

# Chapter 1

## Introduction

The applications of network storage have developed for many years, such as Amazon, Google, which provide users store or share their data in the storage cloud. Users can upload their text file, photo, or music to one of the network storage systems, and the files will be stored in the network storage servers. Afterward, users can fetch the file according to the URL for retrieving copies or sharing copies to others.

These years, a new kind storage services are rising. These network storage service providers build their systems between the users and the network storage servers which mentioned above. Users can sign up their personal information in each web of network storage service system, and usually, each identity will be allocated averaged 5GB free network storage space. In fact, these new storage service systems might not have their own storage devices. The service builder rent a large amount of network storage space from network storage space providers, like Amazon and Google, and then develop services on them. For each user who signed up before can use not only network storage service, but also more extended services like file synchronization, web access and so on. Current public network storage service systems are like Dropbox [11], LiveMesh [12] and Syncplicity [13].

Network storage service systems usually provide some security insurances, for instance

1. Encrypt file while transferring
2. Encrypt file in the storage cloud

When user login the webpage and upload file, the system uses TLS/SSL mechanism to make sure file's privacy during transferring. After file is completely transferred, the services will encrypt the files by using user's personal information and store encrypted files in the storage servers. Basically, services providers offer users elementary file security guarantees.

However, once the network storage service system had been compromised, we can assume that attacker can obtain all users' personal information, including the information for encrypting uploaded files. Sometimes, users cannot retrieve from the cloud, because either the network disconnection, the failure of storage servers or service system is malfunctioned. One of reasons above will cause users cannot retrieve their critical data in time, which might result in property loss. Therefore, not only file privacy, but also file robustness we should concern when we want to improve file security stored on the Internet.

In current public environment, all network storage service system is independent, and there is no trusted third party to communicate and negotiate among them. It is almost impossible to build a system for being a trusted third party due to legal issues and commercial secret, so we did not apply centralized network storage architecture.

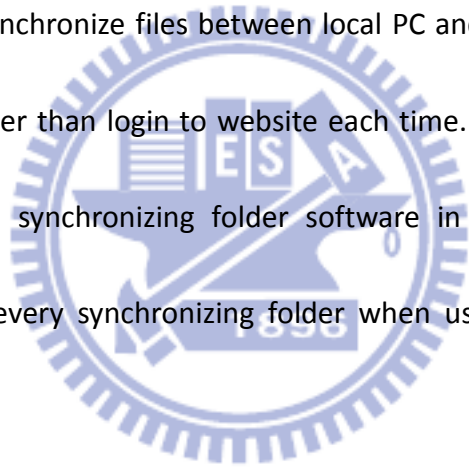
Traditionally, encryption is categorized into two part, asymmetric encryption and

symmetric encryption. The major difference between two categories is the usage of encryption key. In asymmetric category, the encipher and the decipher owns different key. The encipher encrypts file with public key and transmits ciphertext to decipher, and then decipher decrypt ciphertext with secret key to recover plaintext. RSA, ELGAMAL are the current famous asymmetric encryption. On the other hand, in symmetric encryption category, the encipher and the decipher share the same encryption key, the well-known representation, DES, 3DES, AES. Speaking of file security, the asymmetric encryption has an advantage over the symmetric encryption. On the contrary, the symmetric encryption has better computational performance.

Many researches have devoted in improvement of file robustness for decades, such as RAID system. The simplest method to protect file privacy is replication. Uploaded file can easily be duplicated into several copies and stored in different storage nodes. The advantage is low computational cost, but the disadvantage is the high cost of storage space. Another method to protect file robustness is erasure codes, such as EVENODD code [2], RDP code [3], B code [4], STAR code [5], and so on. To improve file robustness, file should be encoded by specified encoder and then generate codeword. Codeword consists of  $n$  symbols, the file owner can retrieve arbitrary  $k$  symbols to recover original file, where  $n > k$ . For some level, erasure codes can provide file privacy, because attacker can recover file only retrieve at least  $k$  symbol files successfully. However, it's barely possible to ensure that attacker only

compromise less k storage systems.

Currently, more and more network storage service systems, like Dropbox [11], LiveMesh [12], Synciplicity [13], AsusWebStorage [14], provide software to download from website, and let users install it to generate synchronizing folder in local device, such as PC or mobile phone. By using synchronizing folder mechanism, users can easily synchronize file with the cloud. Users can add file into each synchronizing folder and each installed system will automatically upload files to each cloud. Therefore, with the help of synchronizing folder mechanism, users can synchronize files between local PC and each network storage servers with simple method rather than login to website each time. But, users who have bad habit might be installed each synchronizing folder software in any location in PC, and it is inconvenient to search every synchronizing folder when users want to use synchronizing folders.



In summary, we try to implement a system that provide users an integrating platform so that users can synchronize files between local PC and each network storage servers and meanwhile protect the file privacy and robustness in clouds.

In these following chapters, we will first discuss the overview and background of our system in chapter 2, and then, the implementation detail will be illustrated in chapter 3. The rest chapters are performance analysis, future work and conclusion.

## Chapter2

### System overview

Our system so far is only suitable for local PC, because our system adopts synchronizing folder to make local file synchronized with the each storage cloud. It does not make sense for users to install synchronizing folder software in public PC, such as PCs in library, airport or any public environment. Therefore, our system is not designed for temporary usage in public environment.

At the beginning, user's PC has multiple synchronizing folders, which can automatically synchronize files with each storage cloud, and user can use them individually. For example, user Anna installed Dropbox at desktop and LiveMesh at "D:\\". Once if Anna wants to upload A.txt to Dropbox and B.mp3 to LiveMesh, she has to add each file into corresponding synchronizing folders separately, although it is the most convenient method to upload file by now.

In order to provide users an integrating platform instead of using synchronizing folders individually, we have strong motivation to generate a unique folder, which is called "system folder" in the rest of the article, to synchronize files with each synchronizing folders. In other words, synchronizing folders synchronize files inside with their own storage servers, and our

system folder synchronize file inside with all user-assigned synchronizing folders. At first glance, our system generates a system folder and continuously detects and synchronizes file with user-assigned synchronizing folders. Our system does not interfere in any transmission mechanism of each synchronizing folder.

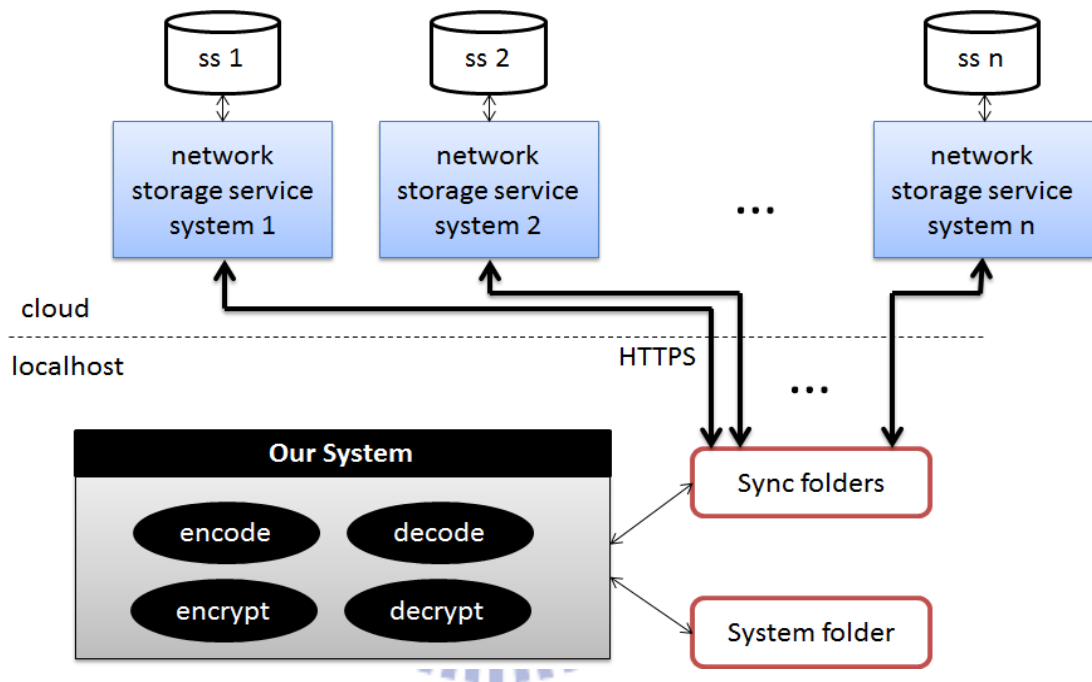


Figure 1

Our system architecture, as shown in Figure 1, includes the cloud and local PC. In the cloud, there are n independent network storage service systems and their corresponding storage servers (SS), which no matter whether belongs to network storage service system itself. In the local host, there are three roles. First, n synchronizing folders (Sync folders),

which user could download software from the website of each network storage service system. Second, one system folder (System folder), generated by our system, is an ordinary folder in PC. Third, it's our system, implemented by JAVA, which execute all computational functions and continuously detect sync folders and system folder.

Our system program is in charge of all file computation, including file encryption, file decryption, file encoding and file decoding. When users want to upload protected file to the each storage cloud through our system, the input is the file in system folder. After being encrypted and encoded, our system write each ciphertext codeword symbol files into n sync folders. On the contrary, our system reads symbol files in each sync folders and execute decoding and decryption to recover original file and write into system folder.

The sync folders and the components in the clouds are the current public network resource, and users can easily fetch on the Internet.

Our system is implemented by JAVA, so localhost should install JRE to execute our system. So far, our system works fine with Windows 7 32bits and Windows 7 64bits, and we have enough reasons to believe our system can also work fine in other operating systems. The only issue we concern is that synchronizing folder software does not support any kind of OSs.

As shown in Figure 2, our system program, ccis.java, is packaged with two text file, a\_181\_603.properties and primitive\_element.txt. Users can double click the ccis.java to start

our system. The two text files are critical, and our system program will fail without these two text files under the same directory. Without any installation procedure, our system program and the two text files can be changed to any location in local PC.

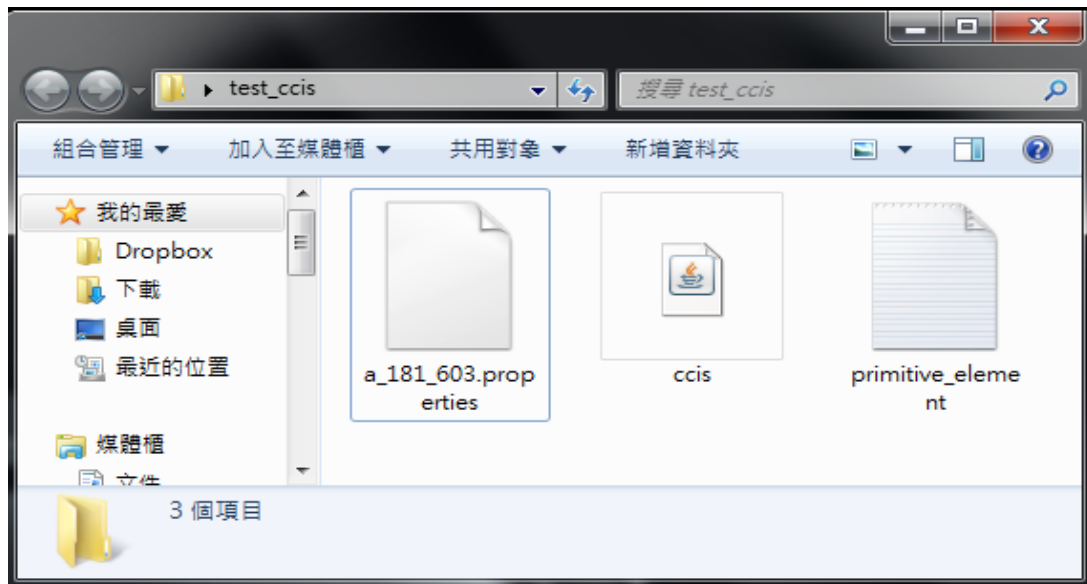


Figure 2

After user starts our system program, our system environmental setup UI pops up every time. As shown in Figure 3, our system program requires four parameters, which are system folder path, sync folders paths, the fault tolerance degree number and an arbitrary string. The purpose of user-assigned system path is that our system let users to decide where system folder is. When system program is executing, our system program generate a new folder, which is named “ccis\_repository”, to be our system folder at user-assigned path. Users can choose an arbitrary valid path in PC and click “Edit” which is next to the first text filed.



The second text field is a list of sync folders paths, and users can insert or delete any sync folders paths. Our system can detect whether the path users assigned is a valid folder, but, our system cannot detect whether it is a sync folders. User has to assign at least two sync folders to our system program. The third text field let users insert the fault tolerance degree users want out system provides. This third text field only adopts number string, furthermore, this number should be less than the count of sync folder paths and larger than zero. The final text field user should insert an arbitrary string, and our system will adopt it to generate user's public key and secret key. In other words, if user wants to recover ciphertext files, he should insert the same string he inserted when encrypted files. After finishing all parameters, click "Start" to launch our system program. At first usage of our system, all text fields are blank.

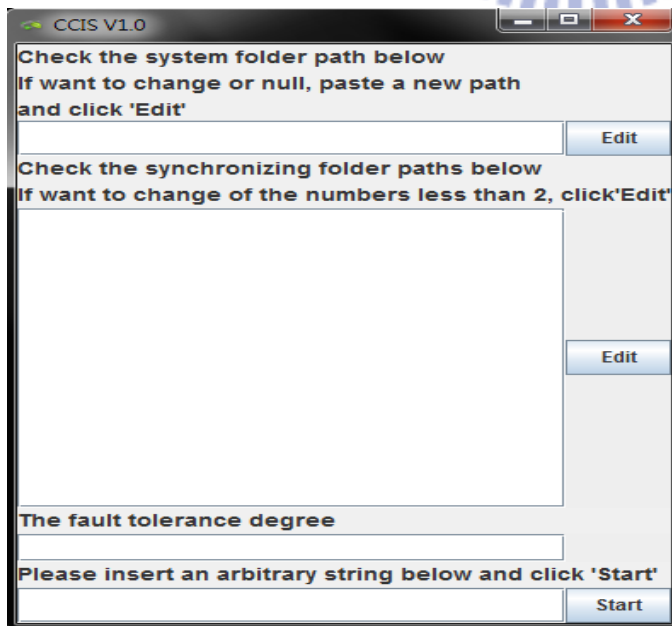
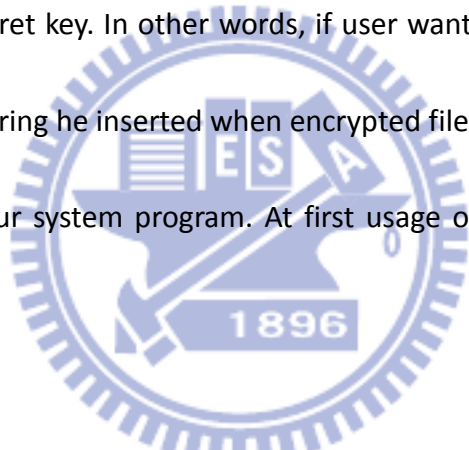


Figure 3

Our system will keep pre-setup information, so at subsequent usage, user only needs to insert the arbitrary if he does not want to change the other information. As shown in Figure 4, our system stores previous information. Our system stores the fault tolerance degree in FT\_num.txt, system folder path in repository\_dir.txt and sync folder paths in sync\_dir.txt. If these three text files lost, which only cause related information text field blank in this UI. Unlike a\_181\_603.properties and primitive\_element.txt, the loss of these three file will not crash our system.

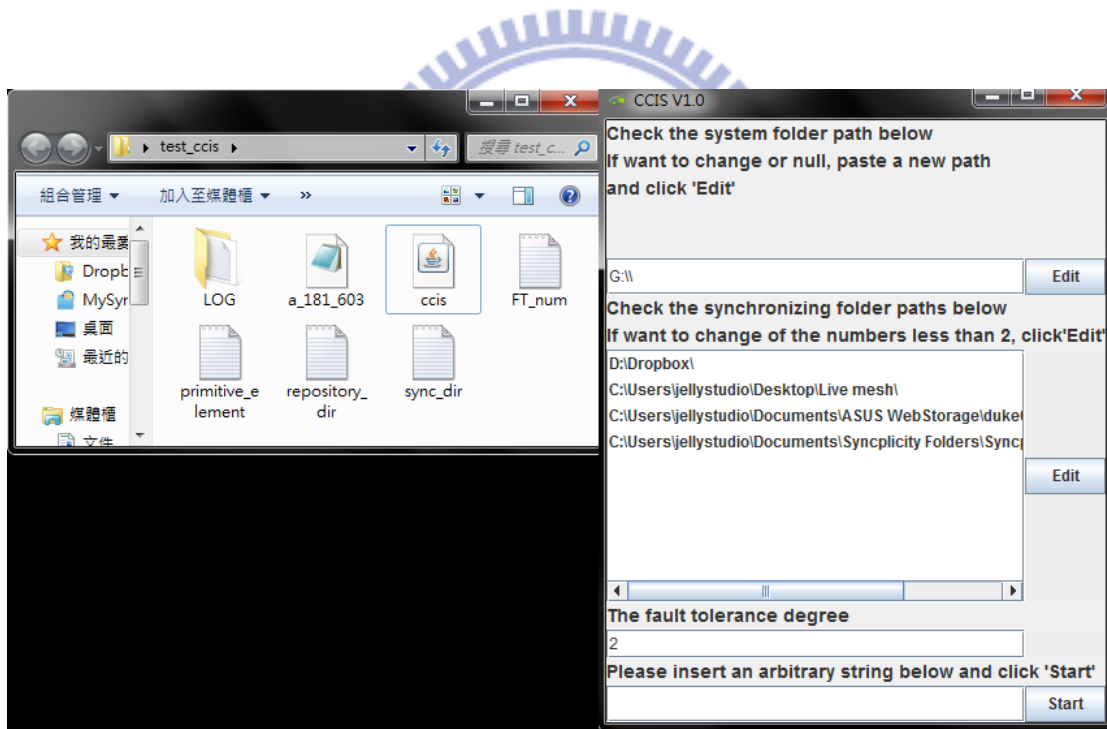


Figure 4

In the next we will introduce our system functionalities. In this chapter, we only illustrate our system's functionalities and the detail implementation techniques will be

discussed in next chapter. Our system provides

1. Detect whether new files added into system folder.
2. Detect whether file in system folder is modified.
3. Detect whether any loss of codeword symbols file in each sync folders.
4. Detect whether any file is deleted in system folder.
5. Encrypt and encode files to protect file privacy and robustness.
6. Decode and decrypt file to recover protected files.

Because our system program continuously scans system folder and n sync folders with infinite loop and detects whether 1, 2 or 3 fits. If yes, our system program will figure out which files should be executed file encryption and file encoding. If 4 fits, our system interprets this situation that user wants to remove file from the cloud. Therefore, our system program will delete all related codeword symbol files in n sync folders, and with the help of synchronizing folder software, the files in the cloud will also be removed. In addition, our system program will decode and decrypt all files in sync folders and write original files into system folder.

Figure 5 is our system execution flow. In the beginning, our system decodes and decrypts all codeword symbol files in sync folders. If recovery fails, our system will skip all subsequent procedure and execute next files. Our system only executes decoding and decryption once during whole system executing time, therefore, our system does not

support real-time file synchronization.

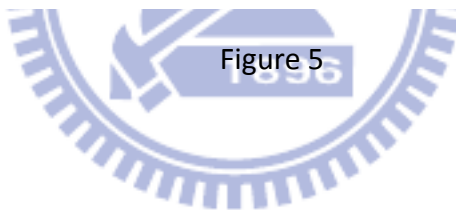
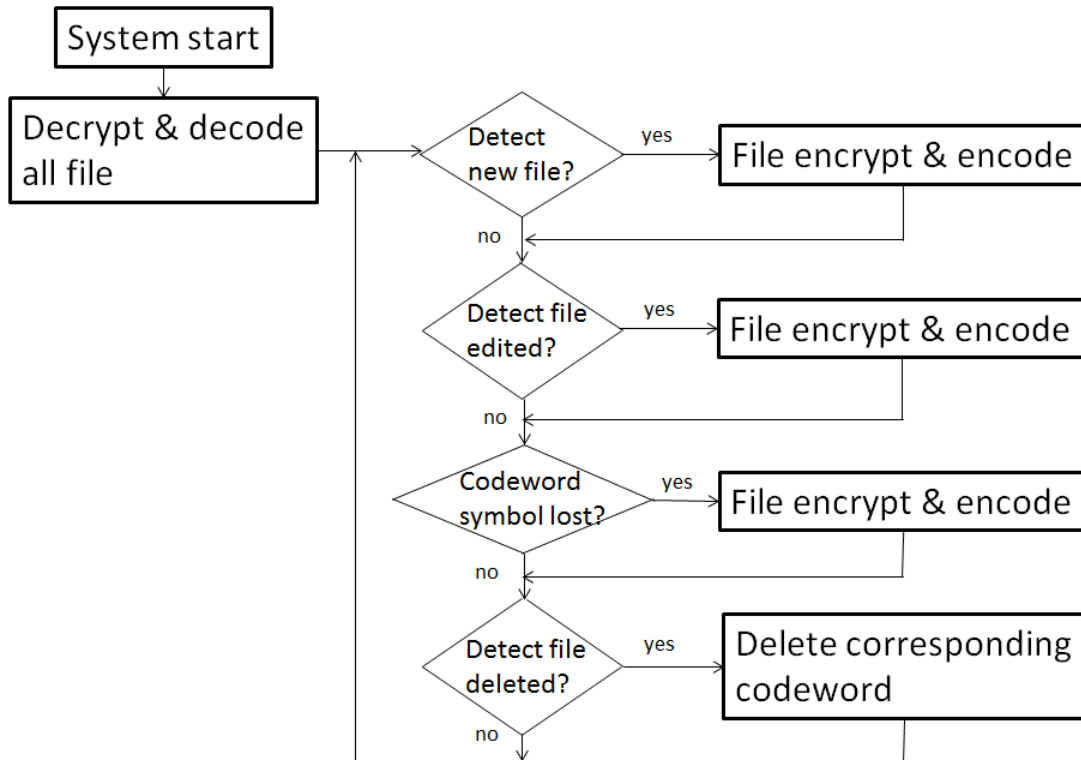


Figure 5

After decoding and decryption complete, our system program pops up an execution information UI and starts next procedure. Subsequently, as shown in Figure 5, our system continuously detects four events. The execution information UI is showed in Figure 6. User will be informed the system folder path and the sync folder paths, and click the two “OPEN” buttons will open the related folders. All information which is generated during system execution will be printed in system information text field.

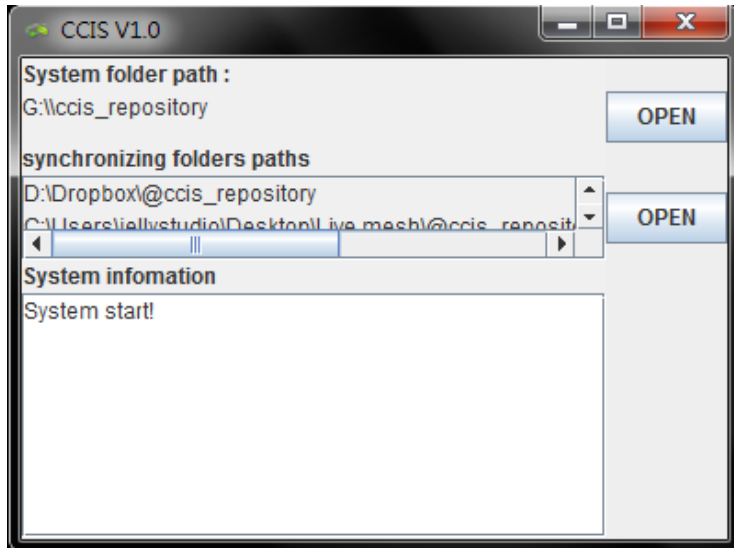


Figure 6

When user adds files or folders into system folder “ccis\_repository”, our system program will detect which files in system folder is newly added and start to execute computation. The information will be printed in the execution information UI, as shown in Figure 7. For example, if user adds a new file DSCF0365.JPG into system folder, some information will be shown on the UI, including upload time, file name, progress, and total time cost.

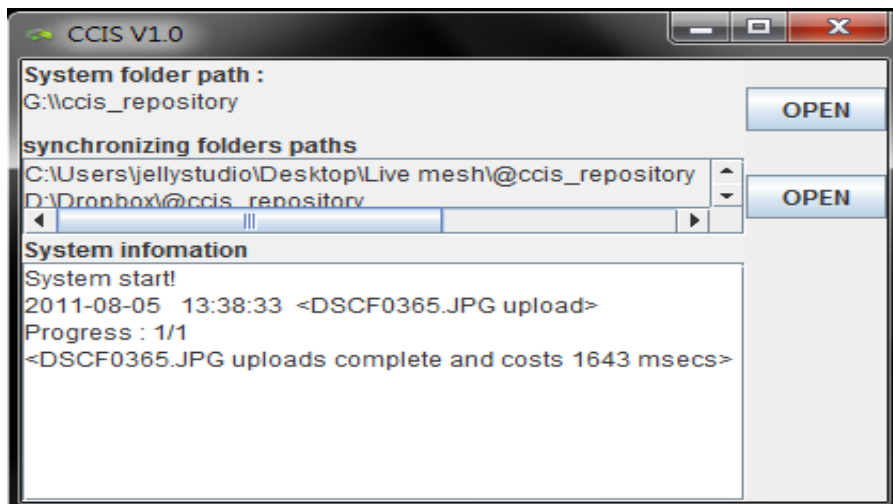


Figure 7

If user uses synchronizing folders, they can modify any file and store directly. Then, the synchronizing folder software will automatically upload modified file to the cloud. Therefore, when user want to retrieve file back next time, he or she can obtain the last modified files. To keep this functionality, our system should let users to modify their file in system folder so that users can retrieve last modified files when our system executes decoding and decryption.

When our system program is executing, user opens his file in system folder and modifies it. Our system program will detect file changed in system folder, and do file encryption and encoding that file again to let n sync folders and relative cloud storage store the latest file codeword symbol files. The example shown in Figure 8

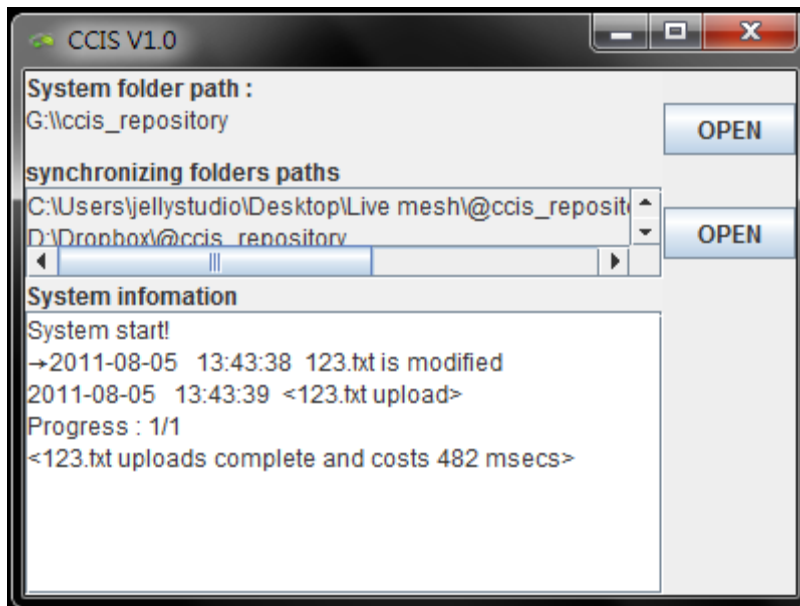


Figure 8

As mentioned before, one of our system's major functionalities is synchronizing file between system folder and n sync folders. The only difference is that file in system folder is original file, and files in n sync folders are codeword symbol files. In other words, our system program can detect whether there are valid amount of codeword symbol files in sync folders by measuring the original file in system folder. Once our system program detects codeword symbol files lost, our system program will encrypt and encode the related file again to generate complete codeword symbol files. Figure 9 shows the example.

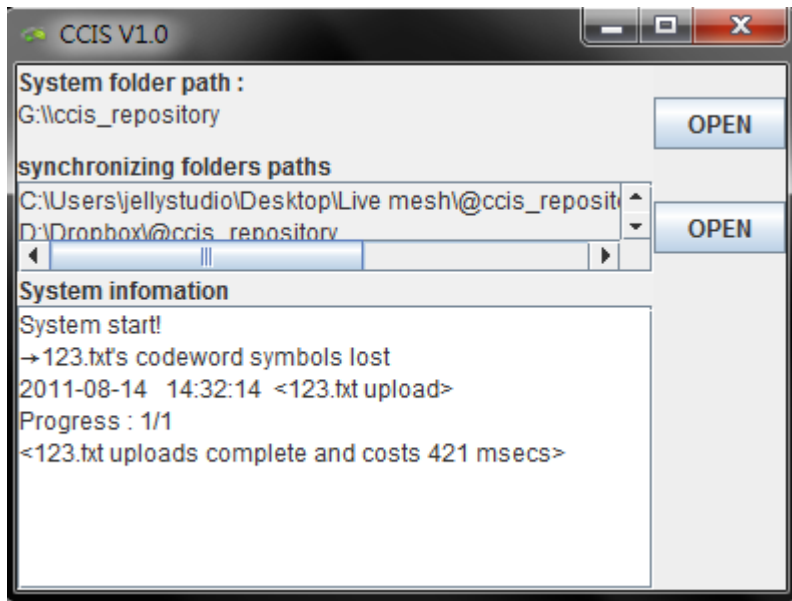


Figure 9

If user wants to remove file in the cloud, because of our system architecture, he or she should delete all related codeword symbol files in sync folders, and that is very inconvenient. Therefore, our system establishes one rule that once if file in system folder is deleted while our system program is executing, our system will determine that user want to remove file in sync folders and also in the cloud. As shown in Figure 10,



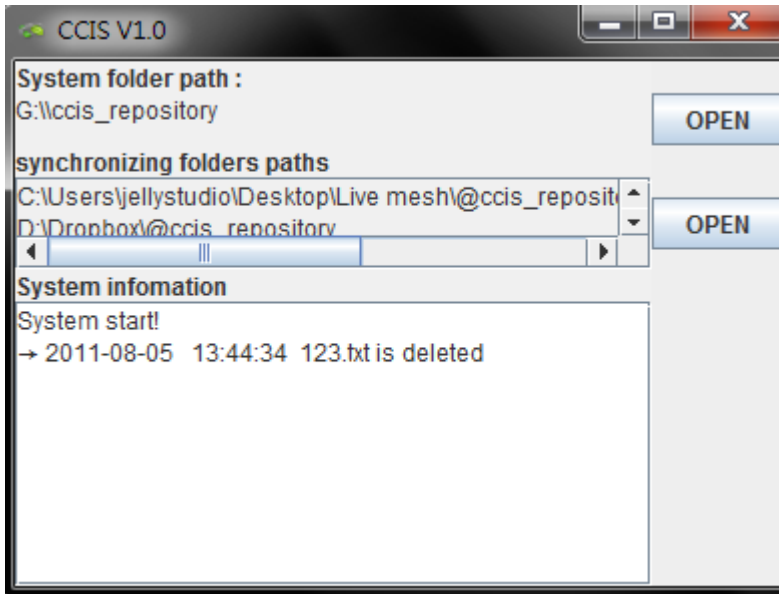


Figure 10

During system program executing, user can minimize the information UI to let system program execute background, however, once if user click the close icon, our system will terminate.

As shown in Figure 11, our system provides file privacy guarantee, the ABC.txt is original file stored in system folder. The two files below are contents of file codeword symbol file and encryption key codeword symbol file. The contents of codeword symbol are unreadable; hence, our system guarantees that even if the attacks compromised all network storage service systems, they cannot obtain any information from uploaded files.

On the other hand, the file robustness ability of our system will be discussed later.



Figure 11

After introducing our system functionalities, we will discuss the preliminary knowledge of encryption, encoding, decoding and decryption schemas of our system. The technique of protecting files is based on [1], which was published on IEEE TRANSCANTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS in 2010. In the following paragraphs, we will introduce how [1] protect file and what parts we modified to be more practically. [1] improves files robustness and privacy in distributed networked storage environment with cryptographic pairing technique and distributed erasure code. Unlike other distributed networked storage researches before, this thesis protect not only file robustness but also file privacy, which result in that even if attack compromises all storage system, he still cannot obtain original file information.

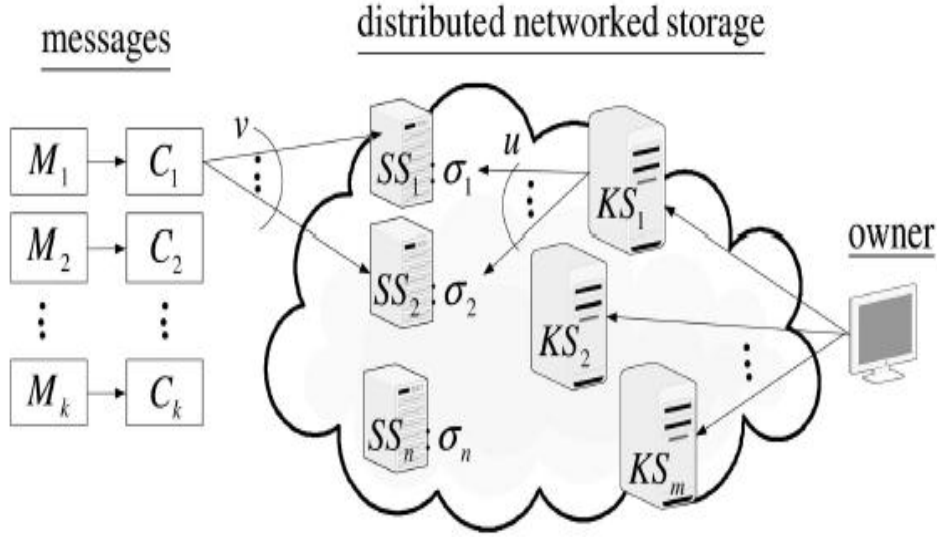


Figure 12

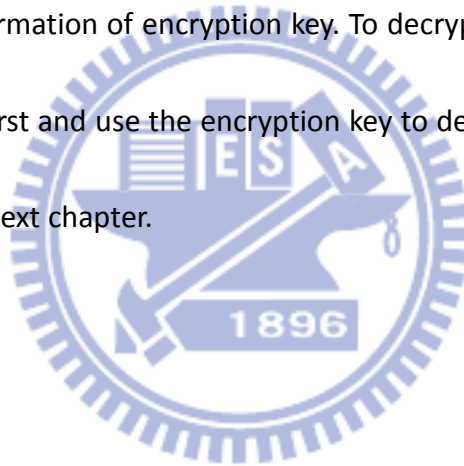
Figure 12 shows the [1] model. There are  $k$  plaintexts  $M_i$ , which are encrypted by  $C_i = (\alpha_i, \beta, \gamma_i) = (g^{r_i}, h_{ID}, M_i \tilde{e}(g^x, h_{ID}^{r_i}))$   $0 \leq i < k$ .  $g$  is generator of  $\mathbb{G}_1$ , which length is  $p$ .  $r_i$  is randomly chosen from  $\mathbb{Z}_p$ . All ciphertexts  $C_i$  have the same  $h_{ID}$ , where  $h_{ID} = H(M_1 || M_2 || \dots || M_k)$  and  $H : \{0,1\}^* \rightarrow \mathbb{G}_1$ . A bilinear mapping function  $\tilde{e}$ , which map elements from  $\mathbb{G}_1$  to  $\mathbb{G}_2$ .  $x$  is user's private key, and  $g^x$  is user's public key. The user shares his secret key shares among  $m$  key servers ( $KS_i$ ) with  $(t,m)$ -threshold secret sharing scheme. The user uploads  $v$  copies of each ciphertexts  $C_i$  to storage servers ( $SS_j$ ), which are randomly chosen. Afterward, every storage servers will encode their own files which received from user. For each storage servers  $SS_j$ , they encode file to generate codeword symbols  $\sigma_j = (A_j, hid, B_j, (\alpha_{1,j}, \dots, \alpha_{k,j}))$   $1 \leq j \leq n$ , where  $A_j = \prod_{0 \leq i < n-1} (g^{r_i})^{\alpha_{i,j}}$  and  $B_j = \prod_{0 \leq i < n-1} (M_i \tilde{e}(g^x, h_{ID}^{r_i}))^{\alpha_{i,j}}$ .  $\alpha_{i,j}$  is randomly chosen from  $\mathbb{Z}_p$  and  $\alpha_{i,j}$  is not

equal to zero. When the user wants to retrieve  $k$  messages, he should send  $h_{ID}$  to all key servers and each  $KS_i$  randomly queries  $u$  storage servers with  $h_{ID}$  and obtains at most  $u$  symbols from storage servers. Then,  $KS_i$  decrypts every codeword symbols by  $\tilde{\zeta}_{i,j} = (A_j, hid, hid^{sk_i}, B_j, (\alpha_{1,j}, \dots, \alpha_{k,j}))$  and sends back to the user. The user randomly chooses  $\tilde{\zeta}_{i,j}$  from all received data to compute Lagrange interpolation over exponent to generate  $h_{ID}^x$ . The user encrypts the data by computing  $w_j = \frac{B_j}{\tilde{e}(A_j, hid^x)}$ ,  $1 \leq j \leq k$ . Finally, the user decodes the file by retrieving  $\alpha_{i,j}$  in codeword symbol and forming a submatrix of generator matrix. If the submatrix is invertible, the user recovers each message  $M_j = \prod_{0 \leq i < n-1} (w_i)^{\beta_{i,j}}$ , where  $\beta_{i,j}$  entries of the inverse matrix are.

In [1], the storage servers execute encoding procedure and store codeword symbols. However, in current practical environment, it is almost impossible to ask each public network storage service system to compute encoding protocol, hence, we should let storage servers to only store file in our architecture and left all computational tasks to local PC.

Another issue is the maintenance of key servers. Once if the amount of key servers failure is beyond threshold, the whole system will crash down and stop serving anymore. Because of the lack of human resource to maintain the key servers, we decide to remove the key server mechanism so that our system only has two roles, the system program in local PC and the public network storage service systems, which should be maintained by experts. Therefore, our system should work as long as the network storage service systems exist.

The other problem is the computation performance. Usually, user's file size today is larger than 1KB, even 1MB. Because the encryption protocol in [1] adopts asymmetric encryption, the computational cost will very high. For this reason, we extend the original encryption protocol to develop a new hybrid one. The hybrid encryption combines the symmetric encryption and the asymmetric encryption, and owns benefits from two kinds of encryption schemes. In our system architecture, our system program generate an encryption key to encrypt file with symmetric encryption first, and then adopts asymmetric encryption method in [1] to encrypt the information of encryption key. To decrypt files, our system program will recover encryption key first and use the encryption key to decrypt file. The detail techniques will be illustrated in the next chapter.



## Chapter 3

### System implementation

#### 3.1 Introduction

Our system program is built by JAVA, and our system consists of these source code

1. ccis.java
2. main\_UI.java
3. Edit\_syn\_dir\_UI.java
4. Dec\_download\_thread.java
5. download\_warn.java
6. AES.java
7. Base64.java
8. group\_gau\_eli.java
9. Gauss.java
10. Dec\_upload\_thread.java
11. upload\_info.java
12. File\_delete\_thread.java
13. CloseHandler.java



To execute pairing based cryptographic computation in JAVA [6], our system import external libraries

1. jna-3.1.0.jar
2. jpbc-api-1.1.0.jar
3. jpbc-crypto-1.1.0.jar
4. jpbc-pbc-1.1.0.jar
5. jpbc-plaf-1.1.0.jar
6. bcprov-jdk16-140.jar

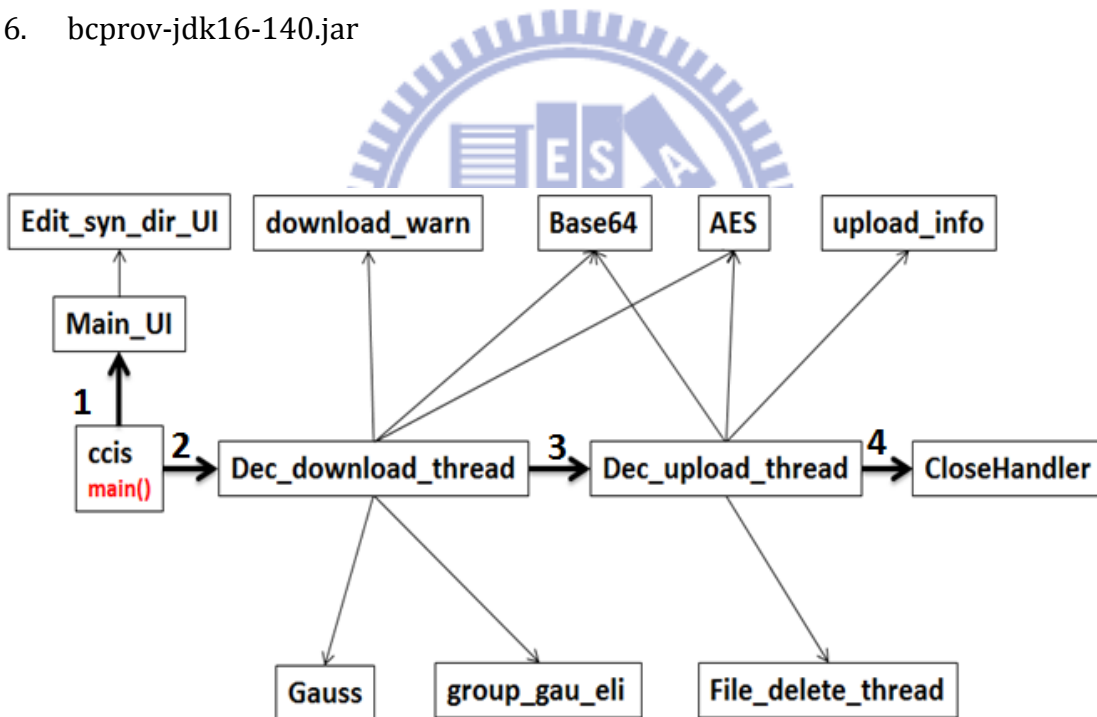


Figure 13

Our system classes flow is shown in Figure 13, and the main entry point is in class ccis.

With following the number sequence, in the beginning, our system program will execute

codes of Main\_UI to launch the environmental setup UI to let user to insert the information, and if necessarily, our system program will execute codes of class Edit\_syn\_dir\_UI to let user modify his synchronizing folder path. After setting all parameter of our system program, our system program will execute file decoding and decryption, whose source is of Dec\_downlaod\_thread class. During file decoding and decryption, our system will call methods in many different classes, such as download\_warn class, Base64 class, AES class, Gauss class and group\_gau\_eli class. If our system program cannot decode and decrypt codeword symbol files, our system program will new a UI object from download\_warn class to inform users. The other classes provide methods to help our system program to finish decoding and decrypting files, we will discuss in detail step by step in the following paragraphs. Consequently, our system program will jump into an infinite loop to continuously scan and detect, which are implemented of Dec\_upload\_thread class, in meanwhile our system will new a UI object form upload\_info class to notify users our system status. And during encryption and encoding procedure, our system program will call the methods from Base64 class, AES class and File\_delete\_thread class to help our system program to finish encrypting and encoding.

In the following paragraphs of this section, we will introduce every parts of our system program source codes according to our system execution sequence. In addition to the computational parts, we also introduce how our system scanning files between system folder



and sync folders.

### 3.2 Parameters setting

In order to make our system work successfully, our system require users to insert parameters and our system will execute by these information. When users click ccis.jar, our system program will soon be launched. In the beginning, our system generates a Main\_UI object to pop up a UI and let user insert parameters, as shown in Figure 3. We set event listeners on each buttons. The First button in the next to system path text field provides user to edit system folder path. Once if user clicks the first button, our system program receives the event messages and starts the following procedures. (1) Check whether this path is valid. (2) If this path is valid, our system program writes it in repository\_dir.txt, which in the same folder as our system program. Otherwise, the Main\_UI will show the warning messages to users and ask for insert the correct path and click first button again. Our system use JAVA method exist() of File class to check whether the target directory exists. If the return value is true, the target directory exists in local PC; otherwise, the directory does not exist. .The second button will close the current UI and generate a new UI from Edit\_syn\_dir\_UI class. The UI let user to insert a new synchronizing folder path or delete old ones. This Edit\_syn

\_dir UI provides the functionality to notify user whether these paths are valid and write the correct paths list in sync\_dir.txt, which is located in the same folder with system program. The final button will examine the whether number of sync folder paths,  $n$ , user inserted is larger than 1, and the fault tolerance degree user inserted in third text field is larger than zero. Besides,  $n$  minus fault tolerance degree is the word dimension,  $k$ . Our system program will save the fault tolerance degree number in FT\_num.txt, which is also at the system program folder. Finally, Main\_UI object write the arbitrary string into Authenticate.txt and close.

After reading non-NULL string from Authenticate.txt by main thread, our system will move on next procedure 2 to decode and decrypt all codeword symbol files.



### **3.3. Preparation**

After handling all parameters, our system program will call the method search\_data\_dec() to scan and generate a file list to execute file decoding and decryption. In search\_data\_dec(), the major task is recursively generate file list from sync folders.

To generate a file list from sync folders, our system program will go through each folders in the sync folders. Our system uses method listfiles() to read files in every folders of sync

folders, and use `isFile()` and `isDirectory()` to detect files and folders. Because our system has a specified naming mechanism to distinguish each files codeword symbols, hence, our system program can generate a files name list and a folder name list by parsing the name string and removing the tag our system program added in. Our system program reads all files and folders in the folders with the same related path in each n sync folders. For example, if there is a folder “apple” in sync folder A, our system program read all files and folders in “sync folder 1’s path\\apple\\” and “sync folder 2’s path\\apple\\” until “sync folder n’s path\\apple\\”. If one of the paths does not exist, our system will treat it as an erasure.

Subsequently, our system program parses the file content in the folder of the same related path in sync folders, and generates two lists of unique file names and folder names.

Our system handles files list first. To decode and decrypt files one by one, our system programs call the member function `decrypt_a_file()` in class `Dec_download_thread` again and again. The parameters of `decrypt_a_file()` require the executing file name, system folder path, the sync folder paths, the related path and the word dimension, k.

After executing all file decoding and decryption in the file list, our system will iteratively add the folder name in the list to current path and recursively call the function `search_data_dec()` with the new paths. Therefore, our system program will scan the child folders of the original ones.

### 3.4 Decoding and Decryption

The `Dec_download_thread` class consists of four member functions, which are `decrypt_a_file()`, `detSplitCount()`, `setFileKeyIndex()` and `read_sk()`. This class is only in charge of decoding and decrypting file, and the files list and executing sequence is determined by `ccis` class. As mentioned before, the `Dec_download_thread` class will call other member functions from `download_warn` class, `AES` class, `Base64` class, `Gauss` class and `group_gau_elic` class.

Before starting to discuss how we implement our system program to execute decoding and decryption, we have to introduce the information of groups our system adopts. To generate two groups with pairing relation, our system program imports external `jPBC` libraries [6]. Our system program also uses the default symmetric pairing parameters, which stored in `a_181_603.properties` and provided by `jPBC` developer. By loading the same parameters, our system program can always generate two groups  $\mathbb{G}_1$  and  $\mathbb{G}_2$  with symmetric pairing relation every time. The length of  $\mathbb{G}_1$  is 603 bits and 181 bits is the order length of both groups. According to the member function of `jPBC` library, our system program will map elements in  $\mathbb{G}_1$  to elements in  $\mathbb{G}_2$ . When execute decryption, in order to use the same primitive element in  $\mathbb{G}_1$ , which also used in encryption, our system also keeps the

primitive element information by storing in primitive\_element.txt. Both groups are generated on ECC; hence, our system provides higher level of security than RSA with 1024 bits, which fit the NIST security suggestion.

The primitive element in  $\mathbb{G}_1$  is  $g =$

```
{x=615734512873428440477488523826308597799520831636693137614193293379606827  
592117047629241685016477711187420535546292089408355875225815409547842815495  
49383080126812686938354177995588953, y=2060401751298941287883225831442437614  
704865289320923746776455740148877803089231620998617527984360811606712097453  
5375335832695868464306816023514802682052300350575909977166528430298774, infF  
lag=0}
```

After generating the files list, our system program will call the member function, decrypt\_a\_file() of Dec\_download\_thread class to decode and decrypt file one by one. Therefore, in the rest part of this section, we will introduce how our system program performs file decoding and decryption.

In the beginning, decrypt\_a\_file() of Dec\_download\_thread class receives seven parameters from the caller, search\_data\_dec() of ccis class, and handles the total path and the related path to generate four string arrays for subsequent usage.

In the next step, our system program will parse the file name received from caller function to make our system program to find the correct codeword symbol files. Our system

program use the member function `split()` of JAVA String class to split file name string into several parts by the dot sign “.”. For example, string 123.456.txt is the executing file name, and our system split the file name into three parts, which are 123, 456 and txt. To read the correct file codeword symbol files and encryption key codeword symbol files, our system should use the file name to generate file codeword symbols name and encryption key codeword symbol name.

Our system has a specified naming mechanism to make our system program find the correct file during file decoding and decryption. Our system program keeps the original uploading file name when generates file and encryption key codeword symbol files. The rule of naming file codeword symbol file is attaching “\$#\_1\_ $\mathcal{T}$ ”, where  $\mathcal{T}$  is the index of file shares, between file name and its extension. Because of the computational limitation, our system program spits each file by 10MB before encrypting and encoding. For example, if our system program detects a 55MB file to execute file encryption and encoding, our system program will split the file into six shares and encrypt and encode each file shares afterward. To distinguish which codeword symbol file belongs to which file shares, our system add an index  $\mathcal{T}$  in codeword symbol file name, where  $0 \leq \mathcal{T} < (\text{file size} / 10\text{MB}) + 1$ . On the other hand, our system program only generates one encryption key codeword for each file, so there is no need of indexes for encryption key codeword symbol files. Our system program add “\$#\_2” between file name and its extension for encryption key codeword symbol files.

For example, if our system program detects a 123.mp3, whose size is 15 MB, our system program generates  $n$  123.\$#\_1\_0.mp3 and,  $n$  123.\$#\_1\_1.mp3 file codeword symbol files and  $n$  123.\$#\_2.mp3 encryption key codeword symbol files.

Our system program executes file decoding and decryption with the following steps

1. Collect  $k$  encryption key codeword symbols to recover encryption key.
2. Use encryption key and other information to recover file shares.
3. Combine all file shares to generate original file.

Our system program declares a string `request_key` to store the encryption key codeword symbol file name of executing file. Combining the sync folder paths and `request_key` to generate the absolute paths of symbol files, our system program is going to detect file according to the absolute paths. So far, our system program already knows the location of the encryption key codeword symbol files of executing files.

To decode files successfully, there are at least  $k$  sync folders which still contain their own encryption key codeword symbol file. Therefore, to recover encryption key, our system should ensure the amount of encryption key codeword symbol files is beyond the threshold  $k$ . If the amount is less than  $k$ , our system program will pop up a warning UI and skip all the following procedures and do the decoding and decryption of next file.

Among all the existing encryption key symbol files, our system will spend at most 1000 times to find out which combination can generate the submatrix of the generator matrix

whose determinant value is not equal to zero. If there is one, our system will keep the sequence of the  $k$  sync folders, whose encryption key codeword symbol file can contribute the inverse matrix of the submatrix; otherwise, our system program will skip all the subsequent computation.

An encryption key codeword symbol file contains two 152-bytes data in  $\mathbb{G}_1$ , one 152-bytes data in  $\mathbb{G}_2$  and  $k$  23-bytes data in  $\mathbb{Z}_r$ , where  $\mathbb{Z}_r$  is the ring of integers modulo  $r$ . Therefore, our system program can fetch each part of file content to obtain all information.

For instance, to read the information of generator matrix entry, our system program reads data from 456<sup>th</sup> position to  $(456 + k * 23)^{th}$  position of file byte array for fetching  $k$  entries information of generator matrix.

Our system program reads different parts of codeword symbol file byte array and use function `setFromByte()` to convert byte array to information in either  $\mathbb{G}_1$ ,  $\mathbb{G}_2$  or  $\mathbb{Z}_r$ . The following Table 1 shows how we transform byte array in a symbol file to element in different groups.



Position of $j^{th}$ key symbol file byte array	Element
byte array[0 to 151]	$A_j \in \mathbb{G}_1$
byte array [152 to 303]	$h_{ID} \in \mathbb{G}_1$
byte array [304 to 455]	$B_j \in \mathbb{G}_2$
byte array [456 to 456 + 23*k]	$\alpha_{i,j} \in \mathbb{Z}_r$

Table 1

Our system program reads k encryption key codeword symbol files and generates k byte arrays according to the sequence which is determined before. Furthermore, according to Table 1, our system program generates  $A_j$ ,  $h_{ID}$ ,  $B_j$  and  $\alpha_{i,j}$  by  $j^{th}$  byte array, where  $0 \leq i, j < k$ . To generate the submatrix of generator matrix, our system program fetches k 23-bytes data started from 456<sup>th</sup> byte of byte array and converts into k elements in  $\mathbb{Z}_r$ . Afterward, aligning k elements of  $j^{th}$  byte array to  $j^{th}$  column and forming a  $k \times k$  matrix. Because the determinant value of this matrix will not be zero, and our system program can use this matrix to compute the inverse matrix.

To compute the inverse matrix over  $\mathbb{Z}_r$ , we implement gauss elimination method of group\_gau\_eli class. This class provides the functionality of generating the inverse matrix over  $\mathbb{Z}_r$  of the input matrix. Therefore, our system program receives the return value of the

method  $\text{inv}()$  to obtain the inverse matrix.

By now, our system program collects all data and starts to decrypt and decode all the data to recover encryption key. First of all, our system program computes

$$w_j = B_j / \tilde{e}(A_j, \text{hid}^x) \quad 0 \leq j < k \quad (1)$$

, where  $x$  is user's private key generated by the arbitrary string user inserted in the beginning. Each  $w_j$  means the combination of plaintexts which stored in  $j^{\text{th}}$  sync folder, and  $w_j$  also means  $w_j = p_0^{\alpha_{0,j}} p_1^{\alpha_{1,j}} p_2^{\alpha_{2,j}} \dots p_{k-1}^{\alpha_{k-1,j}} \quad 0 \leq j < k$ , where  $p_j$  is one part

of encryption key. Then, our system program computes

$$p_j = \prod_{0 \leq i < n-1} (w_i)^{\gamma_{i,j}} \quad 0 \leq i, j < k \quad (2)$$

, where  $\gamma_{i,j}$  is the entry of inverse matrix.

Finally, we transform all  $p_j$  into strings and concatenate together to generate encryption key.

$$\text{encryption key} = p_0 || p_1 || \dots || p_{k-1} \quad (3)$$

After finishing generating encryption key, our system program will use this encryption key to decrypt each encrypted file shares. Then, our system program is going to decode codeword of each file shares and decrypt with the same encryption key.

As mentioned before, a file with large size may have many file codeword, therefore, our system program decodes and decrypts codeword according to the file shares index  $\mathcal{T}$ , where  $0 \leq \mathcal{T} < (\text{file size} / 10\text{MB}) + 1$ . Then, our system program will use FileChannel to

combine all file shares into a large file, which is the original file. In the following paragraphs, we will introduce how we system decodes and decrypts codeword one encrypted file share.

For decoding and decrypting a file codeword, our system program executes the following steps,

1. Choose the proper sequence of  $k$  sync folders.
2. Read files from  $k$  sync folders determined by previous step and generate the submatrix of generator matrix.
3. Convert all the entries of the submatrix to BigInteger and compute the inverse matrix
4. Convert  $k$  byte arrays into BigInteger and align into a BigInteger array.
5. Decoding by matrix multiplication
6. Convert the result to byte arrays
7. Base64 decode and combine together
8. AES decryption with the encryption key.

First, to decode file codeword, our system program will ensure that there are at least  $k$  sync folders which contain both file codeword symbol and encryption key codeword symbol, furthermore, the  $k$  encryption key codeword symbols can contribute a  $k \times k$  matrix whose determinant value is not zero. If yes, our system program will keep the indexes of  $k$  sync folders; otherwise, our system program cannot decode the codeword so that one of file shares cannot recover, hence, our system program will skip all the following decoding and

decryption steps and execute next file on file list.

According to the  $k$  indexes sequence, our system program will read  $k$  file codeword symbols and  $k$  encryption key codeword symbol files from sync folders by using `FileInputStream` class. Our system generates a  $k \times k$  matrix encryption key codeword symbols and converts every entries of this matrix to `BigInteger`. Then, our system program reads  $k$  file codeword symbols and stores in byte arrays. Our system program use the function `toBigInteger()` to cover each byte array to a `BigInteger` number and aligns them into a  $1 \times k$  `BigInteger` matrix.

Subsequently, our system program executes matrix multiplication between the  $1 \times k$  matrix and  $k \times k$  `BigInteger` inverse matrix of submatrix of generator matrix. Then, our system program converts all the entries of the  $1 \times k$  result matrix to byte array and executes each byte array with Base64 decoding. Finally, our system program combines all byte arrays together and decrypts it with the encryption key and the output is the original file share.

Our system program will repeat all step above until all file shares of the original file are done. Finally, our system program will combine all file shares together to generate the original file in the system folder.

After finishing the all files and folders recovery, our system program will start to scan system folder and sync folders with infinite loop. In other words, our system only executes

file decoding and decryption once, so our system does not support real-time file synchronizing mechanism.

### **3.5 Infinite scanning and detection**

After finishing decoding and decryption, our system program will scan system folder and sync folders with infinite loop to detect whether events happened. There are three events to trigger our system program to encrypt and encode,

1. New file is added in system folder.
2. File in system folder is modified.
3. The codeword symbol file in sync folders is lost when our system program is executing.

In this section, we will discuss how we implemented the scanning and detection functionality of our system program.

We implement infinite scanning and detection in Dec\_upload\_thread class. Our system program continuous scans system folder and sync folders, and more, our system program scans every files and folders iteratively.

To check the consistency between system folder and sync folders, our system program uses log files to record the information of current files in system folder. After finishing all files

decoding and decryption, our system program will edit log files first time. The method of editing log files is,

1. Each folder in system folder has its own log file, and the log file is named by the related path of the folder.
2. Each log file records information of contents in corresponding folder

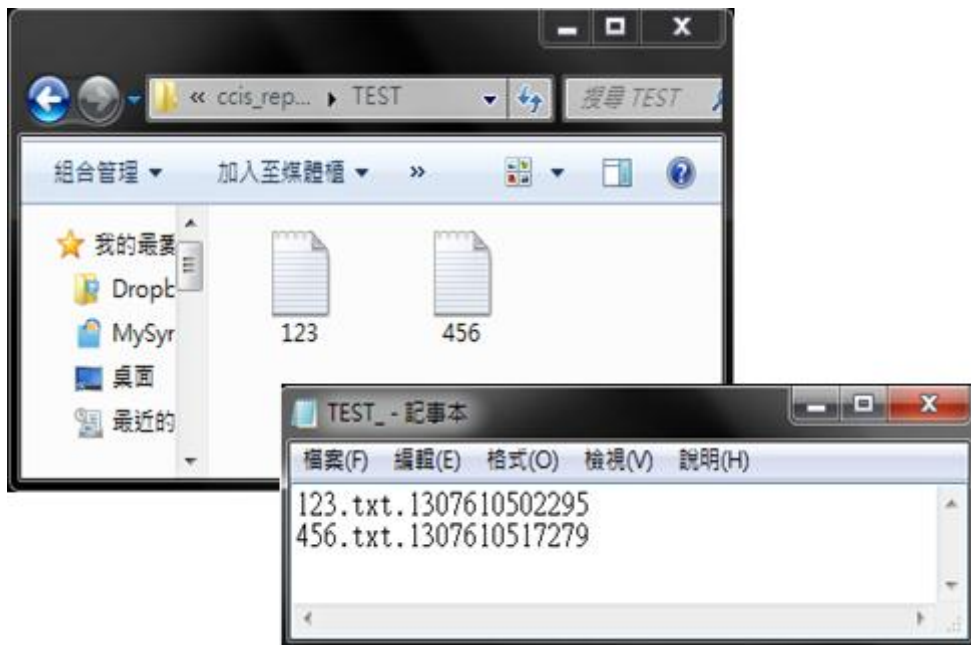


Figure 14

As shown in Figure 14, the log file contains the file names and attached by a long type value, which is the last modified value. The last modified value is generate by lastModified() of File class. Our system program uses the return value as a timestamp to check whether this file is modified or not. All log files are stored in LOG folder as the same directory as system

program is.

Our system program continuously scans each files and folders in system folder while execution. When our system program is reading a folder, our system program will read all the information of contents inside and compare them with the corresponding log file. Therefore, if the amount of contents is larger than records in the corresponding log file, our system program will encrypt and encode the new files or make a new folder at related path of n sync folders. Besides, our system program also check whether each last modified value of each content in system folder is equal to the timestamp in the log file. If not, it means that the file in system folder is modified, and our system program will encrypt and encode the file again, and make corresponding codeword be updated. After encryption and encoding, our system program will edit the log file again.

The other event is that the codeword symbol file is lost when our system program is executing. To detect this situation, our system program should scan every file in system folder, and compute its valid codeword symbol files number. When the amount of codeword symbol files is less than the valid number, our system program determines this as codeword symbol file is lost. Therefore, our system program will encrypt and encode the corresponding file again to make its codeword intact.

Our system program also detect whether the amount of current files in each folder of system folder is less than the records in its corresponding log file. Therefore, our system will

figure out which file in system folder is deleted and remove its all corresponding codeword.

### 3.6. Encryption and Encoding

Generally speaking, our system program encrypts and encodes files in system folder and writes codeword symbol files into each user-assigned sync folders. The encryption and encoding flow is show in Figure 15.

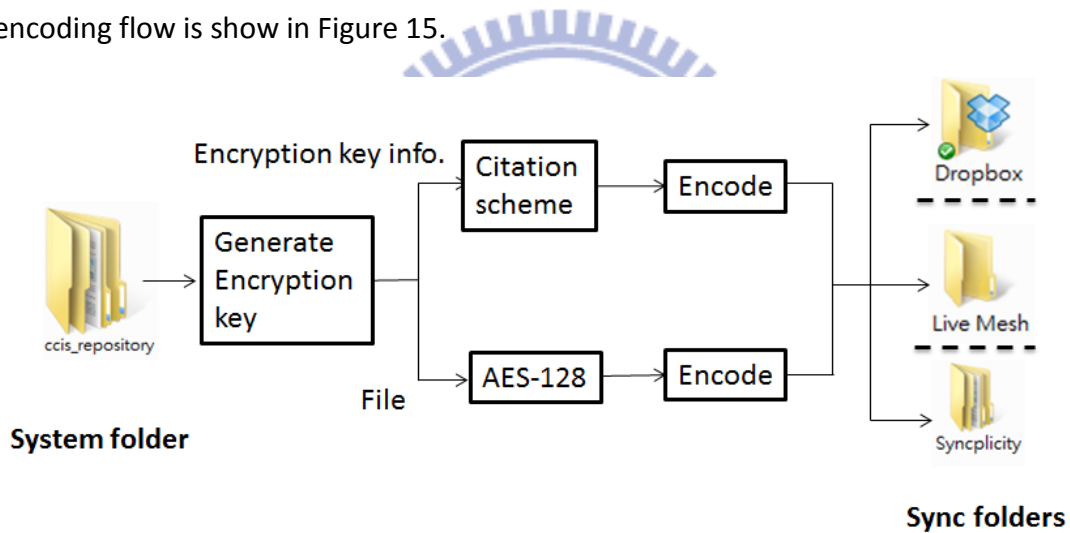


Figure 15

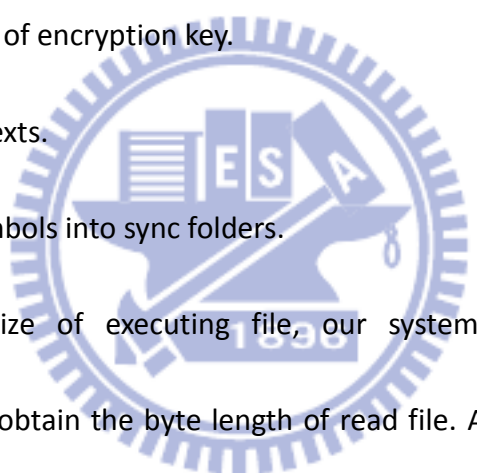
As mentioned before, our system program will figure out which file is going to be encrypted and encoded. Then, our system program will generate an encryption key to encrypt this file by AES-128, and encrypt the information of encryption key by [1].Furthermore, our system program will encode the two ciphertexts independently and



write codeword symbol files in each sync folders.

In the rest of this section, we will discuss how we implement the encryption and encoding functionality of our system program. After determining which file should be executed, our system program will execute the following procedures,

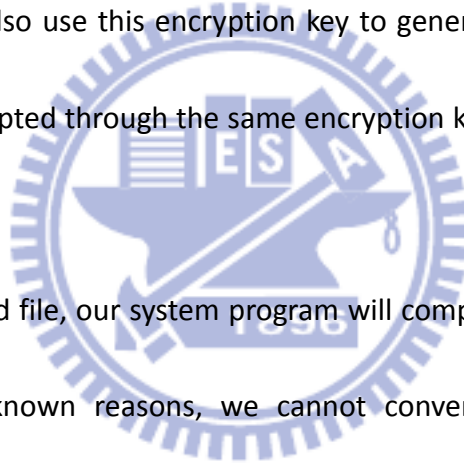
1. Split file by 10MB.
2. Generate an encryption key and encrypt file by AES.
3. Base64 encode.
4. Encrypt information of encryption key.
5. Encode two ciphertexts.
6. Write codeword symbols into sync folders.



To measure the size of executing file, our system program use `available()` of `FileInputStream` class to obtain the byte length of read file. Afterward, our system program divides the byte array by 10MB and the quotient plus one is the number of file shares. The next step is that our system program splits the original file into several `.tmp` files by using `FileChannel` class, and then, our system program will encrypt and encode each `.tmp` file with the same encryption key and generator matrix. All `.tmp` file will be deleted after encryption and encoding complete.

Then, our system program will generate an encryption key to encrypt all `.tmp` files. The encryption key is made of elements in  $\mathbb{G}_2$ , and our system program calls the member

function, `generate_encryption_key()`, to obtain  $k$  random elements in  $\mathbb{G}_2$ . Our system converts the  $k$  elements to strings and concatenates together, and the big string will be the key of AES encryption. Our system program adopts AES with key length 128, however, the encryption key is much longer than 128. Therefore, our system uses `init()` of `KeyGenerator` class to adopt encryption key as parameter to generate a random 128-bits string. In fact, our system program does not use encryption key to encrypt file directly, but uses encryption key to generate another 128-bits string. In other words, if the decipher owns the same encryption key, he can also use this encryption key to generate the same 128-bits string to decrypt files which encrypted through the same encryption key. Besides, our AES adopts CBC and PKCS5Padding.



To encode encrypted file, our system program will compute file and `BigInteger` number. However, for some unknown reasons, we cannot convert byte array of ciphertext to `BigInteger` directly. Fortunately, we found out that the byte array can be converted to `BigInteger` after it had been encoded by Base64. The tradeoff is that the ratio of file expansion is about 125% by using Base64 encoder.

In the next, our system program will encrypt the information of encryption key, which are the  $k$  elements in  $\mathbb{G}_2$ , through [1] protocols. According to [1], the ciphertext contain three elements

$$C_i = (\alpha_i, \beta, \gamma_i) = \left( g^{r_i}, h_{ID}, p_i \tilde{e}(g^x, h_{ID}^{r_i}) \right) \quad 0 \leq i < k \quad (4)$$

To compute  $C_i$ , our system program will randomly generate an element  $r_i$  in  $\mathbb{Z}_r$ . For example, an element in  $\mathbb{Z}_r$  is a quiet large number, such as 1741728797141823590670580703308133950434124779067146267. Our system program computes  $g$  to the power of  $r_i$  to obtain  $\alpha_i$ . Our system program makes a difference about the computation of  $h_{ID}$  with [1] because of the architecture difference. In [1],  $h_{ID}$  is computed by  $h_{ID} = H(M_1 || M_2 || \dots || M_k)$  and  $H : \{0,1\}^* \rightarrow \mathbb{G}_1$ , where  $M_i$  is the plaintext. In our system program architecture, however,  $h_{ID}$  is computed by  $g^{(H(\text{file name})) \bmod 100}$ . The third element  $\gamma_i$  is generated by computing  $p_i$ , which is the information of encryption key, multiplied by the paired number  $\tilde{e}(g^x, h_{ID}^{r_i})$ , where  $g^x$  is the user's public key.

After encrypting the file and the information of encryption key, our system program will generate a  $k \times n$  generator matrix to encode the two ciphertexts. The method of generating the generator matrix is that, for each row, our system program randomly chooses several positions to set random element in  $\mathbb{Z}_r$ , and the rest of this row is set zero. And then, our system program declares another  $k \times n$  BigInteger matrix, whose all entries are generated by converting the entries in the matrix over  $\mathbb{Z}_r$  to BigInteger. Therefore, to encode the ciphertexts of encryption key and file, our system program should prepare two generator matrixes, one is over  $\mathbb{Z}_r$  and the other is over infinite field.

This paragraph will introduce how our system program encodes the ciphertext of encryption key. Our system program encodes the ciphertext by [1],

$$\text{KeySymbol}_j = \left( A_j, \text{hid}, B_j, (\alpha_{0,j}, \dots, \alpha_{k-1,j}) \right) \quad 1 \leq j \leq n \quad (5)$$

, where  $A_j = \prod_{0 \leq i < n-1} (g^{r_i})^{\alpha_{i,j}}$  and  $B_j = \prod_{0 \leq i < n-1} (M_i \tilde{e}(g^x, \text{hid}^{r_i}))^{\alpha_{i,j}}$ . The computation of *hid* adopts the modification version and  $\alpha_{i,j}$  is the entry of the matrix whose entries are elements over  $\mathbb{Z}_r$ . After computation, our system program will convert all elements to byte arrays, as shown in Table 1, and the length of byte array converted from elements in  $\mathbb{G}_1$  and  $\mathbb{G}_2$  is 152 and elements in  $\mathbb{Z}_r$  is 23. Therefore, the size of an encryption key codeword symbol file is  $456+23*k$ . The  $j^{\text{th}}$  symbol file will be named by "file name.\$#\_2.extension" and stored in  $j^{\text{th}}$  sync folder.

In this paragraph, we will discuss the technique about encoding the ciphertext of file shares. As mentioned before, a large file will be split into several small file shares and all file shares will be encrypted by AES-128 with the same encryption key. Because the encoding procedures of all ciphertexts of file shares are the same, we will discuss this only once. After one byte array of file share is encrypted by AES and split into  $k$  byte arrays, which are followed by Base64 decoder. Afterward, our system program will convert each byte array to a BigInteger value and align into a  $1 \times k$  BigInteger array. In other words, our system program converts a file to a BigInteger array for computing, and our system program can recover file if our system program can retrieve the BigInteger array back. Then, our system program computes the matrix multiplication between  $1 \times k$  BigInteger array converted from file and the  $k \times n$  BigInteger generator matrix. The output is a  $1 \times n$  BigInteger array and each entry

is the BigInteger of codeword symbol. And then, our system program converts  $j^{th}$  entry to byte array and write in  $j^{th}$  sync folder, where  $0 \leq j < n$ . The codeword symbol file is named by "file name.\$#\_1\_ $\mathcal{T}$ .extension", where  $\mathcal{T}$  is the index of current executing file share and  $0 \leq \mathcal{T} < (\text{file size} / 10\text{MB}) + 1$ .



## Chapter 4

### Performance analysis

In this chapter, we will talk about our system performance and analysis the outcome.

The total time consumption of using our system is the sum of our system program computational time and the total time sync folders synchronize files with the storage cloud, hence, in the chapter, we only discuss our system program execution overhead.

Table 2 is our experimental environment.

CPU	Inter Core(TM) i7 Q720 @1.60 GHz
RAM	4GB
OS	Windows 7 64 bits
Development tool	Eclipse
JAVA Runtime Version	1.6
External Library	jPBC

Table 2

In the first testing model, our system program will execute five different sizes of rar files to calculate the time consumption of encryption and encoding process. And then, our system

program will calculate the time consumption of decoding and decryption processes of five difference files.

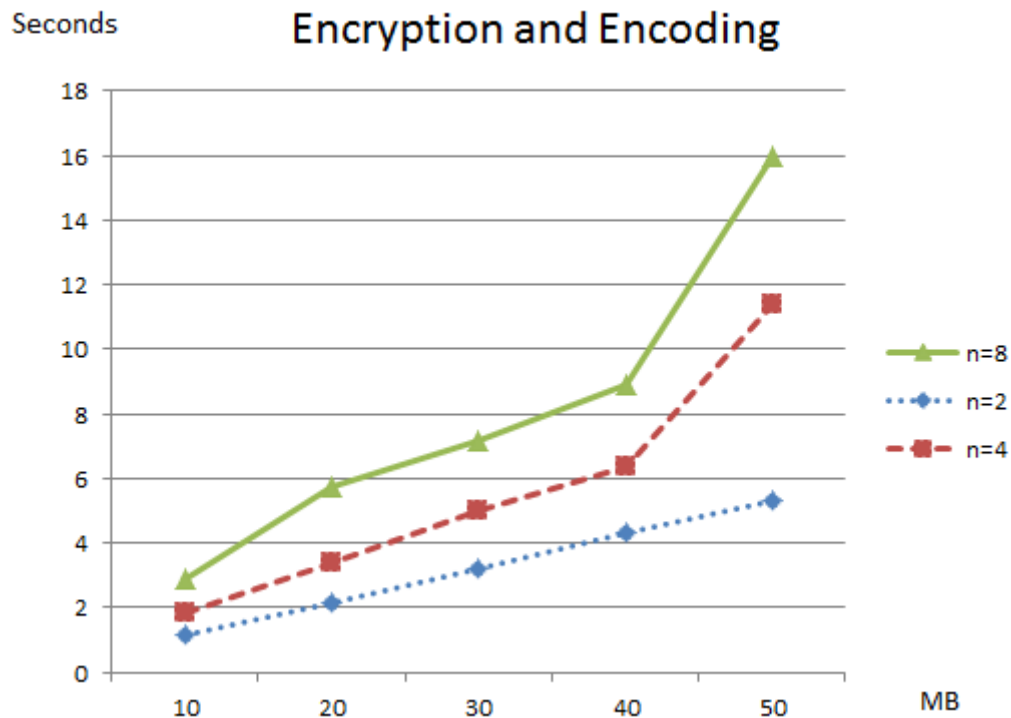


Figure 16

In Figure 16, we present that our system program executed five files (in x axis), whose sizes are different, encryption and encoding, and the corresponding time consumption (in y axis).

The three different lines represent that our system program executed five files with different parameters,  $n=2$ ,  $n=4$  and  $n=8$ . Each corresponding  $k$  is  $n-1$ . As shown in Figure 16, it is obviously that the larger file size is, the more time consumption our system program costs.

The time consumption grew as file size increase smoothly, however, when our system

program executed the 50MB file, the computational cost increased sharply. But unfortunately, this is hard to explain why so far. The other noticeable result is that, the higher  $n$  is, the more time consumption our system program should cost. The reason might be because the higher  $n$  means more sync folders to write codeword symbol files, and the file IO is always a bottleneck for many file system. But, the system with higher  $n$  provides much more erasure correcting ability; hence, it is a tradeoff users should choose.

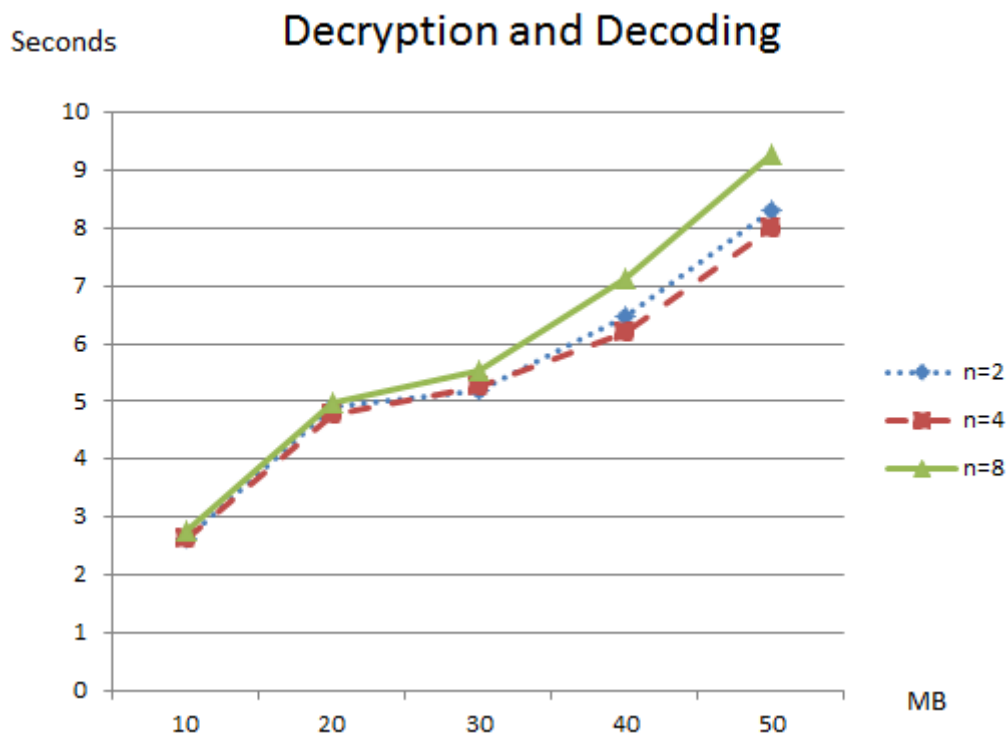


Figure 17

Figure 17 shows the time consumption of the decoding and decryption of five files.

An interesting discovery is that three lines are almost overlapped, and only the line, which



represented  $n=8$ , is higher than the other two lines. The explanation is still that more file IO will affect the computational overhead. According to Figure 17, we can easily know that the affection of the different number of sync folders our system adopts is small.

To compare with file encryption and encoding, file decryption and decoding cause less time consumption. Because there are more computations in file decryption and decoding than in file encryption and encoding, it is strange that decryption and decoding costs less time than file encryption and encoding. The reason why file encryption and encoding consume more time is that, our system program only can read file contents in system folder only after whole file is completed copied into system folder, otherwise, our system program has no permission to read bytes from the target file. Therefore, once if user adds a new file into system folder and expects our system program to execute file encryption and encoding right away, however, our system program must wait until the file is copied completely into system folder, and then start to execute file encryption and encoding. In summary, the encryption and encoding overhead should plus the time of file transfer.

In the next, we will discuss the two pie charts below. The first is the time consumption of each part in file encryption and encoding, and the other is the time consumption analysis of functionalities in file decryption and decoding. The test file is an mp3 file with 10MB.

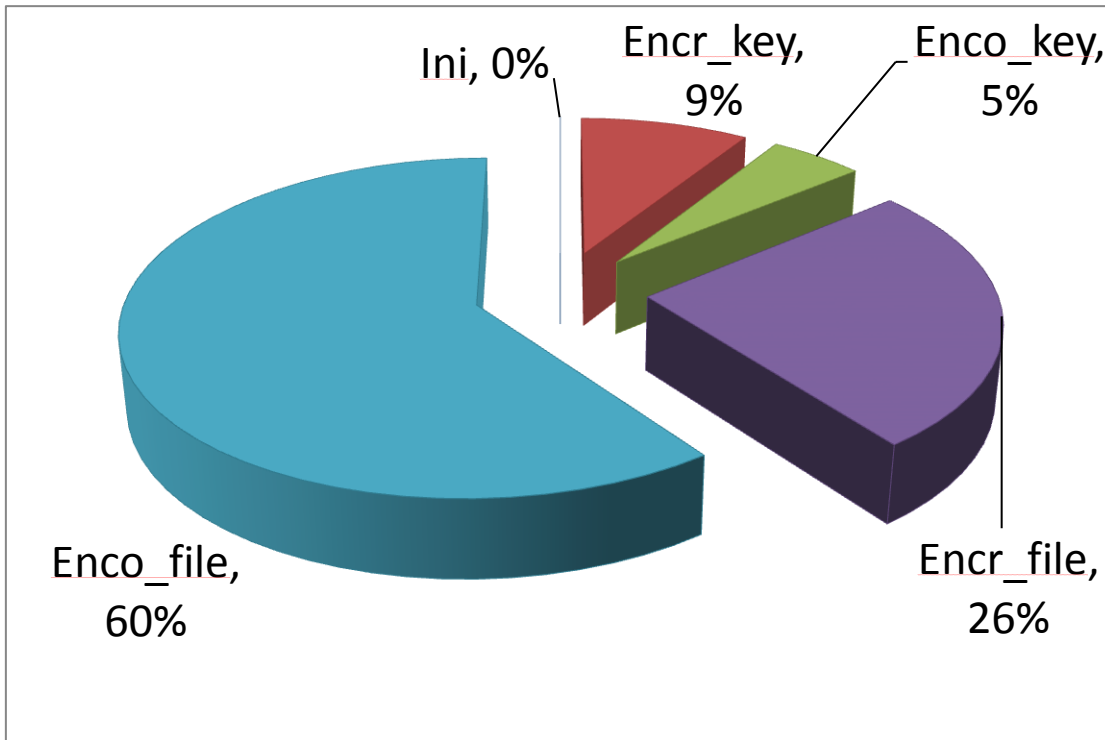


Figure 18

As shown in Figure 18, we counted five statistics about parameter setting (Ini, 0%), encryption of encryption key (Encr\_key, 9%), encoding of ciphertext of encryption key (Enco\_key, 5%), encryption of a file share (Encr\_file, 26%) and encoding of the ciphertext of file share (Enco\_file, 60%). It is not hard to observe that, the time consumption of encryption and encoding of file is the major problem. Because our system program adopts AES-128 to be our symmetric encryption tool, unless our system program change another faster symmetric encryption algorithm, our system program almost has nothing to do to improve the performance of this part. On the other hand, the encoding of the ciphertext of file share is a matrix multiplication. Although our system program can use BigInteger to compute file

correctly, the result value is extremely large. Therefore, the performance drops down according to the computation of large numbers. Unfortunately, we still have no better methods to improve the drawback of our system program.

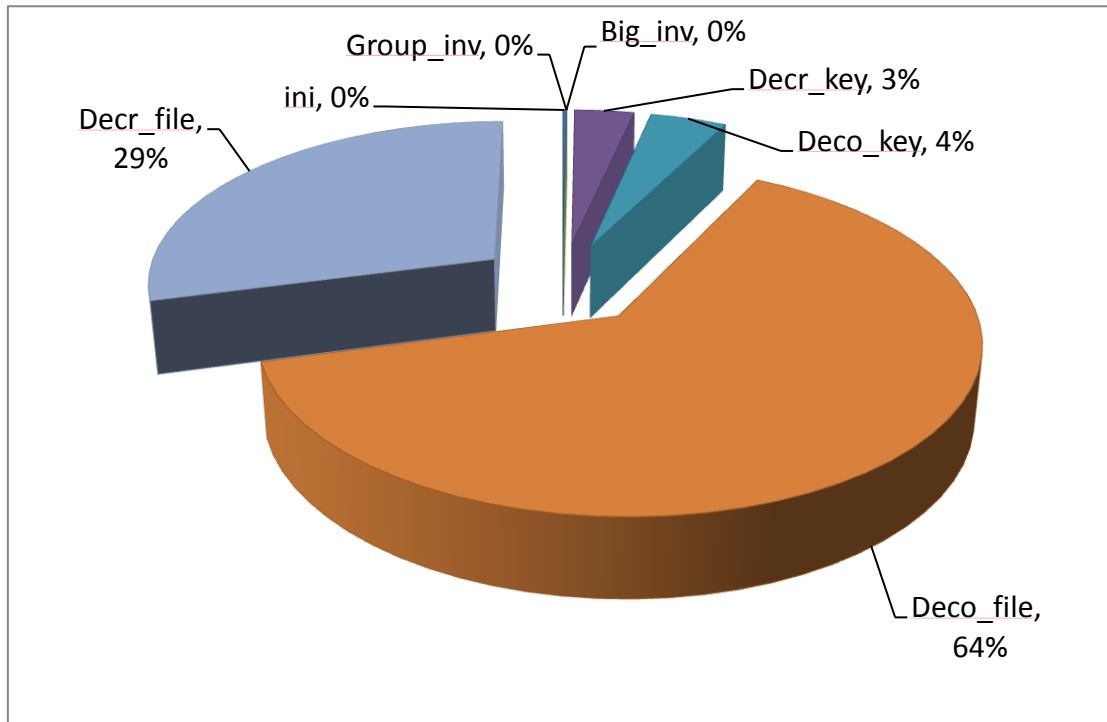


Figure 19

The Figure 19 shows the time consumption statistics of functionalities in file decryption and decoding, and our system program counted seven statistics to analysis, including the parameters setting (ini, 0%), the computation of inverse matrix over  $\mathbb{Z}_r$  (Group\_inv, 0%), the computation of inverse matrix over infinite field (Big\_inv, 0%), the decryption of encryption key (Decr\_key, 3%), the decoding of encryption key (Deco\_key, 4%), the decryption of encrypted file share (Decr\_file, 29%) and the decoding of file (Deco\_file, 64%). The time cost

of the either parameter setting or computation of inverse matrix over  $\mathbb{Z}_r$ , or the computation of inverse matrix over infinite field is less than 1ms, so the percentage is 0 in this graph. Then, we still talk about the bottleneck in decryption and decoding process, the file decryption (29%) and decoding (64%). Our system program adopts AES-128 decryption and is hard to improve the performance, too. However, the decoding of encrypted file is also the matrix multiplication of large numbers, so the execution time is quite large. In summary, the encoding and decoding of encrypted file share is the major problem to our system program.

Besides, because our system infinite loop to do continuous scanning and detection, we will provide the CPU and memory extra costs to execute our system program.

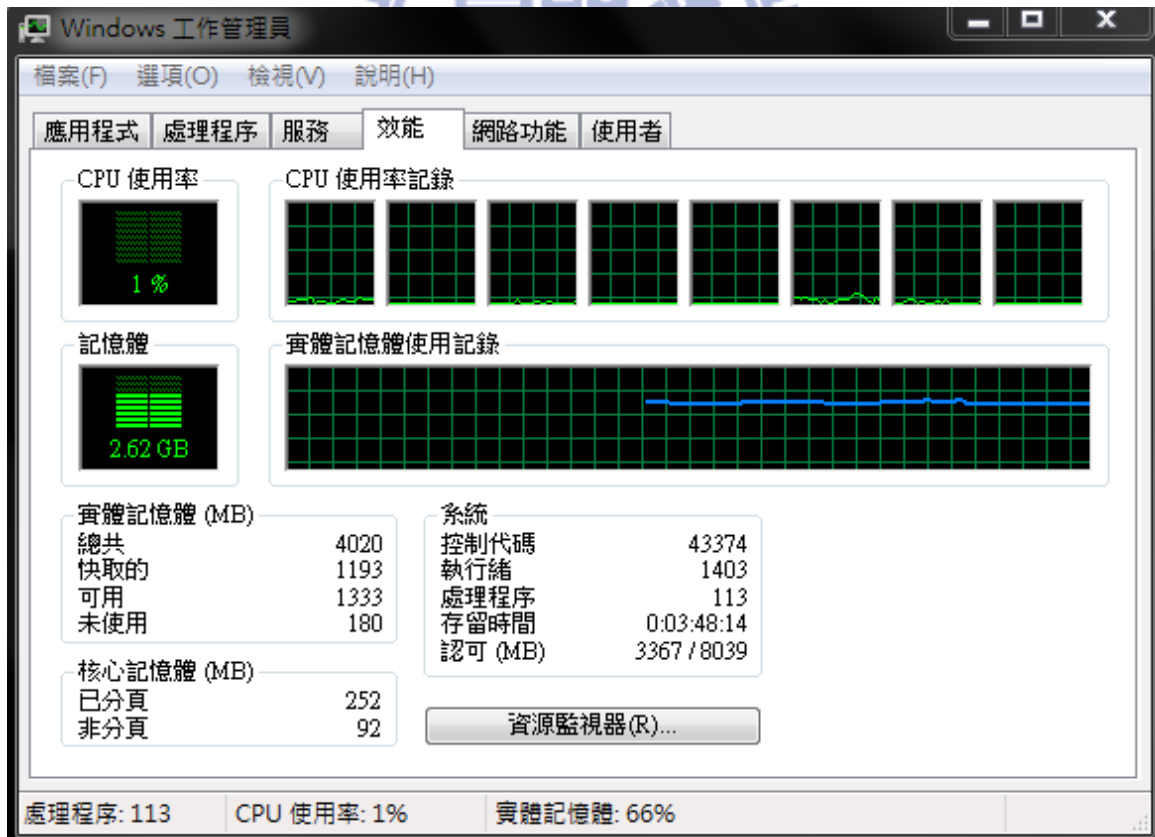


Figure 20

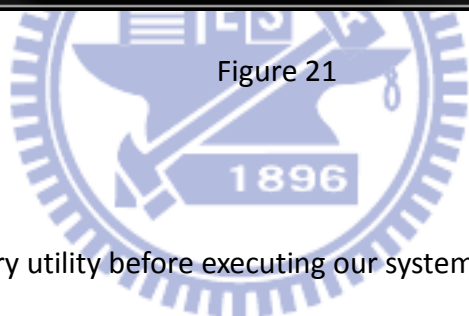
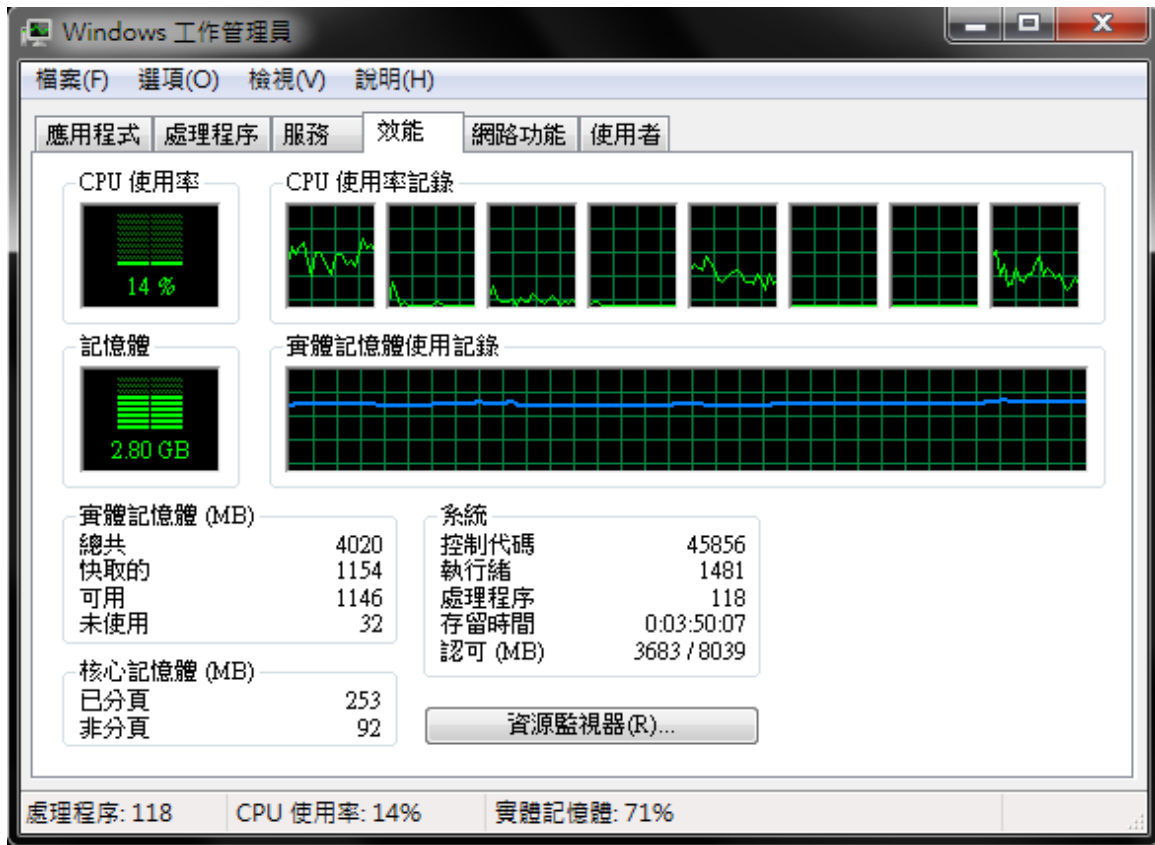


Figure 21

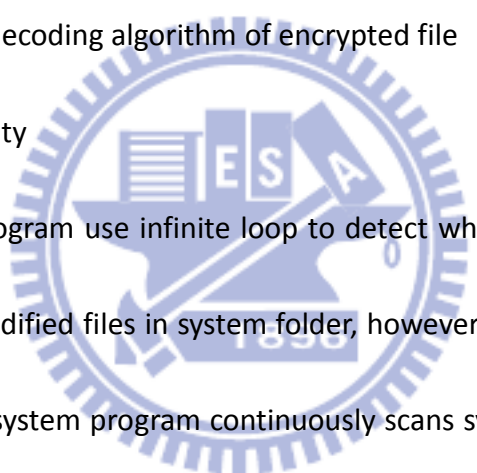
The CPU and memory utility before executing our system program is shown in Figure 20, and Figure 21 shows the CPU and memory utility when the experimental device extra executing our system program. Actually, to perform infinite loop, the CPU utility increase sharply, however, the memory utility increase slightly.

## Chapter 5

### Future work

To improve our system program, we list four possible schemes below.

1. The usage of extra libraries
2. The web interface of our system
3. New encoding and decoding algorithm of encrypted file
4. Error correcting ability

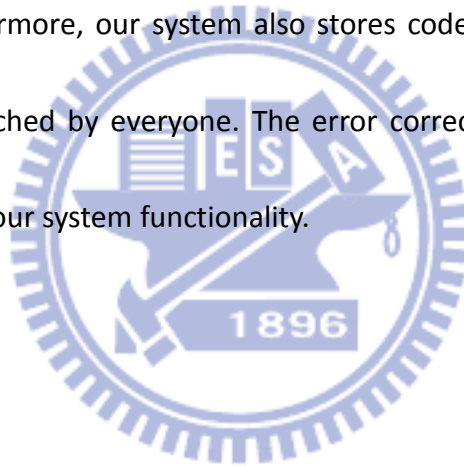


First, our system program use infinite loop to detect whether user added new file into system folder or user modified files in system folder, however, our system costs unnecessary CPU resource when our system program continuously scans system folder but user does not modify anything. Therefore, we wonder if there is another method to detect files changed in any operating system to reduce the CPU cost.

Second, our system synchronizes files with the cloud through sync folders. Therefore, the portability of our system is limited because our system program can run successfully if and only if the devices should own pre-installed synchronizing folder software and JRE. We can solve the problem above by implementing the web interface. User can upload, modified, download or delete file through website without any pre-install procedures.

Third, as illustrated in performance analysis section, the major bottleneck of our system is the performance of encoding and decoding of file, hence, we can emphasize that choose another algorithms of encoding and decoding. The possible resolution is that we can map the BigInteger number into a large enough group to reduce the computation cost.

Forth, our system adopts [1] erasure code to protect file privacy and robustness. However, once if the attacker modified the contents of codeword symbol files, our system program cannot recover file even if no symbol file lost. Our system program is lack of error correcting ability; furthermore, our system also stores codeword symbol files in local sync folder, which can be fetched by everyone. The error correcting ability is a very important issue if want to improve our system functionality.



## Chapter 6

### Conclusion

We implemented a software program together with current public network storage service system, such as Dropbox [11], LiveMesh [12] and so on, to provide users a new kind of network storage system, which is never published before. Users can use our system with simple operating interface and improve the privacy and robustness of files which are synchronized with the cloud. Our system program support any file formats, such as image, text audio, or video, and any size of file.

Our system program will generate a system folder while executing, users can add new files or folders into the system folder or modify the files in system folder. To synchronize contents in system folder with the cloud, our system program will encrypt and encode the corresponding files in system folder and write the codeword symbol files into the user-assigned sync folders and the codeword symbol files will be uploaded to the cloud automatically. Besides, when our system is launched, our system program will decode and decrypt all codeword in sync folders directly and write the original files into the system folder.

To improve file privacy, our system encrypt file by AES-128 with an encryption key,



which is encrypted by an asymmetric encryption algorithm [1]. Our system can guarantee that all the network storage servers are compromised, and the attacker cannot obtain any information about the original file.

To improve file robustness, our system provides a flexible environment to let users select the desirable erasure correcting ability our system supports. If the amount of lost codeword symbol files is below the threshold, our system program still can decode and recover the original file.



## Bibliography

- [ 1 ] Hsiao-Ying Lin, Wen-Guey Tzeng. "A Secure Decentralized Erasure Code for Distributed Networked Storage". IEEE Transactions on PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 21, NO. 11, NOVEMBER 2010
- [ 2 ] Mario Blaum, Jim Brady, Jehoshua Bruck, Jai Menon. "EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures". IEEE Transactions on COMPPUTERS. VOL. 44, FEBRUARY 1995.
- [ 3 ] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. "Row-Diagonal Parity for Double Disk Failure Correction". Proceedings of the Third USENIX Conference on File and Storage Technologies. March 31–April 2, 2004
- [ 4 ] Xu Lihao, V. Bohossian, J. Bruck, D.G. Wagner. "Low-density MDS codes and factors of complete graphs". IEEE Transactions on Information Theory. Vol 45, SEPTEMBER 1999.
- [ 5 ] Cheng Huang ,Lihao Xu, "STAR : An Efficient Coding Scheme for Correcting Triple Storage Node Failures". IEEE Transaction on Computers . Vol 57, JULY 2008
- [ 6 ] Angelo De Caro. "Java Pairing Based Cryptography Library".

<http://gas.dia.unisa.it/projects/jpbc/contact.html>

[ 7 ] "Eclipse". <http://www.eclipse.org/>

[ 8 ] "JWorld@TW". <http://www.javaworld.com.tw/jute/>

[ 9 ] Douglas R. Stinson. Cryptography : theory and practice third edition,  
Chapman & Hall/CRC, Boca Raton, 2006

[ 10 ] D. R. Hankerson, D. G. Hoffman, D. A. Leonard, C. C. Lindner, K. T. Phelps,  
C. A. Rodger, J. R. Wall. Coding Theory and Cryptography : the essentials  
second edition. New York, 2000

[ 11 ] "Dropbox". <https://www.dropbox.com/>

[ 12 ] "LiveMesh". <http://explore.live.com/windows-live-essentials?os=other>

[ 13 ] "Syncplicity". <http://www.syncplicity.com/>

[ 14 ] "Asus Webstorage". <http://www.asuswebstorage.com/navigate/>