, Twum et. al, 2016b, Twum et.al,2017). After the erasure protection techniques have been applied, the shards are saved as individual files on the disk. Figure 5 shows the results of splitting a file.

As a ***fourth layer*** of security, the shards upload to a particular cloud account is not done in the order which the splits are produced. Instead, the splits are ***distributed*** in a **non-deterministic** manner to the user's subscribed selected multiple cloud providers' storage facilities using a shuffling method. Since the data is uploaded in no particular order, it is difficult to determine the total number of shards from the content of one drive. The total number of shards is very necessary if an intruder wants to reconstruct the files from its shards.

A ***file's metadata*** file is used to keep record of which split chunks are sent to which cloud provider's facility for storage. Finally, the SMF System ***user metadata*** is updated to keep track of user's uploaded files and to help in retrieval of uploaded files from the various cloud accounts.

3.1.2 File Download Module: The download module is composed of 7 activities as shown by Figure 6

The user selects the file to download from their profile by clicking on the appropriate file name which is actually kept in the ***user metadata***.

The selected file name is ***hashed*** by applying *the same hash function* as that of the upload module to obtain the same hashed value if the file integrity has not been tempered with; the resulting hashed value is used with the file metadata file to obtain the required split chunks/shards from the user's cloud accounts storing them to a temporary storage (**a buffer area**).

The downloaded shards are then decoded or joined using the ***Reed-Solomon decoding method*** (Twum et. al., 2017) ***or the CDR decoding method*** (Twum et. al., 2019b).

The resultant file is ***re-transposed*** by applying the **decryption function** of the Rubik's Cube transposition cipher (Twum et. al., 2019a) used in the upload module.

The decrypted file output is renamed to the selected file name and delivered to the SMF user.

4. IMPLEMNTATION

The system uses the hashing methods in the java.security.MessageDigest package to hash the file name. The input to the system's hashing method is a string (the file name). The method returns the digest of the file name as a string. Listing 4.1 is a code snippet of the implementation of the hashing method.

```
public static String getHash(String txt){
   java.security..MessageDigest md =
                  MessageDigest.getInstance("SHA1");
   byte[] array = md.digest(txt.getBytes());
   StringBuilder sb = new StringBuilder();
   for (byte anArray : array)
      sb.append(Integer.toHexString((anArray & 0xFF) |
                        0x100).substring(1, 3));
   return sb.toString();
}
```

Listing 4.1 - getHash method to generate the hash of a string

Implementation Of File Splitting And Erasure Protection Module (FSEPM) Using Java

For the SMF system to guard against data loss or corruption, it makes use of Reed Solomon coding (Twum et. al, 2016a, Twum et. al, 2016b, Twum et.al,2017) or the CDR technique (Twum et.al., 2019b) to create parity data with which the file can be reconstructed in the event of data loss or corruption. The SMF system user is given the choice of choosing a file priority at the time of uploading a file to the cloud. The selected file priority determines the parameters for creating the parity data. The system provides four file priority levels as follows; "**Low**", "**Normal**", "**Important**" and "**Critical**."

Implementation of the Erasure Protection via Reed Solomon Coding

Reed Solomon Coding refers to a method of error detection and correction that computes recovery information before the file is transmitted. The recovery information, called "parity", is transmitted together with the file data. The presence of an "error" can be detected by examining the parity information while an "erasure" is the complete absence of a portion of the data. To perform a Reed Solomon Encoding, the system needs the number of data shards as well as the number of parity shards. These numbers are passed to the encoding method which breaks the file into "data" number of shards and computes "parity" number of metadata information which can be used to recover lost or corrupt data shards (Twum et. al, 2016a; Twum et. al, 2016b, Twum et. al, 2017).

The system receives a file priority setting from the user at the time of selecting a file for upload. Based on the file priority setting, the system determines the number of data and parity shards to use. The system can recover up to "parity" number of shard loss and can correct up to half "parity" number of shard corruption. This means that the larger the number of parity data, the more likely it is that the file can be reconstructed in

case there is some loss of data. Hence, files with a higher priority are given a larger number of parities. This makes recovering them more likely than recovering files with a lower priority. However, the higher the number of parity shards, the more the computation that goes into checking for errors and recovering missing or corrupt files. As such, files with higher priority take a longer time to reconstruct and also require more memory for the reconstruction operation. They also require more disk space locally and on the cloud.

The priority levels that use Reed Solomon coding and their associated data and parity shard counts are specified below.

Low - Files with "low" priority are split into 120 data shards and 24 parity shards, regardless of the size of the file. In other words, the splitting has no bearing on the size of the file. Each of the 6 CSPs receives 24 shards for storage. This offers the least protection since this configuration allows for 12 error correction and 24 erasure recovery, meaning that the subscriber can recover data if one of the CSPs is not available. However, the computation is fastest and uses less memory.

Normal - Files with "normal" priority are split into 96 data shards and 48 parity shards. Therefore, 24 errors can be corrected and 48 erasures can be recovered. This means that the data can be recovered even if two of the subscriber's CSPs are down. However, the encoding operation is slower than with the "low" priority files and it requires more memory.

Important - Files with "important" priority are split into 72 data shards and 72 parity shards, allowing 36 errors to be corrected and 72 erasure recoveries. In this configuration, the data can be recovered even if three CSPs are unavailable. This uses the most computational power as well as memory.

In all cases, SecureMyFiles breaks the file into a number of data shards and computes the parity shards so that the total number of shards is 144. Any number of shards from 2 to 256 can be used for the process, but 144 is an optimal value because it balances the computation time and strength of the Reed Solomon Encoding/Decoding process. In other words, using a total shard count of 256 would have been the most secure implementation of Reed Solomon encoding/decoding but that would likely make heavy use of the device's CPU and memory. Further, SMF prefers that the minimum number of CSPs connected to the SMF client is 6.

Listing 4.2 presents a snippet of code that shows how the data and parity shard counts are set based on the user's choice of a priority setting (i.e. Low, Normal, or Important).

```
if (filePriority.equalsIgnoreCase("low")){
   dataShards = 120;
   parityShards = 24;
}
else if (filePriority.equalsIgnoreCase("normal")){
   dataShards = 96;
   parityShards = 48;
}
else if (filePriority.equalsIgnoreCase("important")){
   dataShards = 72;
   parityShards = 72;
}
```

Listing 4.2 - Snippet of code to show how the data and parity shard counts are set based on the priority setting

Implementation of the Erasure Protection via the CDR Technique

Unlike the other file priority options, the fourth priority level of the SMF system dubbed "**critical**" does not use Reed Solomon coding to protect the data. Instead it employs the ***CDR*** technique that computes checksums for a file by taken bytes from several different sets of data bytes. The checksum information is stored in a metadata server for future reference in order to ascertain whether the user's data has been corrupted. Furthermore, the checksum data is used to correct errors within the file.

Encoding:

The processes involved in using the CDR technique are presented below.

Internal Data Representation:

The system first reads data into a three-dimensional array. The array has a special property that the cross section is 4×4 array that represents a single module (a data shard). The pseudocode in Listing 4.3 represents a method to perform the conversion of the file data into a three-dimensional array. Figure 7 illustrate the Operations of the CDR. It depicts the formation of shards from the modules of the CDR, using a 512-byte file.

```
function readFileToArray(file)
        oneDimensionalArray = readFileToByteArray(file)
        breadth = ceiling(oneDimensionalArray.length /
16.0)
        threeDimensionalArray = new array[breadth][4][4]
        a = 0
        for i=0 to breadth
                for j=0 to 4
                        for k=0 to 4
                          threeDimensionalArray[i][j][k]
=
                    oneDimensionalArray[a]
                          a++
                        endfor
                endfor
        endfor
endfunction
```

Listing 4.3 – Function to read from a file into a three-dimensional array

To implement above algorithm, the system read data from a file into a one-dimensional array using a method from the Apache Commons IO library. The system then writes the data into a three-dimensional array in order to get modules for the computation of parity information. Listing 4.4 shows a snippet of the code used to perform the data reading and conversion.

```
public Data(String fileName) throws Exception {
   this.filename = fileName;
   byte [] fileAsByteArray =
FileUtils.readFileToByteArray(new File(fileName));
   int i=0;
   int four = 4;
   int breadth = (int) Math.ceil(fileAsByteArray.length / 16.0 );
   array = new byte[breadth][four][four];
   for (int j=0; j<breadth; j++){
      for (int k = 0; k < four; k++){
         for(int l=0; l < four; l++){
            if (i < fileAsByteArray.length){
               array[j][k][l] = fileAsByteArray[i++];
            }
            else {
               array[j][k][l] = 1;
               i++;
            }
         }
      }
   }
}
```

Listing 4.4 – reading file to 3-dimensional array

Parity Data Computation:

Module Parity Computation - After reading data into a three-dimensional array, the system computes the parity value for each 4×4 module of the three-dimensional array. The pseudocode in Listing 4.5 represents a function to compute the module parity.

```
function computeModuleParity()
        sum = 0
        for row = 0 to 4
                for column = 0 to 4
                        sum = sum XOR
module[row][column]
                endfor
        endfor
        return sum
endfunction
```

Listing 4.5 – Function to compute the module's parity

Row and Column Parity - The system proceeds to compute the parity values for each row and each column, but instead of simply summing all the values in each row and column, the system sums all the different combinations of the row data as well as the different combinations of the column data. The code snippet of Listing 4.6 represents a function that computes the row and column parities.

```
for(int module = 0; module < this.data.length; module++){
        this.moduleParity[module] =
ComputeModuleSum(this.data[module]);
        int x =0, b = 0;
        for(int i =0; i < this.data[module].length; i++){
                for(int j =0; j < this.data[module][i].length;
j++){
                        this.rowsParity[module][i][6] ^=
this.data[module][i][j];
                        for(int k = j+1; k <
this.data[module][i].length; k++){

        this.colsParity[module][x][i] =
(byte)(this.data[module][j][i] ^ this.data[module][k][i]);
                                x++;
                        }
                        for(int l = j+1; l <
this.data[module][i].length; l++){

        this.rowsParity[module][i][b] =
(byte)(this.data[module][i][j] ^ data[module][i][l]);
                                b++;
                        }
                }
                x = 0;
                b = 0;
        }
        this.colsParity[module][6] = this.data[module][3];
}
```

Listing 4.6 – Function to compute row and column parities

The system computes the parity data for each module as well as the rows and columns in the modules. The parity for a module is computed as the XOR sum of all the elements in the 4×4 grid that made up the module and is stored in a single-dimensional array.

The code snippet in Listing 4.7 shows the implementation of the module parity computation.

```
private byte ComputeModuleSum(byte[][] data){
        byte sum = 0;
        for(int row = 0; row < data.length; row++){
                for(int col = 0; col < data[row].length;
col++){
                        sum ^= data[row][col];
                }
        }
        return sum; }
```

Listing 4.7 – Computation of module parity

The row parities are computed as the XOR sums of all the combinations of the elements in the module row. Each row's parity data is stored in a row of a two-dimensional array that holds the rows' parity information for that module. The system also computes the columns parity data as the XOR sums of all the combinations of the column elements for any single column in the module. Each column's parity data is stored in a column of a two-dimensional array that holds the columns' parity information for that module.

Listing 4.8 shows a snippet of code which implements the computations of the row's parity and column's parity data.

```
for(int module = 0; module < this.data.length; module++){
    this.moduleParity[module] = ComputeModuleSum(this.data[module]);
    int x =0, b = 0;
    for(int i =0; i < this.data[module].length; i++){
        for(int j =0; j < this.data[module][i].length; j++){
            this.rowsParity[module][i][6] ^= this.data[module][i][j];
            for(int k = j+1; k < this.data[module][i].length; k++){
                this.colsParity[module][x][i] = (byte)(this.data[module][j][i] ^ this.data[module][k][i]);
                x++;
            }
            for(int l = j+1; l < this.data[module][i].length; l++){
                this.rowsParity[module][i][b] = (byte)(this.data[module][i][j] ^ data[module][i][l]);
                b++;
            }
        }
        x = 0;
        b = 0;
    }
    this.colsParity[module][6] = this.data[module][3];
}
```

Listing 4.8 – Computation of row parity and column parity

Writing Parity Data to File:
The system creates a parity object from the computed parity information and writes the data to three files for the module parity, row parity and column parity. The code snippets in Listing 4.9 and Listing 4.10 respectively shows implementations of how the parity objects are populated and written to file.

```
Parity p = new Parity();
p.setColumn(colsParity);
p.setRow(rowsParity);
p.setModule(moduleParity);
```

Listing 4.9 – Population of the parity object

```
public void writeToFile(String fileName) throws IOException
{
    int length = row.length * 28;
    byte[] array = new byte[length];
    File dir = new File(DIR);
    if (!dir.exists()) dir.mkdir();
    fileName = DIR + "/" + fileName;
    FileUtils.writeByteArrayToFile(new File(fileName + ".module"), module);
    int i = 0;
    for (byte[][] a : row){
        for (byte[] b : a){
            for (byte c : b){
                array[i++] = c;
            }
        }
    }
    FileUtils.writeByteArrayToFile(new File(fileName + ".row"), array);
    i = 0;
    for (byte[][] a : column){
        for (byte[] b : a){
            for (byte c : b){
                array[i++] = c;
            }
        }
    }
    FileUtils.writeByteArrayToFile(new File(fileName + ".col"), array);
}
```

Listing 4.10 – Writing parity data to file

Data Splitting:
After the parity data has been computed for the file data, the system splits the file into 16 shards. Each shard comprises of all module elements of a particular location in the 4×4 grid. That is, every first element in the 4×4 grid is collected into one shard. The algorithm in Listing 4.11 represents a function to split the data into shards for uploading to the cloud.

```
function splitData()
        splits = new
array[16][threeDimentionalArray.length]
        for i = 0 to 4
                for j = 0 to 4
                        current = (4 * i) + j
                        for k = 0 to
threeDimensionalArray.length
                                splits[current][k] =
threeDimensionalArray[k][i][j]
                        endfor

        writeByteArrayToFile(splits[current])
                end for
        end for
end function
```

Listing 4.11 – Function to split file into shards

The algorithm is implemented using the Apache Commons IO Java library (Listing 4.12).

```
public void split(Data data){
        String dir = "temp/split";
        File directory = new File(dir);
        if (!directory.exists()) directory.mkdirs();
        byte [][] splits = new byte[16][data.length];
        for (int i = 0; i < 4; i++){
                for (int j = 0; j < 4; j++){
                        int a = 4 * i + j;
                        for (int k = 0; k < data.length;
j++){
                                splits[a][k] =
data[k][i][j];
                        }

        FileUtils.writeByteArrayToFile(splits[a], new
File(dir + "/" + data.fileName + "." + a));
                }
        }
}
```

Listing 4.12 – Splitting data into 16 shards

Checksum Data Recovery (CDR) Technique Metadata:
There are three types of metadata, the rows parities metadata, columns parities metadata, and module sum parity.

Module sum parity is a single dimensional array of length similar to the length of the 3D data array. The entry values are computed from the XOR of all entries of a module and stored in the array. The module sum parity array has two main benefits:

- It ensures no module is dependent on another module (i.e. module abstraction).
- It reduces unnecessary iterations. Thus, the program checks for corrupted entries in a module only when the sum of entries in the module is not equal to the value of module sum parity entry at that index.

Each of rows parities and columns parities metadata is a 3D array of size similar to the 3D data array computed from the data file. Each entry in the metadata array is a 2D array. Which implies the 3D metadata arrays also contains modules in its entries.

A module in the rows parities array is a 4 x 4C2+1 matrix and the columns parities array is a 4C2+1 x 4 matrix, where C is combination. Thus, for a byte of data to be uniquely recovered in the event of an error, each byte in a module must have relationships with entries on the same row and also entries on the same column. For example, for a module of size 4x4 matrix, entry a(0,0) is related to entries a(0,1), a(0,2) and a(0,3) as these are entries are on the same row. These relationships are stored as row parities metadata. Similarly, entry a(0,0) is also related to entries a(1,0), a(2,0) and a(3,0) as these are entries on the same column. These relationships are also stored as columns parities metadata.

Clearly there is a relationship between any two entries on the same row or column as there are four entries on each row and four on each column of a module. The parity metadata for a given file is obtained by computing the relationships on each row as 4 combination 2 (i.e. taking 2 entries at a time). The combination (not permutation) used here is important because the order of combining is not important since the relationship is computed as the XOR of each pair of entry and this minimizes the size of the metadata arrays.

The additional column on the row parities array holds the XOR of all entries in a row. This column is very important because the algorithm uses the value held in the column to recover corrupted entries on the row when the sum of the row does not equate to the value held.

The additional row in a column parities array module contains the AND of the last row and itself. This is important because the whole error detection algorithm will use the last row as its base or reference point.

How a Module Locates Its Metadata in the Checksum Data Recovery (CDR) Technique
Given a module M [4] [4] at index [x] [4] [4] in the 3D data array,

- Its corresponding module sum parity is at index [x] within the module sum parity metadata array (a single dimensional).
- Its rows parity module is at index [x] of the rows parity metadata array.
- Its columns parity module is at index [x] of the columns parity metadata array.

Implementation of the Data Dispersal Technique (The Shuffling Method)
The shuffling is implemented using the Java Collection class in the Utility package. An array list is created to contain integer numbers that correspond to the number of the derived shards after splitting of the file. The shuffling of the integer numbers in the array list is achieved by passing the array list as a parameter to the static Shuffle method called from the Collections class. Appending the numbers from the newly shuffled list to the original file name, a new name is formed (filename.extension.number) which is used to get the individual shards from the source (temporary storage where the shards are kept after splitting) to be ready for upload. Now the data shards are distributed depending on the SMF user's selected number of cloud providers. For instance, if two cloud storage providers are selected (i.e. Dropbox and Box), then half

of the appended list are randomly distributed to the Dropbox and the other half to the Box. Figure 8 is the snippet code of the implementation of the data dispersion technique.

Implementation of SMF System's Metadata
In order to keep track of file information, the system uses a metadata file (dubbed, ***file metadata*****)** to record information about the file and its shards. The information the system captures includes:

1. File Name
2. File Size
3. File Priority
4. Date of File Upload
5. Number of Data Shards
6. Number of Parity Shards
7. Cloud Service Destinations for the Shards

The system hosts the *file metadata* on SMF servers for easy access from multiple devices and locations. A plain text file is used to hold the metadata information. The data is encoded in ***JavaScript Object Notation (JSON)***. The **Google *Gson library*** is used for encoding and decoding the metadata.

After a file is selected for upload and the file priority is set, a metadata object is created and the fields in the object are set with the file name, size, priority, number of data and parity shards. After the shards are shuffled and assigned cloud accounts, the destination information is also captured into the metadata object (Listing 4.13).

The system uses the Gson library to create a JSON string from the metadata object. The string is then written to a file with the hash of the original file name as the name of the metadata file (Listing 4.14). The extension for metadata files is "meta". When the system is done creating and updating the contents of the metadata file, the file is uploaded to the SMF Metadata Server for safekeeping.

```
metadata.File metadata =
    new metadata.File(tempFile.getName(),
        Date.from(Instant.now()),
        filePriority,
        dataShards,
        parityShards,
        cols,
        size);
```

Listing 4.13 - Instantiation of a metadata file object

```
public static void writeFile(String content, File file) {
   FileWriter writer = new FileWriter(file);
   writer.write(content);
   writer.close();
}
```

Listing 4.14 - Method to write metadata string to file

Shards Upload Module
After the shards have been prepared for upload, SMF makes use of the Application Programmer Interface (API) of the Cloud Service Provider to send the file over the internet to the CSP. Listing 4.15 shows a snippet of code from the SMF that does the file upload to Dropbox and Box.

```
if (cloud.equalsIgnoreCase("dropbox"))
        try {
                dbx.uploadFile(f);
        } catch (IOException | DbxException e) {
                e.printStackTrace();
        }
else if (cloud.equalsIgnoreCase("box"))
        try {
                box.uploadFile(f);
        } catch (IOException e) {
                e.printStackTrace();
        }
```

Listing4.15 - Snippet of code showing file upload to Dropbox and Box

File Downloading Process
When the user selects a file to download, the system downloads the metadata file from SecureMyFiles servers. The system then read the text from the metadata file as a string (Listing 4.16) and then uses a method from the Gson library to create a metadata object that contains all the information in the file (Listing 4.17). The selected file name is hashed by applying the same hash function as that of the upload module to obtain the same hashed value if the file integrity has not been tempered with; the resulting hashed value is used with the file metadata file to obtain the required split chunks or shards from the user's cloud accounts storing them. The downloaded shards are then decoded or joined through using the ***Reed-Solomon decoding method***. The resultant file is ***re-transposed*** by applying the decryption function of the proposed Rubik's Cube transposition cipher (Twum et. al., 2019a) used in the upload module. The decrypted file output is renamed to the selected file name and delivered to the SMF user.

```
public static String readFile(String fileName){
   String fileContent = "";
   try {
      Scanner scan = new Scanner(new java.io.File(fileName));
      while (scan.hasNextLine())
         fileContent += scan.nextLine();
   } catch (FileNotFoundException e) {
      e.printStackTrace();
   }
   return fileContent;
}
```

Listing 4.16 -Method to read text from a file to a string

```
String metadata = FileHandler.readFile("metadata/" + hash + ".meta");

metadata.File meta = new Gson().fromJson(metadata, metadata.File.class);

List<Destination> destinations = meta.getDestinations();
```

Listing 4.17 - Snippet of code showing how the system creates a metadata object from a file and accesses the list of destinations from the object

CDR decoding:

Reconstruction of the file from file shards

The system read the shards that are downloaded from the cloud and write them into a three-dimensional array representing the data. Listing 4.18 shows a code snippet of the function that joins the file shards to form a three-dimensional array.

```
public byte[][][] join(String fileName){
        String dir = "temp/split";
        File directory = new File(dir);
        if (!directory.exists())
                throw new FileNotFoundException("No 'split' directory");
        byte[] bytes = null;
        int length = FileUtils.readFileToByteArray(new File(dir + "/" + fileName + "." + 1)).length;
        byte[][][] data = new byte[length][4][4];
        for (int i = 0; i < 4; i++){
                for (int j = 0; j < 4; j++){
                        int a = 4 * i + j;
                        bytes = FileUtils.readFileToByteArray(new File(dir + "/" + fileName + "." + a));
                        for (int k = 0; k < bytes.length; k++){
                                data[k][i][j] = bytes[k];
                        }
                }
        }
}
```

Listing 4.18 – Reading file shards to form a data array

Reading Parity Data from Files

The system read the parity data from three files, one for the row parity, one for the column parity and a last one for the module parity. The data from the files are used to populate a parity object. Listing 4.19 shows a code snippet used for the reading of files into the parity object.

```
public void readFromFiles(String fileName) throws IOException {
   fileName = DIR + "/" + fileName;
   byte[] rowArray = FileUtils.readFileToByteArray(new File(fileName + ".row"));
   byte[] colArray = FileUtils.readFileToByteArray(new File(fileName + ".col"));
   module = FileUtils.readFileToByteArray(new File(fileName + ".module"));
   int length = rowArray.length / 28;
   row = new byte[length][7][4];
   column = new byte[length][4][7];
   int a = 0;
   for (int i = 0; i < row.length; i++) {
      for (int j = 0; j < row[i].length; j++) {
         for (int k = 0; k < row[i][j].length; k++) {
            row[i][j][k] = rowArray[a++];
         }
      }
   }
   a = 0;
   for (int i = 0; i < column.length; i++) {
      for (int j = 0; j < column[i].length; j++) {
         for (int k = 0; k < column[i][j].length; k++) {
            column[i][j][k] = colArray[a++];
         }
      }
   }
   del(fileName); }
```

Listing 4.19 – Reading parity data from files

Error Location

The system re-computes the module parity for the three-dimensional data array and compare the computed parities with the module parity data that was read form the file. The two parity information are compared for equality. Where there is inconsistency, the data is flagged as corrupt and the system proceeds to check the rows and columns for the offending array element (Listing 4.20).

```
void LocateDefectedModule(){
        boolean errorExist = false;
        for(int i =0; i < data.length; i++){
                if(ComputeModuleSum(data[i]) != moduleParity[i]){
                        System.out.println("module "+(i+1)+" has error");

        LocateErrorInModule(this.data[i],i);
                        errorExist = true;
                }
        }
        if(!errorExist)
          System.out.println("no error exist");
}
```

Listing 4.20 – Checking modules for errors

Error Correction

The system resolves the data corruption by looking through the rows parity and columns parity that is read from file and

performs an XOR sum at the appropriate locations to correct the error

Writing Corrected Data to File
The system finally writes the corrected data to file. Listing 4.21 shows the function that writes the data to file.

```
public void writeToFile() throws IOException {
int length = array.length * 16;
 int a = 0;
byte[] bytes = new byte[length];
for (byte[][] b : array){
for (byte[] c : b){
for (byte d : c)
bytes[a++] = d;
}
}
 a = length - 1;
 while (bytes[a] != 1)
a--;FileUtils.writeByteArrayToFile(new File(filename),
Arrays.copyOf(bytes, a + 1));
}
```

Listing 4.21 – writing data from the three-dimensional array to file

5. TESTING, RESULTS AND DISCUSSIONS

The sections that follow disclose the process used to test various sub-systems that were implemented and integrated to create the SMF system and the results obtained. The testing was carried out in an experimental lab set-up using JAVA, SQL, and PHP software development tools installed on a very high-spec PC (64-bit Intel Core-i7 CPU running at 3.60GHz, 12.0GB RAM) and Laptop (64-bit Intel Core-i7 CPU running at 2.20GHz, 8.0GB RAM). In addition, the experimental setup required a stable computer network infrastructure. Various testing scenarios were set-up for experimentations to evaluate the SMF system capability of recovering a file in an event of corruption (whether shards modifications or shards deletion). Results from the experiments conducted using the Reed Solomon Erasure protection and also the CDR erasure protection under different testing scenarios are presented later in this section.

Cloud Service Providers
There are a number of cloud service providers that provide premium and free services. Those that are available for use with the SMF App are Dropbox, Google Drive, and Box. Each of these cloud accounts is connected to the SMF App via their respective Java APIs. These APIs provide safe and secure connections to the cloud accounts just as it is when accessing an account on the cloud provider's website. A user needs to sign up to a minimum of two cloud accounts and a maximum of six e.g. Dropbox, Google Drive, iCloud, and Box before using the SMF App. For testing the proposed system, three cloud accounts were used. However, six is highly recommended for a good balance between performance and security.

5.1 RESULTS

File Uploading Sequence
The final phase of the upload process is to connect to the cloud service through the service provider's API and send the file over secure HTTP into the user's cloud account. Depending on the SMF user choice of a file priority level (Low, Normal, important, or critical) the system distributes shards to the user's cloud accounts in accordance to the data and parity computation as set out in section 4 above. Testing and results from the selection of the 'Normal' and 'Important' file priority levels which employs the Reed Solomon Erasure protection for file recovery under different scenarios are presented (See appendix 1 - Scenarios 1 and 2). The test results from the critical priority level which uses CDR described in Section 4 is also presented (See appendix 1 - Scenario 3). By choosing the critical file priority option, the SMF user may lose up to 12 shards out of the 16 total shards for any file size uploaded (Refer to figure 7 in section 4) and still be able to recover the corrupted file.

File Downloading Sequence
The file downloading sequence follows the process outlined and implemented by the file download module presented in section 3 (See Figure 6).

Scenario 1: Downloading a file uploaded using the 'Normal' file priority level.

Case 1
Figure 11 (Appendix 1) depicts file reconstruction during download for a scenario where 24 of the distributed shards are corrupted or an event where one of the six CSP's refuses to grant a subscriber access to shards stored on their storage servers in the event of a dispute over say subscription payments or for any other reason.

Case 2
Figure 12 (Appendix 1) depicts a situation where 48 shards are corrupted or where two of the CSP's systems are down but the SMF system is able to recover the full file and present to the owner.

Case 3
Unlike Case 1 and Case 2, Case 3 presents a scenario that depicts a situation where more than 48 shards are corrupted i.e. when more than two CSP's systems are inaccessible. As described above, the 'Normal' file priority level stores 48 parity information and hence cannot be able to recover data corruption of more than 48 shards. Hence as depicted in figure 13 (Appendix 1) the file recovery process failed for files uploaded using the 'Normal' priority level with more than 48 corrupted shards.

Scenario 2: Downloading a file by choosing the 'Important' file priority level.

Case 1
The 'Important' priority level when selected splits a file into 72 data shards and 72 parity shards that are used for file recovery in the event of corruption making a total of 144 shards that are uploaded to the six CSP's using the data dispersal technique. Although slower in recovering a file than the 'Normal' priority level, the 'Important' priority level enables a file to be recovered even if 72 shards are corrupted or when three CSP's servers are inaccessible.

Figure 14 (Appendix 1) depicts a scenario where 72 of the distributed shards are corrupted or a situation where three of the six CSP's servers are unreachable but the SMF system still recovered the full file during download.

Case 2
Case 2 presents a scenario that depicts a situation where more than 72 shards are corrupted i.e. when more than three of the CSP's systems are inaccessible. As stated above, the 'Important' file priority level stores 72 parity information and hence cannot recover data corruption of more than 72 shards. Therefore, as depicted in figure 15 (Appendix 1) the file recovery process failed.

Scenario 3: Downloading a file by choosing the 'Critical' file priority level.

Case 1
Figures 16, 17, and 18 respectively (Appendix 1) depict a scenario where 4, 8 and 12 of the distributed shards are corrupted during download and the SMF system reconstructs the file successfully with Critical option.

Case 2
It was realized during testing as shown by Figure 19 (Appendix 1) that the 'Critical' priority option cannot recover a file when more than 12 of the shards are corrupted.

6. CONCLUSION
The SMF system with its characteristics provides solutions to the cloud data security challenges outlined by this study. The system is unique as no single product of its kind was found in the market. All the existing cloud security solution systems either secures subscribers data on a single CSP's infrastructures (Direct model) or employ the service of SECaaS provider through setting up regulations and policies via a CASB. Appendix 3 presents how the SMF systems which is based on the CDDI model compares with the existing systems.

7. REFERENCES
[1] O'Reilly, J. (2017). *7 Ways to Secure Cloud Storage*. [Online]Availablefrom:https://www.networkcomputing.com/data-centers/7-ways-secure-cloud-storage/866645128 [Accessed: 15th Oct., 2017]

[2] The Treacherous 12, (2017). The Treacherous 12: Cloud Computing Top Threats in 2016 [Online]. Avalable from: http://www.storm-clouds.eu/services/2017/04/the-treacherous-12-cloud-computing-top-threats-in-2016/ [Accessed: 25th October, 2017]

[3] Shapland, R. (2017). Multi-Tenancy Cloud Security Requires Enterprise Awareness. Available from: http://searchcloudsecurity.techtarget.com/tip/Avoid-the-risks-of-multi-tenant-cloud-environments-through-awareness [Accessed: 10th October, 2017]

[4] CSA (2011). Security guidance for critical areas of focus in cloud computing V3.0. [Online] Available from: https://cloudsecurityalliance.org/guidance/csaguide.v3.0.pdf [Accessed: 10th May, 2015]

[5] OpenCirrus (2017). *Cloud Computing Challenges In 2017*. [Online] Available from: http://www.opencirrus.org/cloud-computing-challenges-2017/ [Accessed: 4th Sept., 2015]

[6] Fahmida Y. R. (2016). The dirty dozen: 12 cloud security threats. [Online]. Available from: https://www.infoworld.com/article/3041078/security/the-dirty-dozen-12-cloud-security-threats.html [Accessed: 15th March, 2017]

[7] Twum F., Hayfron-Acquah J. B, Panford J.K. A Proposed New Framework for Securing Cloud Data on Multiple Infrastructures using Erasure Coding, Dispersal Technique and Encryption, International Journal of Computer Applications, Vol. 181, No. 50, pp. 38-49, April 2019.

[8] SkyHigh (2017). What is CASB? Available from: https://www.skyhighnetworks.com/cloud-security-university/what-is-cloud-access-security-broker/ [Accessed: 10th October, 2017]

[9] DoubleHorn,(2017). Cloud Services Brokers: The future of SaaS and IaaS Consumption [Online]. Available from: https://doublehorn.com/cloud-services-brokers-the-future/ [Accessed: 15th October, 2017]

[10] Rubens, P. (2017). Six Top CASB Vendos. [Online]. Available from: https://www.esecurityplanet.com/products/top-casb-vendors.html [Accessed: 5th Nov., 2017]

[11] Forcepoint, (2017). How Forcepoint Web Security Cloud Works. [Online]. Available from: https://www.websense.com/content/support/library/web/hosted/getting_started/cws_explain.aspx [Accessed: 15th Oct., 2017]

[12] Kessler, G. C. (2017). An overview of Cryptography. [Online]. Available from: http://www.garykessler.net/library/crypto.html [Accessed: 1st May, 2017]

[13] Twum F., Hayfron-Acquah J. B, Morgan-Darko W., (2019a). A Proposed Enhanced Transposition Cipher Algorithm Based on Rubik's Cube Transformations, International Journal of Computer Applications, Vol. 182, No. 35, pp 18-26, January 2019.

[14] Twum F, Hayfron-Acquah, J. B., Oblitey, W. W., Morgan-Darko, W., (2016a). Reed Solomon Encoding: Simplified explanation for Programmers. *International Journal of Computer Science and Information Security (IJCSIS)*, Vol. 14, No. 12

[15] Twum F, Hayfron-Acquah, J. B., Oblitey, W. W., Boadi, R. K., (2016b). A proposed algorithm for generating the Reed-Solomon Encoding Polynomial Coefficients over GF(256) for RS[255,223]8,32. *International Journal of Computer Applications (IJCA)*, Vol. 156, No. 1, pgs. 24-39

[16] Twum F, Hayfron-Acquah, J. B., Oblitey, W. W., Morgan-Darko, W., (2017). Reed Solomon Decoding Simplified for Programmers. *International Journal of Computer Science and Information Security (IJCSIS)*, Vol. 15, No. 1

[17] Twum F., Hayfron-Acquah J. B, Panford J.K., (2019b). A Proposed New Framework for Securing Cloud Data on Multiple Infrastructures using Erasure Coding, Dispersal Technique and Encryption, International Journal of Computer Applications, Vol. 181, No. 50, pp. 38-49, April 2019.

[18] TipTopSecurity,(2016).Is Google Drive Safe to Use? How Google Secures Your Files Online [Online]. Available from: https://tiptopsecurity.com/is-google-drive-safe-to-use/ [Accessed: 1st Nov., 2017]

[19] Chima, R. (2016). Cloud Security – Who owns the data? [Online]. Available from: https://www.bbconsult.co.uk/blog/cloud-security-who-owns-the-data [Accessed: 20th February, 2016]

[20] FileCloud, (2016). Data Ownership in the Cloud – How does it affect you? [Online]. Available from: https://www.getfilecloud.com/blog/2016/11/data-ownership-in-the-cloud-how-does-it-affect-you/#.WgG7_I-0Pct [Accessed: 15th March, 2017]

File Upload Module

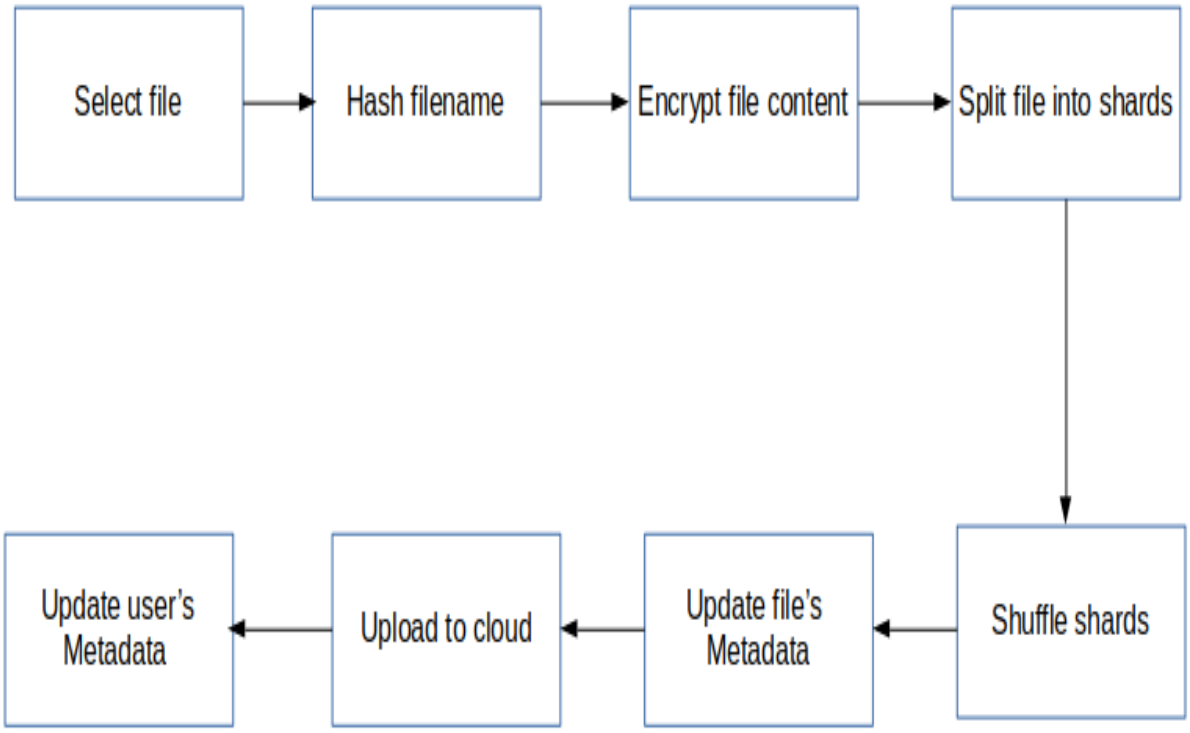


Figure 4 – File Upload Module

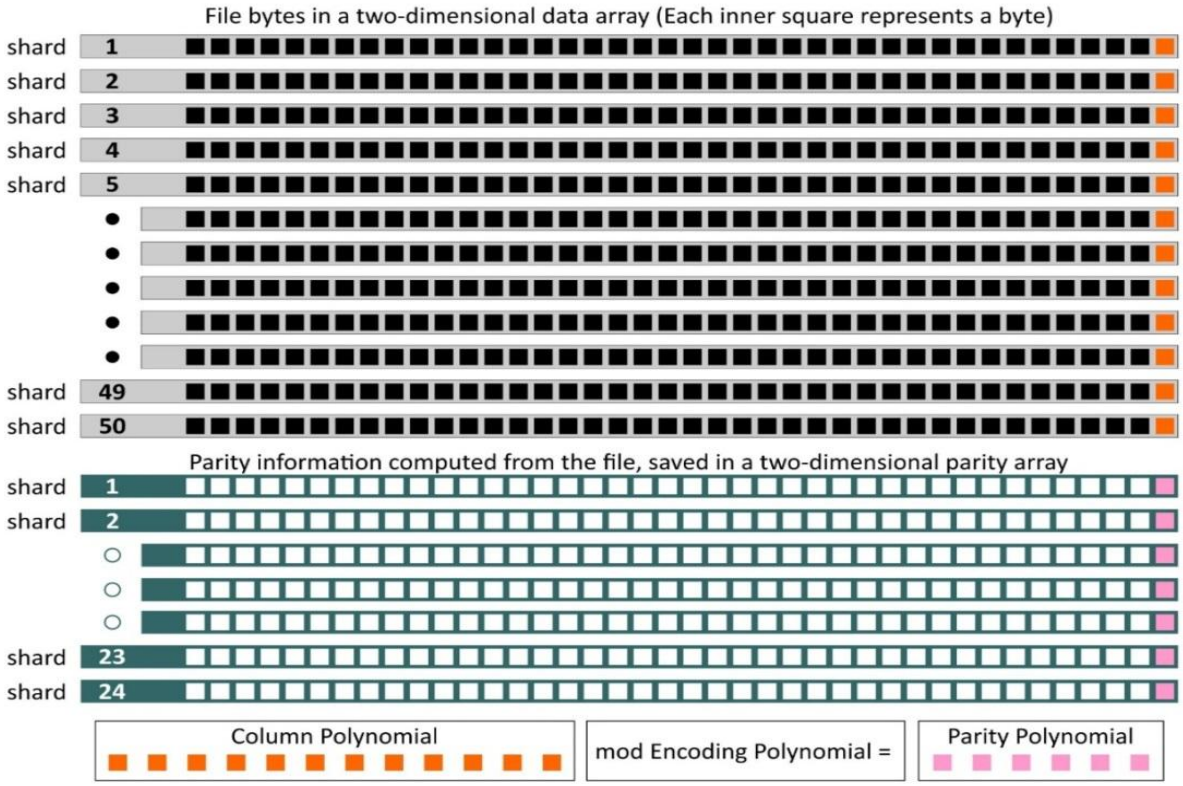


Figure 5 - Splitting of file and computation of parity.

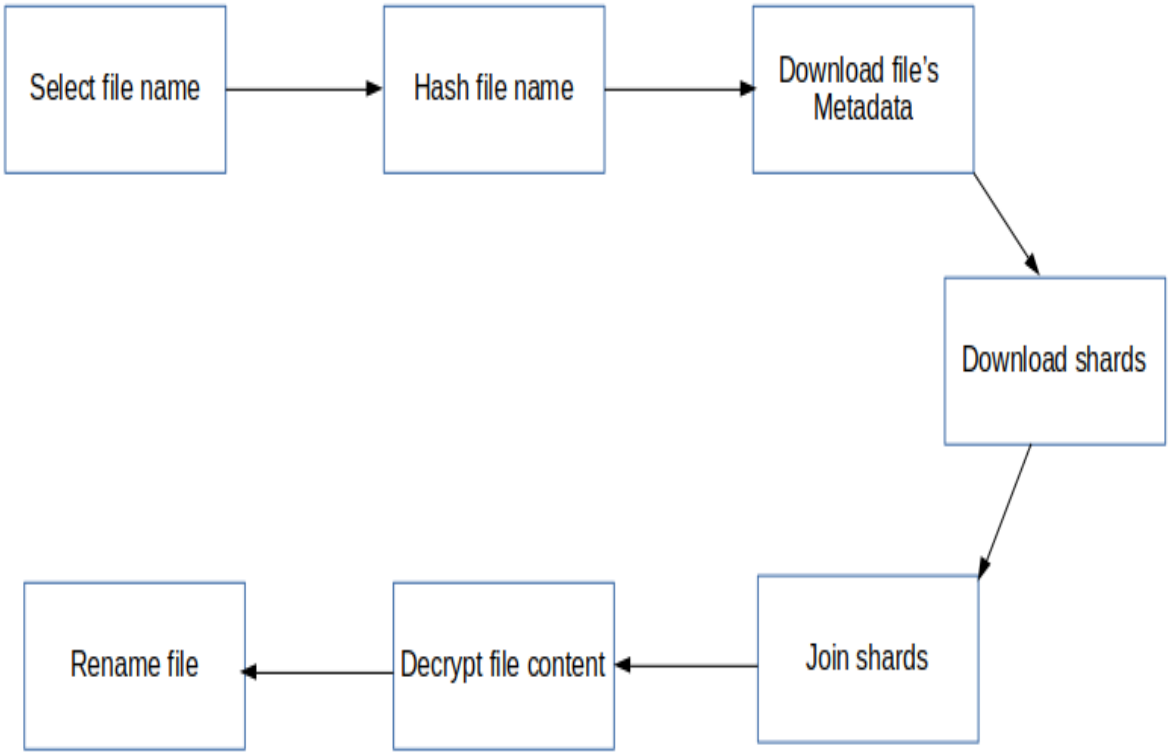


Figure 6 - File Download Module

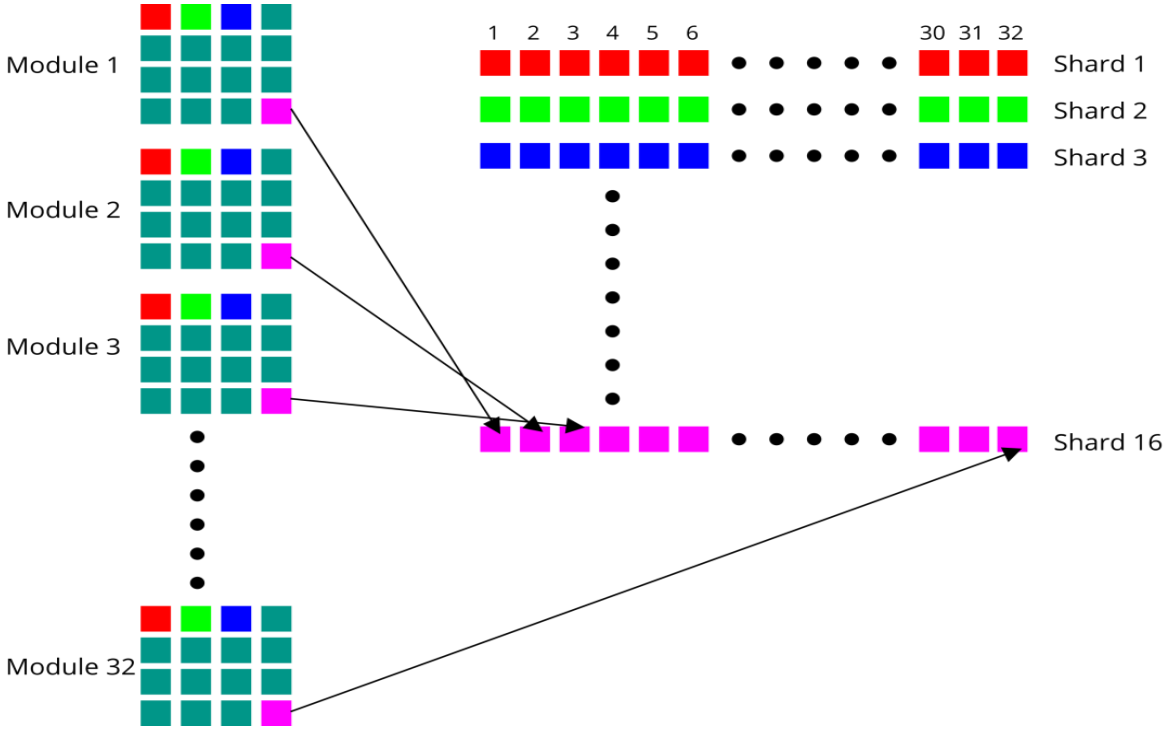


Figure 7: The formation of shards from the modules of the CDR using a 512 byte file.

```
47      //define ArrayList to hold Integer objects
48      ArrayList<Integer> numbers = new ArrayList<>();
49
50      //populate the Integer ArrayList
51      for(int i = 0; i < shards; i++){
52              numbers.add(i);
53              }
54      // shuffle the Integer entries
55      Collections.shuffle(numbers);
56      System.out.println(numbers);
57
58      //shards are now distributed based on the no. of cloud storage.In this case 2
59      for(int j = 0; j < shards; j++){
60              try{
61                 if(j< shards/2) {
62                      File afile =new File(source + "." + numbers.get(j));
63                      if(afile.renameTo(new File(destA + afile.getName()))){
64                          System.out.println(afile.getName() + " shard moved to "+ destA + " successfully!");
65                      }else{
66                           System.out.println(afile.getName() + " shard failed to move to " + destA);
67                      }
68                   }else{
69                      File afile =new File(source + "." + numbers.get(j));
70                      if(afile.renameTo(new File(destB + afile.getName()))){
71                          System.out.println(afile.getName() + " shard moved to "+destB+" successfully!");
72                      }else{
73                      System.out.println(afile.getName()+ " shard failed to move to "+destB);
74                      }
75                   }
```

Figure 8 - Code for implementation of the data dispersion technique (the shuffling method)

Appendices

APPENDIX 1

Scenario 1: Uploading a file by choosing the 'Normal' file priority level.

The '**Normal**' priority level splits a file into 96 data shards and 48 parity shards that are used for file recovery in the event of corruption. This result with a total of 144 shards that are uploaded using the data dispersal method to the six CSP's each receiving 24 shards.

Case 1

Figure 9 depicts a successful file upload operation where the SMF user selected the '**Normal**' priority level. The figure indicates none of the distributed shards are corrupted.

Upload Details

Summary

File Name: test_file.mp3
Hashed Name: 9d347e3564d127dcac9a1a7d28315e04d69a51fd
File Priority: Normal [96 data shards, 48 parity shards]
Cloud Services Signed In: 6 [GDrive(1), GDrive(2), Dropbox, Box(1), Box(2), OneDrive]
Number of successfully uploaded shards: 144
Number of failed shard uploads: 0
Total number of file shards: 144

Shard Upload Details

Shard Number	Uploaded To	Success/Fail	Remarks
1	DropBox	Success	None
2	Box(2)	Success	None
3	GDrive(2)	Success	None
4	DropBox	Success	None
5	Box(1)	Success	None
6	Box(2)	Success	None
7	Box(2)	Success	None
8	Box(2)	Success	None
9	Box(2)	Success	None

Figure 9 – A successful file upload choosing the Normal file priority level

Scenario 2: Uploading a file by choosing the 'Critical' file priority level.

Reed Solomon coding can recover any number of data errors up to half the number of parity data stored and can correct any number of erasures up to the number of parity data stored (Twum et. al., 2016b). This feature of Reed Solomon coding places some limitation on the number of data that can be successfully recovered in the event of data corruption. The proposed CDR which although cannot recover data in the event of total deletion without relying on backup can in most cases recover a file to some extent if at least four of the data shards exist. The '**Critical**' file priority option uses the CDR for erasure protection. Case 1 depicts a successful file uploading operation where the SMF user selected the '**Critical**' priority level.

Case 1

Figure 10 - Depicts a successful file upload operation where the SMF user selected the '**Critical**' priority level. The figure indicates none of the distributed shards are corrupted.

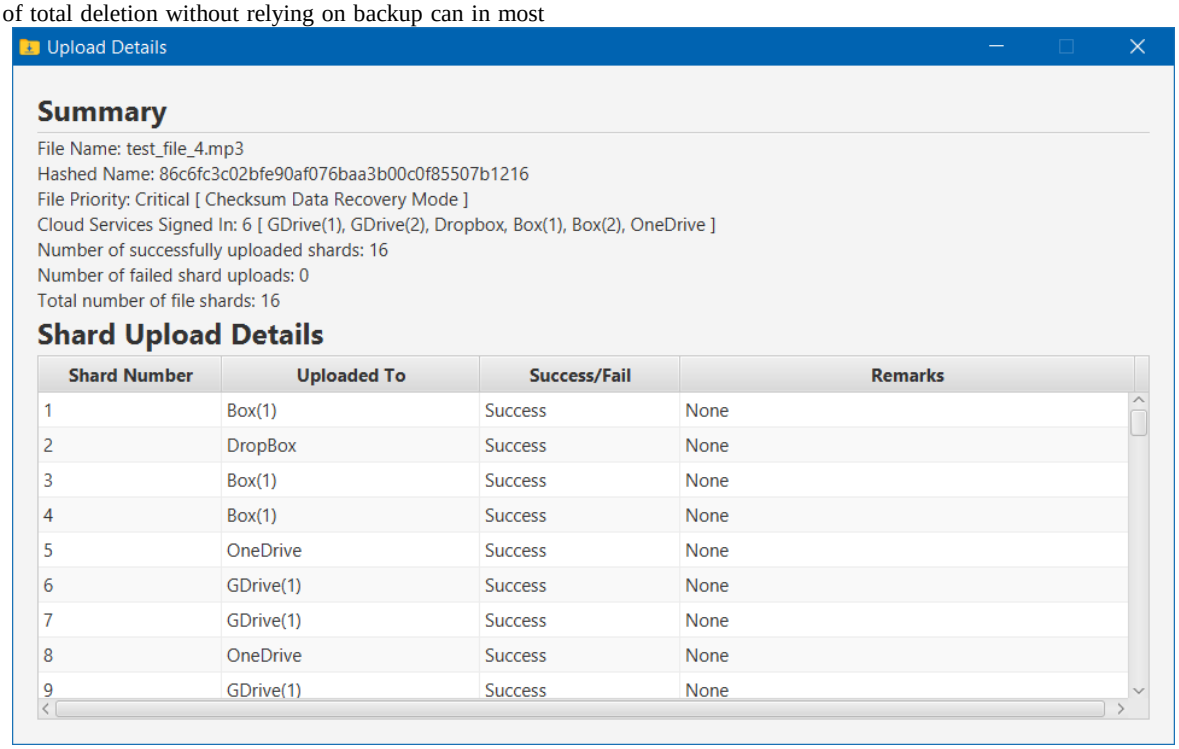


Figure 10 - A successful file upload choosing the Critical file priority level

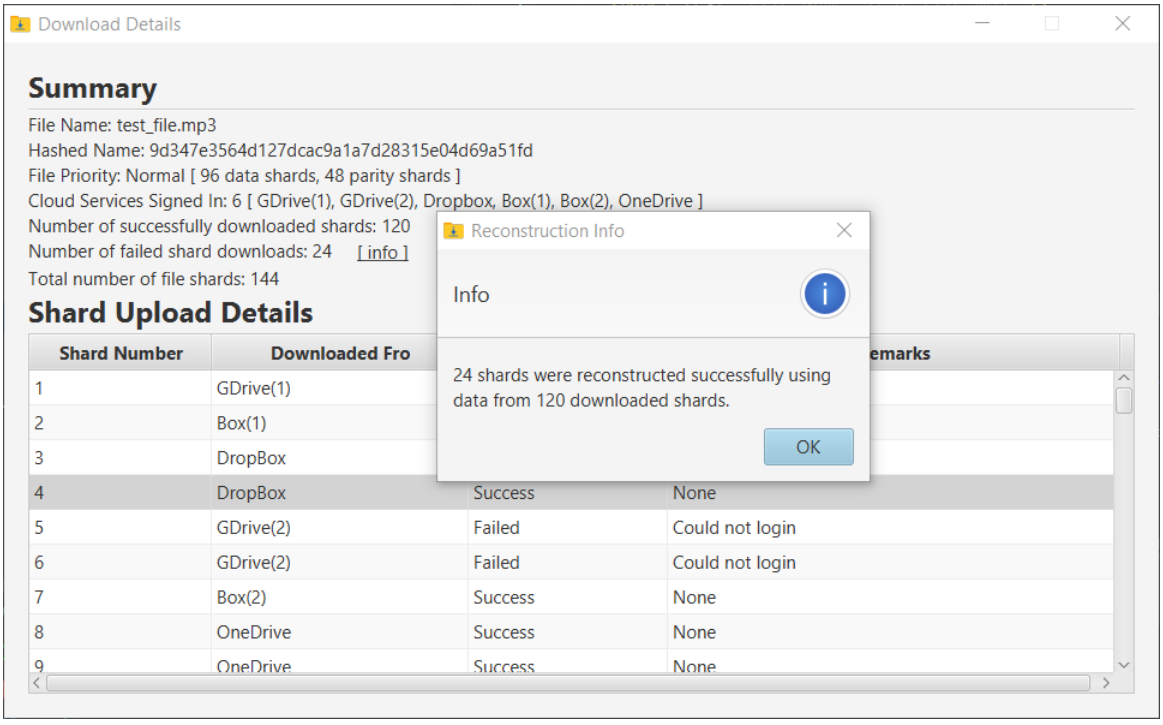


Figure 11 – successful file reconstruction during download for 24 corrupted shards with Normal option

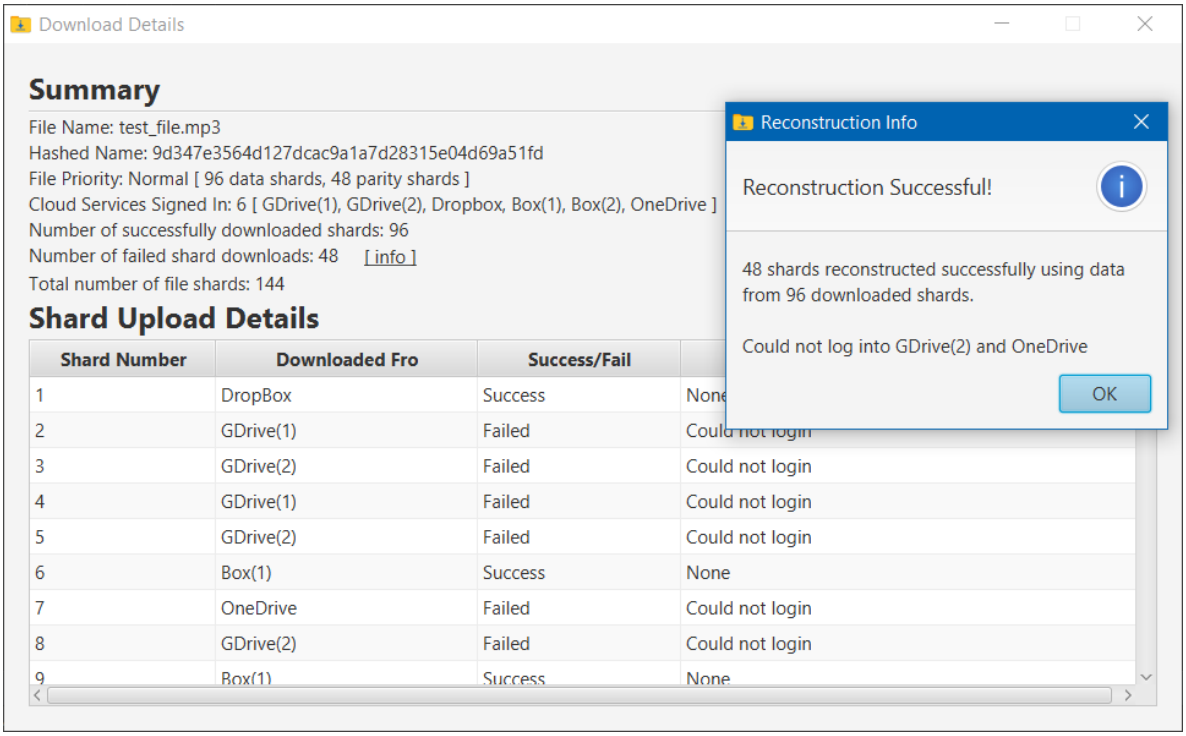


Figure 12 – successful file reconstruction during download for 48 corrupted shards with Normal option

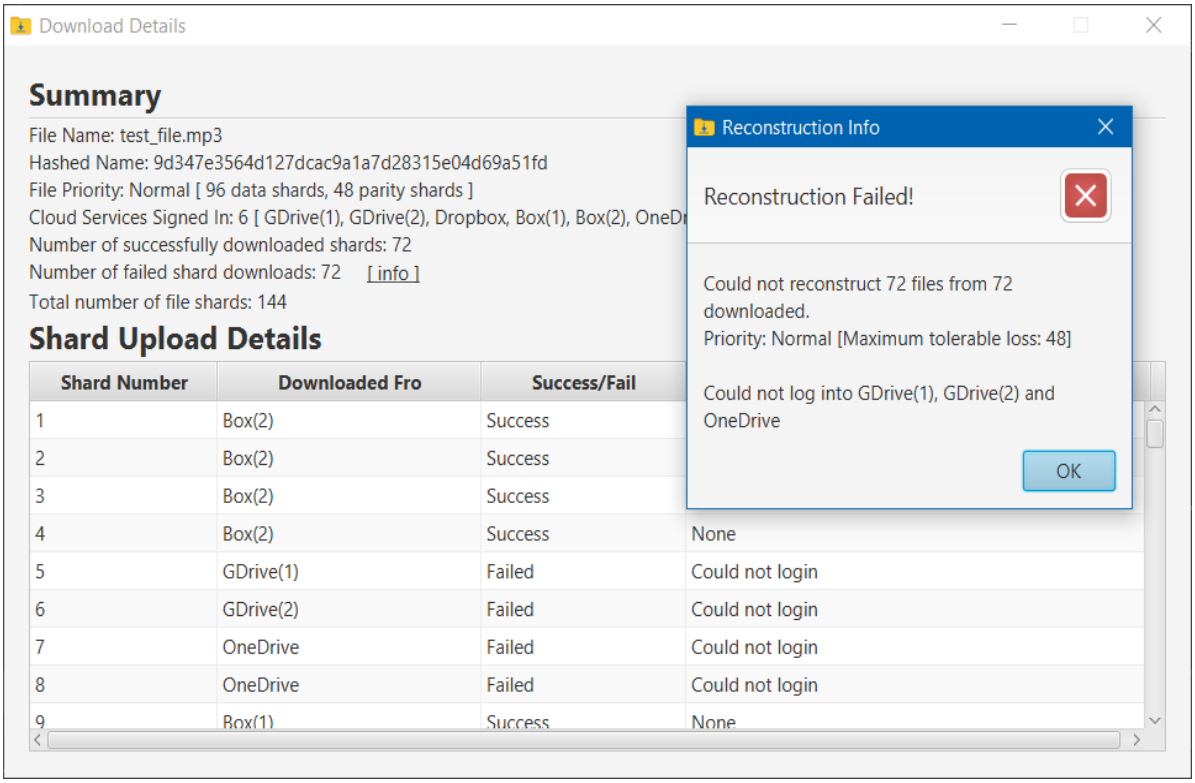


Figure 13 - Error! No text of specified style in document. **– failed file reconstruction during download for more than 48 corrupted shards with Normal option**

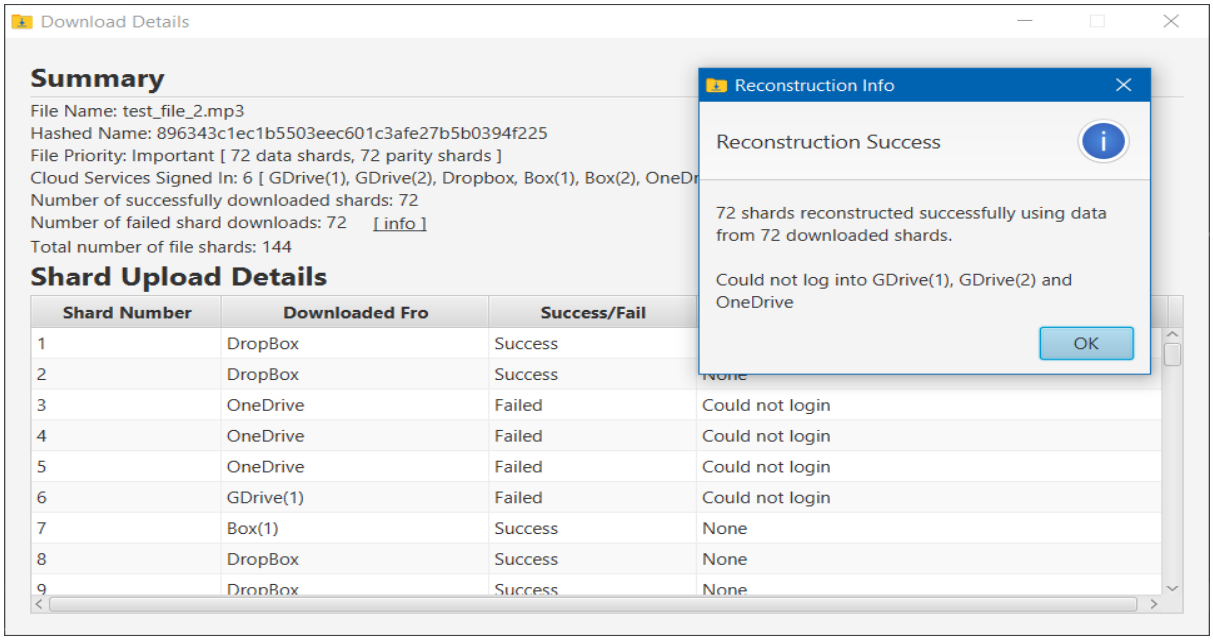


Figure 14 – successful file reconstruction during download for 72 corrupted shards with Important option

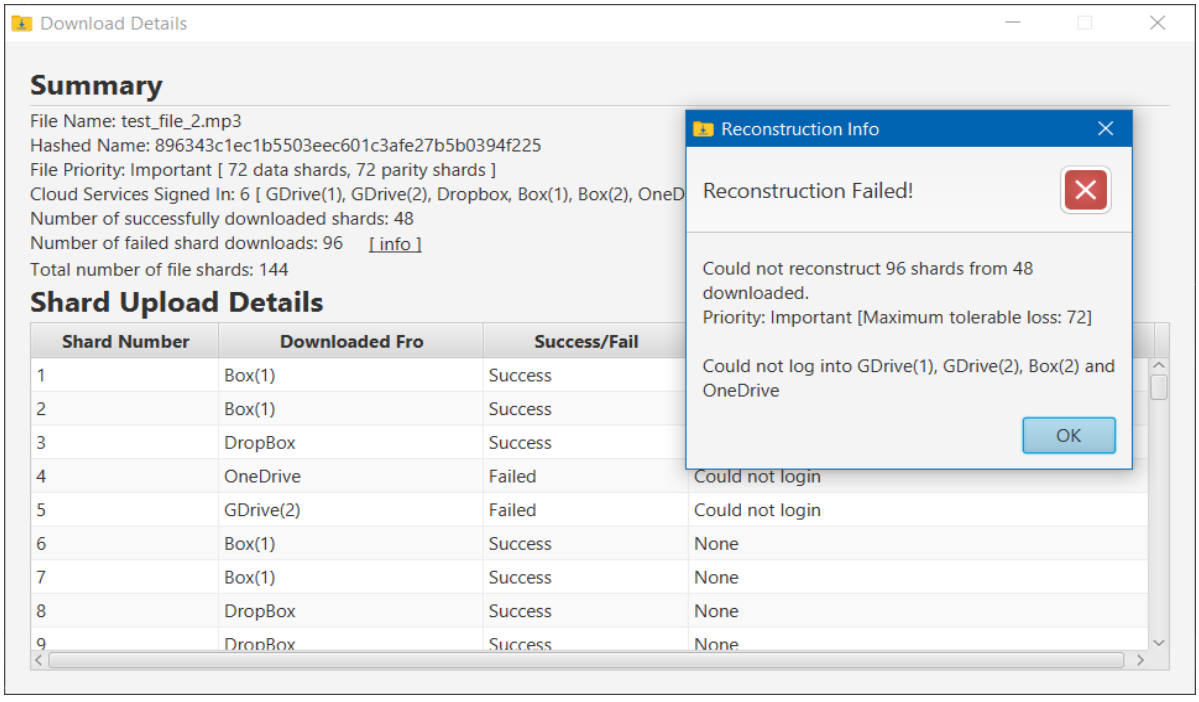


Figure 15 – failed file reconstruction during download for more than 72 corrupted shards with Important option

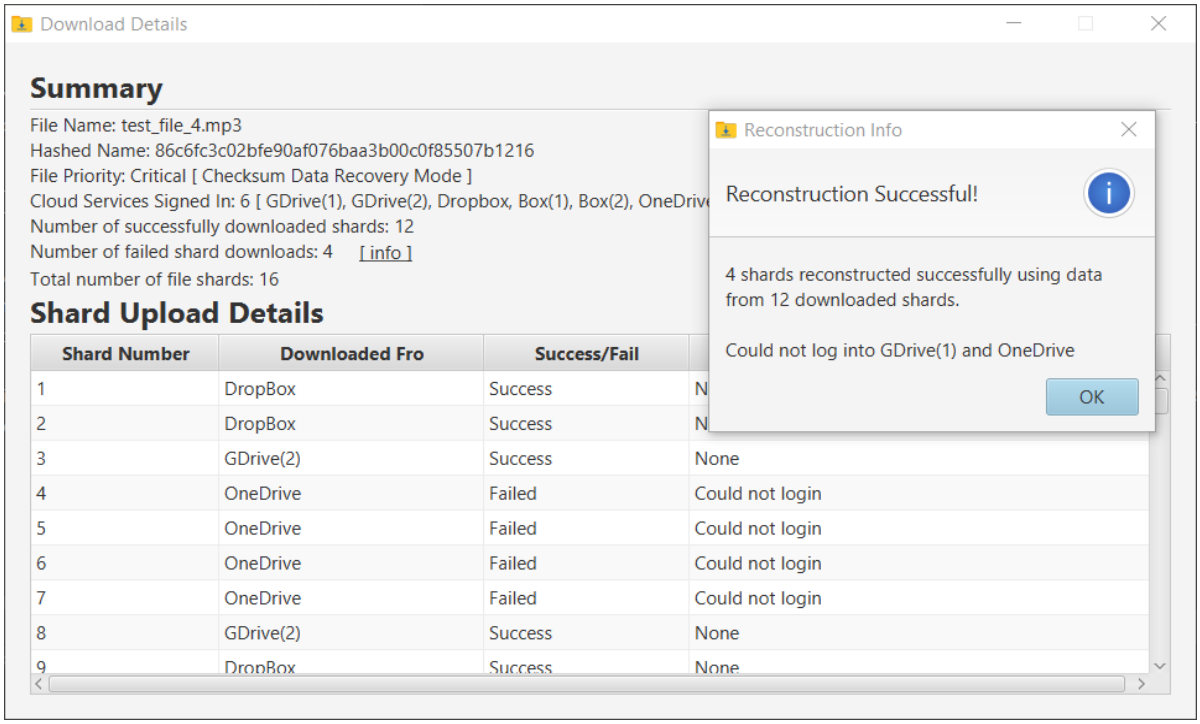


Figure 16 – successful file reconstruction during download for 4 corrupted shards with Critical option

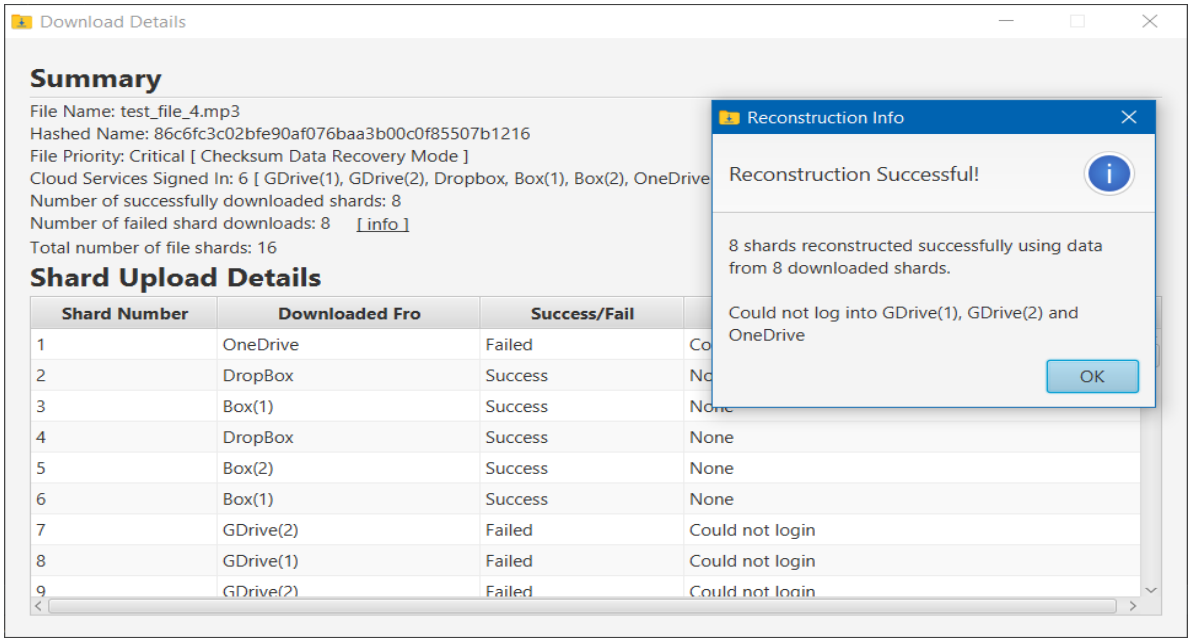


Figure 17 – successful file reconstruction during download for 8 corrupted shards with Critical option

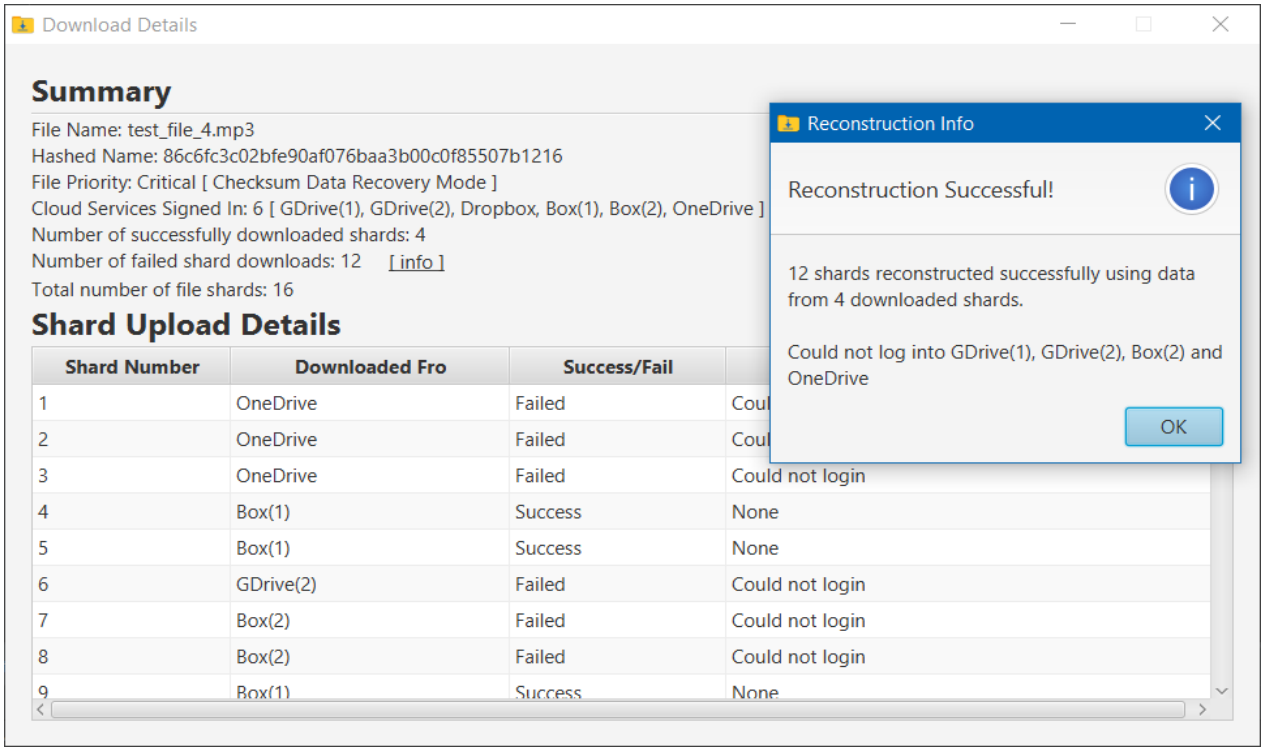


Figure 18 – successful file reconstruction during download for 12 corrupted shards with Critical option

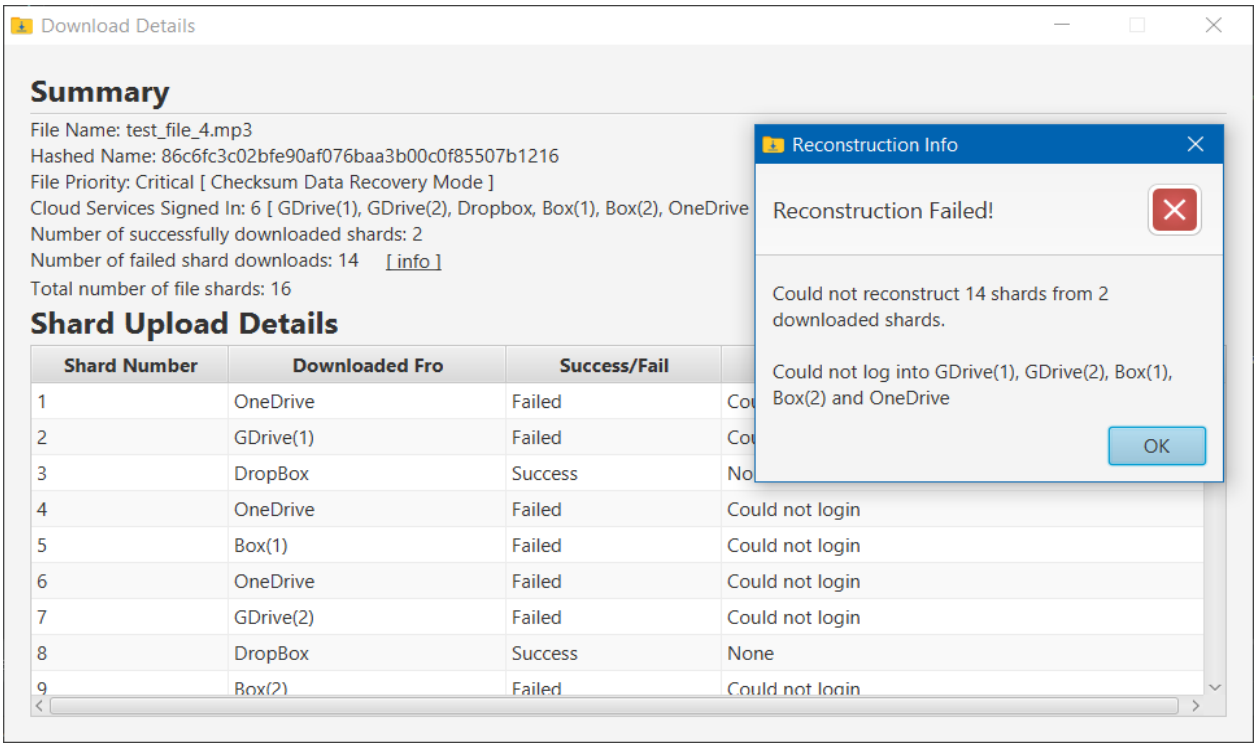
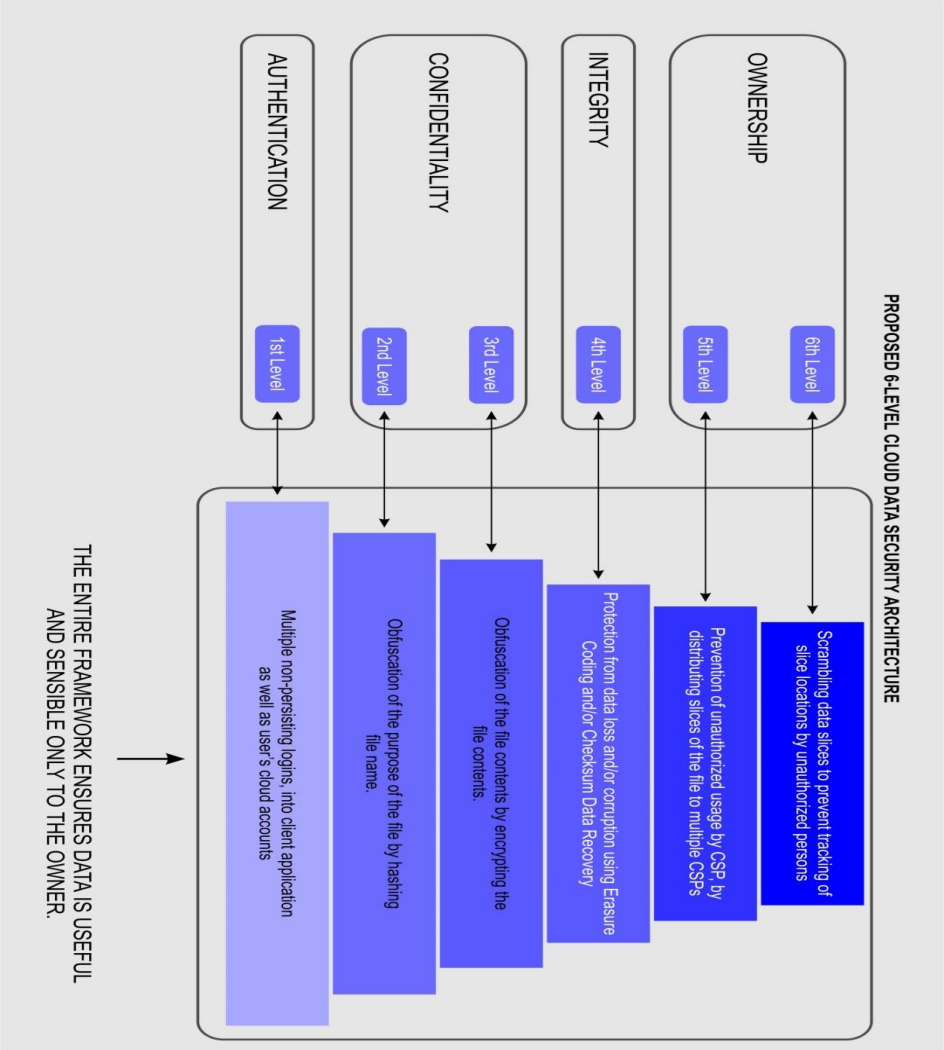


Figure 19 – failed file reconstruction during download for more than 12 corrupted shards with Critical option

APPENDIX 2



Proposed Six-level Cloud Data Distribution Intermediary (CDDI) Framework (Twum et. al. 2019b)

APPENDIX 3

Comparison of the CDDI framework with existing architectures

	Direct Model	Indirect (CASB) Model	Indirect (CDDI) Model
Architecture	The subscriber places a file into the CSP's interface for onward processing and uploads to the Cloud.	The CASB monitors all data transfers within the organization as well as the transfer of files out of (and into) the organization. All transfers are therefore filtered by the CASB. (Rubens, 2017)	The CDDI handles interactions with the CSPs on behalf of the subscriber. The subscriber is required to have a minimum of 6 Cloud Storage accounts. The CDDI performs data obfuscation on behalf of the subscriber.
Data Privacy	The CSP is responsible for ensuring the privacy of the subscriber's data. The method used to encrypt the file is known to the CSP, as well as the key or manner in which the key is generated. As such, if the CSP so desires, they may decrypt the file for their personal purposes (TipTopSecurity, 2016).	The CASB prevents unauthorized access to the organization's confidential data by preventing the confidential data from ever being transferred to the cloud. The CASB uses machine learning to determine data transfers that infringe the organization's regulations, and then halts the transfer.	In the proposed model, the CDDI allows all transfers of data to the cloud, but first encrypts the data locally (outside the CSPs reach) then splits the data into a number of shards, and randomly distributes the shards to multiple CSPs. The number of shards received by each CSP is insufficient to reconstruct the

			original file. This way, the file remains confidential and useful to only the owner.
Unauthorized Data Use	The CSP has access to the entire data and how to decrypt it. As such the CSP can use the data for any purpose without notifying the owner or requiring the owner's permission (Chima, 2016).	The CSP has access to the organization's non-critical data (data not captured in the organization's privacy regulations). The CSP is able to decrypt this data to use as they please.	The CSP has access only to incomplete and encrypted portions of the data. Without the other portions of the data, it is very difficult to decrypt as a result of the encryption algorithm used. Thus the CSP is prevented from accessing the data for their own purposes.
Unauthorized Data Access	One set of credentials are required to gain access to the data. Anyone with this single set of credentials can access the entire data.	The CASB serves as a proxy that also filters the traffic moving into and out of the organization. Thus the only credentials needed to access the data on the cloud, is the login credentials for the cloud account. Any individual with the login credentials, therefore, has access to the organization's data.	The CDDI is designed to demand login credentials of the system (SMF). Further, to access the data stored on the cloud, each cloud account must be signed into individually. Thus requiring multiple authentications before access and usage.
Data Ownership	Unless the client applies encryption before sending the data to the Cloud Service Provider, the provider can claim full ownership of the data as they have full access and control over it (FileCloud, 2016).	The CASB may encrypt the data before forwarding to the Cloud Service Provider, to ensure that the data is safe on the cloud.	The CDDI encrypts the data at the SMF client side and splits the data before distribution to the CSPs. Since the data is sent to multiple CSPs, no individual CSP can claim ownership of the complete data, except the data owner.
Data Integrity	The CSP is responsible for ensuring the integrity of the data. Most CSPs provide version control services which allow the subscriber to revert to a previous version of the file if the current version is damaged or otherwise modified. However, a malicious insider within the CSP may delete all traces of the file, making it irrecoverable (TipTopSecurity, 2016).	The CASB turn over responsibility of ensuring data integrity to the CSP upon the upload of the data. As such the subscriber has access to previous versions of the file but also suffers in the event of a malicious insider attack.	The CDDI uses Reed-Solomon Coding as well as the proposed Checksum Error Detection and Correction Program to verify the integrity of data and perform error corrections in the event that portions of the data get corrupted. The subscriber however does not get access to previous versions of the file but is protected from malicious insider attacks that happen in one or more CSPs.
Data Availability	The data is available anytime the CSP is in operation. However, in the event of a DoS/DDoS attack on the CSP, the subscriber has no access to the data.	The CASB relies on the CSP to ensure availability of the subscriber's data. (Rubens, 2017)	In the event of DoS/DDoS attack on any of the CSPs, the remaining data that is stored on the other CSPs can be used to reconstruct the original file. Hence the subscriber is insured and assured of data availability even when some of the CSPs are offline.