

ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

EFFICIENT AND SECURE STORAGE METHOD

FOR

LARGE SCALE FILE SERVERS
UTILIZING CLIENT SIDE DE-DUPLICATION

By

Tesfamichael Gebregziabher

January 2015

Addis Ababa



ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

EFFICIENT AND SECURE STORAGE METHOD FOR LARGE SCALE FILE SERVERS UTILIZING CLIENT SIDE DE-DUPLICATION

A Thesis submitted to Addis Ababa Inistitute of Technology, School of Graduate Studies, Addis Ababa University in partial fulfillment of the requirements for the Degree of Master of Science in Computer Engineering

By

Tesfamichael Gebregziabher

Advisor

Srineevas Nune, PHD

January 2015

Addis Ababa



ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

EFFICIENT AND SECURE STORAGE METHOD FOR LARGE SCALE FILE SERVERS UTILIZING CLIENT SIDE DE-DUPLICATION

By

Tesfamichael Gebregziabher

Approved by the Board of Examiners:

Dr. Srineevas Nune	_____
Advisor	**Signature**
Fitsum Asamnew	_____
Examiner	**Signature**
Dr. Pandem Suresh	_____
Examiner	**Signature**

ACKNOWLEDGEMENT

I would like to forward the deepest of my appreciation and gratitude to my advisor Assistant Professor Sreenivas Nune (PhD) for his patience and constructive advice throughout the course of the thesis. Not only did he help me with invaluable advice, I have also learned a lot from him. His brotherly treatment has been an inspiration for the completion of this study.

I would also like to thank the School Electrical and Computer Engineering, for creating a conducive environment.

Finally I would like to thank Jigjiga University for sponsoring my M. Sc. study.

Table of Contents

Introduction 1
1.1. Background 1
1.2. Problem Statement 2
1.3. Objective 3
1.3.1 General Objective 3
1.3.2 Specific Objective 3
1.4. Methodology 4
1.5 Thesis Outline 5
Literature Review 6
Hash and Encryption Standards 8
3.1 Secure Hash Standard (SHS) 8
3.2 Advanced Encryption Standard (AES) 9
3.2.1 Description of AES Algorithm 10
3.2.2 High-level description of the AES algorithm 10
Client Side De-duplication 22
4.1 The Challenge of Client Side de-duplication 23
4.2 Convergent Encryption 24
Analysis, Design and Implementation 28
5.1 Overview of proposed scheme 28
Case I: First upload. 28
Case II: Subsequent uploads. 29
5.1.1 Storage Manager 31
5.1.2 Proof of Ownership Verifier 32
5.2. Implementation of the system 34
5.2.1. The Server Side Application 35
5.2.2 The Client Component 48
Result and Discussion 54
Conclusion and Recommendation 59
Bibliography 60
Appendix I - Source code of server UI 68

Appendix II Source code of the authentication service of server component implementation 70
Appendix III source code of the implementation of the proof service 75
Appendix IV source code of the implementation of File Transfer Service 86
Appendix V Source code Helper classes of the implementation of server component 92
Appendix VI source code configuration of the server component..... 103
Appendix VII Source code of the implementation ILogIn interface on the client 105
Appendix VIII Source code of the implementation IProtocol interface on the client 107
Appendix IX Source code of the implementation IFileTransfer interface on the client 111

LIST OF FIGURES

Figure 1 The SubBytes Step 13
Figure 2 The ShiftRows Step 14
Figure 3 The MixColumns step 15
Figure 4 The AddRoundKey step 15
Figure 5 an example of users trying to upload the same file. 22
Figure 6 Example of file de-duplication 23
Figure 7 The Challenge: Encryption vs de-duplication 23
Figure 8 Convergent Encryption 25
Figure 9 Program flow of the proposed system 30
Figure 10 Example of merkle hash tree with four blocks 33
Figure 11 Merkle Tree and Proof of Ownership 34
Figure 12 Server Side Data Model 35
Figure 14 Services of the Server Component 37
Figure 17 Sequence Diagram of User Registration 40
Figure 18 Flow Chart of User Registration 41
Figure 19 Flow Chart of Login method 42
Figure 20 Flow Chart of ListUserFiles Method 43
Figure 23 Proof of Ownership Protocol 45
Figure 26 Flowchart of Upload Method 47
Figure 27 MVVM design Pattern 49
Figure 28 Login/Register Window 49
Figure 29 Main window when upload tab is selected 50
Figure 30 Main window when download tab is selected 50
Figure 31 Code-Map of the de-duplication client application 51
Figure 32 Duplication of Electrical Engineering files with respect to other disciplines 54
Figure 33 Summary of File duplication across all departments 55

LIST OF TABLES

Table 1 Secure Hash Algorithm Properties..... 9
Table 2 Functions of FileData table columns. 35
Table 3 Specification of the experiment machines 52
Table 4 Experiment Parameters 53
Table 6 Ellapsed time of first upload vs subsequent uploads 56
Table 7 Worldwide leaders in cloud storage..... 58

CODE LISTINGS

Listing 1 38

Listing 2 39

Listing 3 39

Listing 4 44

Listing 5 46

Acronyms

AES	Advanced Encryption Standard
CDN	Content Distribution Network
DES	Data Encryption Standard
FLOPS	Floating point operations per second
GF	A finite field or Galois field (so named in honor of Évariste Galois).
HTTP	Hyper Text Transport Protocol
IDC	International Data Corporation
NIST	National Institute of Standards
NSA	National Security Agency
POW	Proof of ownership
SHA	Secure Hash Algorithm
SHS	Secure Hash Standard
UI	User Interface
UTP	Unshielded twisted pair cables
WCF	Windows Communication Foundation
WPF	Windows Presentation Foudantion.

Abstract

According to a recent survey by Iternational Data Corporation [63], 75% of today's digital data are duplicated copies. To reduce the unnecessarily redundant copies, the storage servers would handle duplication (either at a file level or chunks of data sized 4KB and larger). De-duplication can be managed both at the **server-side** and the **client-side.** In order to identify duplicated copies, it is required that files be un-encrypted. However users may be worried about the security of their files and may want their data to be encrypted. However encryption makes cipher text indistinguishable from theoretically random data, i.e., encrypted data are always distributed randomly, so identical plaintext encrypted by randomly generated cryptographic keys will very likely have different cipher texts which cannot be de-duplicated. In this research, a method that resolves the **conflict** between **de-duplication** and **encryption** is presented.

Keywords - Cloud Storage, Client-side de-duplication, Proof of Ownership, File Server, Security, Secure Hash Standard, Advanced Encryption Standard, Windows Communication Foundation, Windows Presentation Foundation, and Dot Net Framework.

Chapter One
Introduction

1.1. Background

Currently, commercial large scale storage services including Microsoft Skydrive, Amazon and Google drive Storage have attracted millions of users. While data redundancy was once an acceptable operational part of the backup process, the rapid growth of digital content in the data center has pushed organizations to rethink how they approach this issue and to look for ways to optimize storage capacity utilization across the enterprise. Explosive data growth over the recent years has brought much pressure on the infrastructure and storage management.

The Flud backup system [4] and Google [6] etc can save on storage costs by removing duplication. According to a recent survey by IDC [63], 75% of today's digital data are duplicated copies. To reduce the unnecessarily redundant copies, the storage servers would handle duplication (either at a file level or chunks of data sized 4KB and larger) by keeping only one or few copies for each file and making a link to the file for every user who asks to store the file, regardless of how many copies there are. The copies are replaced by pointers which reference the original block of data in a way that is seamless to the user, who continues to use a file as if all of the blocks of data it contains are his or hers alone.

Duplication can be managed both at the server-side and the client-side; client-side de-duplication is mostly known for effectively

1. Reduced band width requirement
2. Reduced storage space requirement
3. Lower electric consumption (hence a greener environment)
4. Lower overall cost of storage.

The server side de-duplication method has the following features

1. No reduction of network trafic.
2. Increased server load. As all tasks are done at the server no Load Balancing scheme can be applied.
3. Clents do not have a role in generating their encryption key.

1.2. Problem Statement

File duplication is a common scenario in large scale servers. A good example of large scale file server is a cloud server. As the number of clients increases so does the chances of duplication. File de-duplication (removal of duplicate copies) is crucial to maintain storage efficiency and it can be performed either on the server or on the client machine. If the file de-duplication is performed on client side, it would not only save storage space but also reduce the network band width requirement.

As users may legitimately worry about the security of their files, the issue of security should also be addressed alongside with de-duplication. How to implement client-side de-duplication securely in an un trusted environment, is far more challenging than it seems. Convergent encryption [26], a client-side de-duplication method, has been used by cloud storage services such as **Farsite, Permabit, Freenet, MojoNation, GNUnet, flud and Bitcasa.** However convergent encryption [26] and its derivatives have security vulnerabilities.

The assumption is the cloud user is potentially un trusted, since anyone from the Internet could become a cloud user and the cloud server is unable to distinguish honest users from malicious users (i.e. adversaries) in general. Hence, in order to provide de-duplication service securely, a private key must be shared between the owners of the file without the knowledge of the service provider.

Cloud users may want their data be **encrypted** before it is uploaded to a server. To mention some reasons:

1. They may not want the storage provider to see their data.

2. Even if they trust the storage provider, they may be worried about errant employees.

3. The server may be hacked by external adversary

Therefore they will encrypt their data before they upload it to the server. This, however, makes de-duplication impossible. Say Tesfa stores not his file F but its encryption C_T under his password PWD_T. Michael would store C_M, the encryption of F under his password PWD_M. Two issues arise: (1) how the server is to detect that the data underlying the two cipher texts is the same, and (2) even if it can so detect, what can it store short of (C_T;C_M) that allows both parties, based on their separate respective passwords, to recover the data from what is stored.

Just storing Tesfa's cipher text, for example, does not work because Michael cannot later decrypt it to recover the file, and vice versa. While de-duplication exploits identical content, encryption attempts to make all content appear random. Thus, combining the space efficiency of de-duplication with the secrecy aspects of encryption is problematic.

The significance of this research is how to resolve the **contradiction between security** and **storage efficiency.**

1.3. Objective

1.3.1 General Objective

The General objective of this research is to fulfill three important factors that affect storage efficiency and file security.

1. Provide data security (concern of the client).

2. Maintain storage efficiency (concern of the storage provider).

3. Maintain Bandwidth efficiency (concern of both the user and the service provider) .

1.3.2 Specific Objective

The goal, in this research, is to formulate a mechanism where

- The confidentiality of client data is preserved
- Storage efficiency is maintained
- Network bandwidth requirement is reduced

Further more a method should be formulated that is secure against malicious insiders working at the storage provider and external attackers with access to communication channels between clients and the Storage Server. This method should implement client-side de-duplication and the confidentiality of the file should be preserved even if the storage server is compromised by an external adversary.

1.4. Methodology

The **Quantitative Reasearch Methodology** that follows a linearized form of scientific methody was applied in this research.

1. Define the question
2. Gather information and resources
3. Form an explanatory hypothesis
4. Test the hypothesis by performing an experiment and collecting data in a reproducible manner
5. Analyze the data
6. Interpret the data and draw conclusions that serve as a starting point for new hypothesis
7. Publish results to communicate them to others

This research study follows and non-controlled experimental or intervention study. The study sample is the electronic library of the College of Engineering of Jigjiga University. The total size is 128.27GB. The content of this files are static that is they are ment to be read. They are not written to be modified hence it would suit perfectly for this reasearch.

I have conducted rigorous literature review with regard to de-duplication. My sources include electronic books, printed books and online publications.

1.5 Thesis Outline

The next section discusses review of available literature with regard to de-duplication. Brief description of secure hash standard and advanced encryption standard is discussed in section three. The client side de-duplication and its challenges are discussed in section four. Convergent encryption and its security vulnerabilities are discussed briefly in section five. In section six, the analysis, design and implementation of the proposed system is discussed. The results found during the experimentation are demonstrated in section seven. Finally, conclusions and recommendations are given in the last section.

Chapter Two
Literature Review

Convergent encryption (CE) was first discovered by Douceur[26] et al. in the Farsite distributed file system [26] where the hash of the data was used as the encryption key. This makes identical plaintexts be encrypted to the same cipher text regardless of which user they belonged to. In this scheme the key is derived deterministically from the file.

Douceur [26] studied the problem of duplication in large scale servers in which de-duplication has to be reconciled with confidentiality. The authors propose the use of convergent encryption, i.e., deriving keys from the hash of the plaintext, so that two users will produce the same cipher text from the same plaintext block, and the cipher text can then be de-duplicated. Hence "secure" client-side duplication would be possible.

Storer [45] pointed out some security problems of convergent encryption, proposing a security model and two protocols for secure data duplication management. This securuty model is based on two secure de-duplication models: authenticated and anonymous. Although this scheme was able to solve the problems of poison attack, it does not address many of the potential attacks.

[56, 57, 58, 59, 60] have built a secure file system on top of a storage server. However this breaks de-duplication mechanisms. The security implemntations of these schemes has been a great experience to this thesis.

Convergent encryption (CE) based solutions [7, 45] provide security only for unpredictable messages even in the best case, and are vulnerable to brute-force attacks. The simple approach of sharing a secret key across clients with a deterministic encryption scheme [11, 55] fails to achieve compromise resilience.

D. Harnik and Pinkas [32] first discuss the security related shortcomings of client-side duplication management. Several works [51, 52, 53, 54] have looked at this security problem, but none provide solutions that meet all security vulnerabilities against malicious insiders working at the provider and external attackers with access to communication channels between clients and the Storage Server.

Halevi [31] targets the critical security vulnerability in the convergent encryption's "hash-as-a-proof" method. Their work proposes a cryptographic primitive called "proofs of ownership" (POW). Unfortunately, Halevi's [31] formulation does not address confidentiality of data against the cloud storage server. Users have every reason not to trust the cloud server. While insiders of the could server may be errant; the cloud server may be hacked as well. In addition, the cloud server administrators may make careless technical mistakes, which may expose user data to unauthorized persons. Furthermore the scheme has extremely high I/O operations and heavy computations at the client.

Ng [51] solution is an unsatisfactory as it suffers from "divide and conquer" attack. As pointed out by Xu [52], Ng [51].

Roberto Di Pietro and Alessandro Sorniotti[53] dealt with the proof that if a client really possesses the file, The file F ($F[i_1,...,F[i_\lambda]$) is projected onto λ randomly chosen bit-positions ($i_1,....,i_\lambda$) as the "proof" of ownership of file F. , This work is extremely efficient however similar to the "hash-as-a proof" method it is insecure.

Mironov, Pandey, Reingold and Segev [38] suggest de-duplication as a potential application of their incremental deterministic public-key encryption scheme. But this will only work with a single client. It won't allow de-duplication across clients, since they would all have to share the secret key.

Mihir Bellare, Sriram Keelveedhi and Thomas Ristenpart [54], have finally come up with Server-Aided Encryption for De-duplicated Storage. However their scheme relies on limiting the rate of user requests to the Key Server so as to avoid brute force attacks.

In summary, although the conflict between de-duplication and encryption has been researched for years, there are still challenging problems to solve, such as trading off de-duplication and encryption under different security requirements, protecting from information leakage, lowering performance overhead for additional operations, key management for shared data chunks among large groups of users, etc.

Chapter Three
Hash and Encryption Standards

3.1 Secure Hash Standard (SHS)

Issued by the National Institute of Standards and Technology (NIST) [61]

1. **Name of Standard**: Secure Hash Standard (SHS) (FIPS PUB 180-4).
2. **Category of Standard**: Computer Security Standard, Cryptography.
3. **Explanation**: This Standard specifies secure hash algorithms -SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 and SHA-512/256 -for computing a condensed representation of electronic data (message). When a message of any length less than 2^{64} bits (for SHA-1, SHA-224 and SHA-256) or less than 2^{128} bits (for SHA-384, SHA-512, SHA-512/224 and SHA-512/256) is input to a hash algorithm, the result is an output called a message digest. The message digests range in length from 160 to 512 bits, depending on the algorithm. Secure hash algorithms are typically used with other cryptographic algorithms, such as digital signature algorithms and keyed-hash message authentication codes, or in the generation of random numbers (bits).

The hash algorithms specified in this Standard are called secure because, for a given algorithm, it is computationally infeasible

1) To find a message that corresponds to a given message digest, or
2) To find two different messages that produce the same message digest.

   Any change to a message will, with a very high probability, results in a different message digest. This will result in a verification failure when the secure hash algorithm is used with a digital signature algorithm or a keyed-hash message authentication algorithm.

These hash algorithms differ most significantly in the security strengths that are provided for the data being hashed. The security strengths of these hash functions and the system as a whole when each of them is used with other cryptographic algorithms, such as digital

signature algorithms and keyed-hash message authentication codes is half the length of its hash value. This is currently believed to be true for all the approved hash functions except SHA-1. The latest cryptanalytic results for SHA-1 [SHA1 Attack] indicate that it may have a collision resistance strength that is considerably less than its expected strength of 80 bits.

Additionally, the algorithms differ in terms of the size of the blocks and words of data that are used during hashing or message digest sizes. Table 2 presents the basic properties of these hash algorithms.

Algorithm	Message Size	Block Size	Word Size	Message Digest Size
(bits)	**(bits)**	**(bits)**	**(bits)**	**(bits)**
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512
SHA-512/224	$< 2^{128}$	1024	64	224
SHA-512/256	$< 2^{128}$	1024	64	256

Table 1 Secure Hash Algorithm Properties

3.2 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) [64] in 2001.

AES is based on the Rijndael cipher developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, who submitted a proposal to NIST during the AES selection process. Rijndael is a family of ciphers with different key and block sizes. For AES, NIST selected three members of the Rijndael family, each with a block size of 128 bits, but three different key lengths: 128, 192 and 256 bits.

AES has been adopted by the U.S. government and is now used worldwide. It supersedes the Data Encryption Standard (DES)[65], which was published in 1977. The algorithm described

by AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data.

3.2.1 Description of AES Algorithm

AES is based on a design principle known as a substitution-permutation network, combination of both substitution and permutation, and is fast in both software and hardware. Unlike its predecessor DES [65], AES does not use a Feistel network. The Rijndael specification is specified with block and key sizes that may be any multiple of 32 bits, both with a minimum of 128 and a maximum of 256 bits.

AES operates on a 4×4 column-major order matrix of bytes, termed the state, although some versions of Rijndael have a larger block size and have additional columns in the state.

Most AES calculations are done in a special finite field.

The key size used for an AES cipher specifies the number of repetitions of transformation rounds that convert the input, called the plaintext, into the final output, called the cipher text. The number of cycles of repetition are as follows:

10 cycles of repetition for 128-bit keys.

12 cycles of repetition for 192-bit keys.

14 cycles of repetition for 256-bit keys.

Each round consists of several processing steps, each containing four similar but different stages, including one that depends on the encryption key itself. A set of reverse rounds are applied to transform cipher text back into the original plaintext using the same encryption key.

3.2.2 High-level description of the AES algorithm

KeyExpansions—round keys are derived from the cipher key using Rijndael's key schedule. AES requires a separate 128-bit round key block for each round plus one more.

InitialRound

1. **AddRoundKey**—each byte of the state is combined with a block of the round key using bitwise xor.

Rounds

1. **SubBytes**—a non-linear substitution step where each byte is replaced with another according to a lookup table.
2. **ShiftRows**—a transposition step where the last three rows of the state are shifted cyclically a certain number of steps.
3. **MixColumns**—a mixing operation which operates on the columns of the state, combining the four bytes in each column.
4. **AddRoundKey**

Final Round (no MixColumns)

1. **SubBytes**
2. **ShiftRows**
3. **AddRoundKey.**

Galois Field

In algebra, a finite field or Galois field (so named in honor of Évariste Galois) is a field that contains a finite number of elements, called its order (the size of the underlying set). As with any field, a finite field is a set on which the operations of commutative multiplication, addition, subtraction and division (by anything except zero) have been defined. Common, but not the only, examples of finite fields are given by the integers modulo a prime, that is, the integers mod n where n is a prime number.

A field is a set of elements *F* for which addition, multiplication, subtraction and division performed with its elements result in another element of the same set. The definition of a field is based on the operations described over such a field. For addition and multiplication operations, the following conditions define a field

1. *F* is a commutative group with respect to the addition operation. The identity element for the addition is called '0'.

2. F is a commutative group for the multiplication operation. The identity element for multiplication is called '1'.
3. Multiplication is distributive with respect to addition:

$$a(b + c) = ab + ac \quad (1)$$

The number of elements of a field is called the order of that field.

The inverse for the addition operation of an element of the field $a \in F$ is denoted as $-a$, and inverse for the multiplication operation of an element of the field is denoted as a^{-1}. Subtraction and division operations are defined as a function of the inverse elements as

$$a - b = a + (-b)$$

$$a/b = a(b^{-1}) \quad (2)$$

Affine Transformation

An affine transformation is any transformation that preserves collinearity (i.e., all points lying on a line initially still lie on a line after transformation) and ratios of distances (e.g., the midpoint of a line segment remains the midpoint after transformation). In this sense, affine indicates a special class of projective transformations that do not move any objects from the affine space to the plane at infinity or conversely. An affine transformation is also called an affinity.

Geometric contraction, expansion, dilation, reflection, rotation, shear, similarity transformations, spiral similarities, and translation are all affine transformations, as are their combinations. In general, an affine transformation is a composition of rotations, translations, dilations, and shears.

An affine transformation preserves:

1. The collinearity relation between points; i.e., points which lie on the same line (called collinear points) continue to be collinear after the transformation.
2. Ratios of vectors along a line; i.e., for distinct collinear points P_1, P_2, P_3, the ratio of $\overrightarrow{P1P2}$ and $\overrightarrow{P2P3}$ is the same as that of $\overrightarrow{f(P1)f(P2)}$ and $\overrightarrow{f(P2)f(P3)}$.

An affine transformation is invertible if and only if A is invertible.

While an affine transformation preserves proportions on lines, it does not necessarily preserve angles or lengths. Any triangle can be transformed into any other by an affine transformation, so all triangles are affine and, in this sense, affine is a generalization of congruent and similar.

The SubBytes Step

In the SubBytes step, each byte $a_{i,j}$ in the *state* matrix is replaced with a SubByte $S(a_{i,j})$ using an 8- bit substitution box, the Rijndael S-box. This operation provides the non-linearity in the cipher. The S-box used is derived from the multiplicative inverse over GF (2^8) [64], known to have good non-linearity properties. To avoid attacks based on simple algebraic properties, the S-box is constructed by combining the inverse function with an invertible affine transformation. The S-box is also chosen to avoid any fixed points (and so is a derangement), i.e., $S(a_{i,j}) \neq a_{i,j}$, and also any opposite fixed points, i.e., $S(a_{i,j}) \oplus a_{i,j} \neq$ 0xFF. While performing the decryption, Inverse SubBytes step is used, which requires first taking the affine transformation and then finding the multiplicative inverse (just reversing the steps used in SubBytes step).

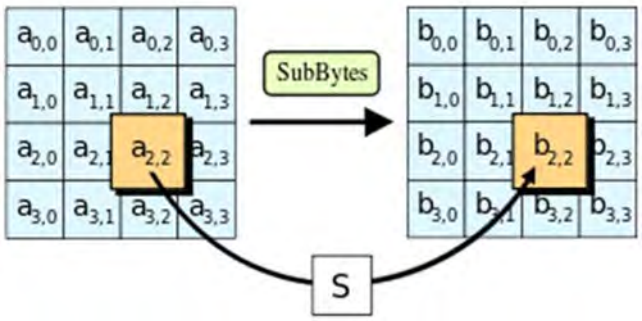


Figure 1 The SubBytes Step

The ShiftRows Step

The ShiftRows step operates on the rows of the state; it cyclically shifts the bytes in each row by a certain offset. For AES, the first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively. For blocks of sizes 128 bits and 192 bits, the shifting pattern is the same. Row n is shifted left circular by n-1 bytes. In this way, each column of the output state of the ShiftRows step is composed of bytes from each column of the input state. (Rijndael variants

with a larger block size have slightly different offsets). For a 256-bit block, the first row is unchanged and the shifting for the second, third and fourth row is 1 byte, 3 bytes and 4 bytes respectively—this change only applies for the Rijndael cipher when used with a 256-bit block, as AES does not use 256-bit blocks. The importance of this step is to avoid the columns being linearly independent, in which case, AES degenerates into four independent block ciphers.

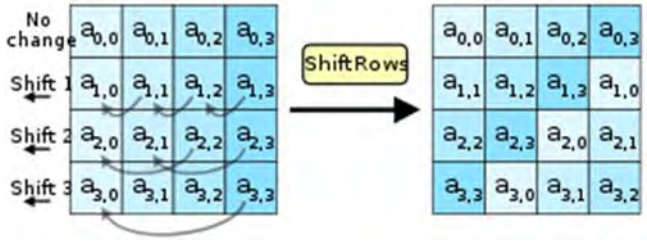


Figure 2 The ShiftRows Step

The MixColumns step

In the MixColumns step, the four bytes of each column of the state are combined using an invertible linear transformation. The MixColumns function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes. Together with ShiftRows, MixColumns provides diffusion in the cipher. During this operation, each column is multiplied by a fixed matrix:

Matrix multiplication is composed of multiplication and addition of the entries, and here the multiplication operation can be defined as this: multiplication by 1 means no change, multiplication by 2 means shifting to the left, and multiplication by 3 means shifting to the left and then performing XOR with the initial unshifted value. After shifting, a conditional XOR with 0x1B should be performed if the shifted value is larger than 0xFF. (These are special cases of the usual multiplication in **GF**(2^8)[64].) Addition is simply XOR.

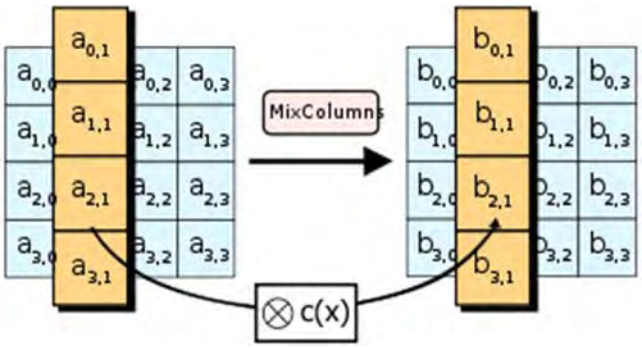


Figure 3 **The MixColumns step**

The AddRoundKey Step

In the AddRoundKey step, the subkey is combined with the state. For each round, a subkey is derived from the mainkey using Rijndael's key schedule; each subkey is the same size as the state. The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR.

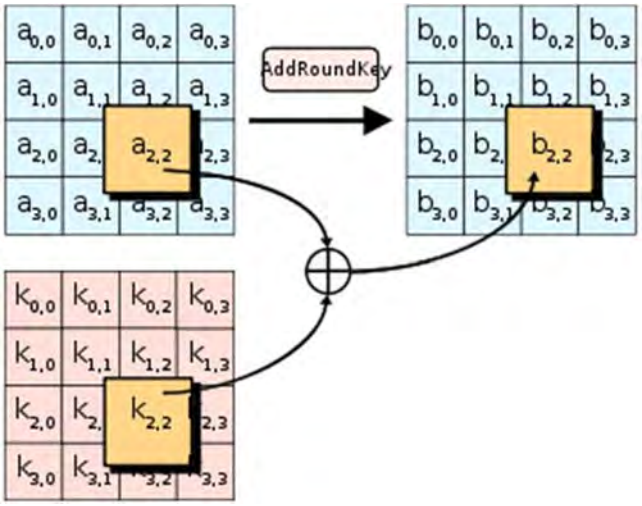


Figure 4 The AddRoundKey step

Optimization of the cipher

On systems with 32-bit or larger words, it is possible to speed up execution of this cipher by combining the SubBytes and ShiftRows steps with the MixColumns step by transforming them into a sequence of table lookups. This requires four 256-entry 32-bit tables, and utilizes a total of four kilobytes (4096 bytes) of memory - one kilobyte for each table. A round can then be done with 16 table lookups and 12 32-bit exclusive-or operations, followed by four 32-bit exclusive-or operations in the AddRoundKey step.

If the resulting four-kilobyte table size is too large for a given target platform, the table lookup operation can be performed with a single 256-entry 32-bit (i.e. 1 kilobyte) table by the use of circular rotates. Using a byte-oriented approach, it is possible to combine the SubBytes, ShiftRows, and MixColumns steps into a single round operation.

Security

Until May 2009, the only successful published attacks against the full AES were side-channel [68] attacks on some specific implementations. The National Security Agency (NSA) reviewed all the AES finalists, including Rijndael, and stated that all of them were secure enough for U.S. Government non-classified data. In June 2003, the U.S. Government announced that AES could be used to protect classified information:

The design and strength of all key lengths of the AES algorithm (i.e., 128, 192 and 256) are sufficient to protect classified information up to the SECRET level. TOP SECRET information will require use of either the 192 or 256 key lengths. The implementation of AES in products intended to protect national security systems and/or information must be reviewed and certified by NSA[72] prior to their acquisition and use.

AES has 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. By 2006, the best known attacks were on 7 rounds for 128-bit keys, 8 rounds for 192-bit keys, and 9 rounds for 256-bit keys.

Known attacks

For cryptographers, a cryptographic "break" is anything faster than a brute force—performing one trial decryption for each key. This includes results that are infeasible with current technology. The largest successful publicly known brute force attack against any block-cipher encryption was against a 64-bit RC5 key by distributed.net[69] in 2006.

AES has a fairly simple algebraic description. In 2002, a theoretical attack, termed the "XSL [1]attack", was announced by Nicolas Courtois and Josef Pieprzyk, purporting to show a weakness in the AES algorithm due to its simple description[70]. Since then, other papers have shown that the attack as originally presented is unworkable.

During the AES process, developers of competing algorithms wrote of Rijndael, "...we are concerned about [its] use...in security-critical applications."[73] However, in October 2000 at the end of the AES selection process, Bruce Schneier, a developer of the competing algorithm Twofish[2], wrote that while he thought successful academic attacks on Rijndael would be developed someday, he does not "believe that anyone will ever discover an attack that will allow someone to read Rijndael traffic."

A related-key attack[77] on the 192-bit and 256-bit versions of AES, discovered by Alex Biryukov and Dmitry Khovratovich[76], which exploits AES's somewhat simple key schedule and has a complexity of 2^{119}. In December 2009 it was improved to $2^{99.5}$. This is a follow-up to an attack discovered earlier in 2009 by Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolić, with a complexity of 2^{96} for one out of every 2^{35} keys [77]. However, related-key attacks are not of concern in any properly designed cryptographic protocol, as properly designed software will not use related-keys.

Another attack was released as a preprint by Bruce Schneier [79] on August 3, 2009. This new attack, by Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir, is against AES-256 that uses only two related keys and 2^{39} time to recover the complete 256-bit key of a 9-round version, or 2^{45} time for a 10-round version with a stronger

[1] In cryptography, the eXtended Sparse Linearization (XSL) attack is a method of cryptanalysis for block ciphers.
[2] In cryptography, Twofish is a symmetric key block cipher with a block size of 128 bits and key sizes up to 256 bits.

type of related subkey attack, or 2^{70} time for an 11-round version. 256-bit AES uses 14 rounds, so these attacks aren't effective against full AES.

In November 2009, the first known-key distinguishing attack against a reduced 8-round version of AES-128 was released as a preprint [80]. This known-key distinguishing attack is an improvement of the rebound or the start-from-the-middle attacks for AES-like permutations, which view two consecutive rounds of permutation as the application of a so-called Super-Sbox. It works on the 8-round version of AES-128, with a time complexity of 2^{48}, and a memory complexity of 2^{32}. 128-bit AES uses 10 rounds, so this attack isn't effective against full AES-128.

The first key-recovery attacks on full AES were due to Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger, and were published in 2011[82]. The attack is a biclique, a variant of the meet-in-the-middle (MITM) method of cryptanalysis, attack and is faster than brute force by a factor of about four. It requires $2^{126.1}$ operations to recover an AES-128 key. For AES-192 and AES-256, $2^{189.7}$ and $2^{254.4}$ operations are needed, respectively. This is a very small gain, as a 126-bit key (instead of 128-bits) would still take billions of years. Also, the authors calculate the best attack using their technique on AES with a 128 bit key requires storing 2^{88} bits of data. That works out to about 38 trillion terabytes of data, which is more than all the data stored on all the computers on the planet. As such this is a theoretical attack that has no practical implication on AES security[83].

According to the Snowden documents, NSA is doing research on whether a cryptographic attack based on tau, Kendall tau rank correlation coefficient statistics may help to break AES [84]. As for now, there are no known practical attacks that would allow anyone to read correctly implemented AES encrypted data.

Side-channel attacks

Side-channel attacks do not attack the underlying cipher, and thus are not related to security in that context. They rather attack implementations of the cipher on systems which inadvertently leak data. There are several such known attacks on certain implementations of AES.

In April 2005, D.J. Bernstein announced a cache-timing attack that he used to break a custom server that used OpenSSL's AES encryption[85]. The attack required over 200 million chosen plaintexts[86]. The custom server was designed to give out as much timing information as possible (the server reports back the number of machine cycles taken by the encryption operation); however, as Bernstein pointed out, "reducing the precision of the server's timestamps, or eliminating them from the server's responses, does not stop the attack: the client simply uses round-trip timings based on its local clock, and compensates for the increased noise by averaging over a larger number of samples."[87]

In October 2005, Dag Arne Osvik, Adi Shamir and Eran Tromer presented a paper demonstrating several cache-timing attacks against AES[88]. One attack was able to obtain an entire AES key after only 800 operations triggering encryptions, in a total of 65 milliseconds. This attack requires the attacker to be able to run programs on the same system or platform that is performing AES.

In December 2009 an attack on some hardware implementations was published that used differential fault analysis and allows recovery of a key with a complexity of 2^{32}[89]. Differential fault analysis is a type of side channel attack in the field of cryptography, specifically cryptanalysis. The principle is to induce faults and/or unexpected environmental conditions into cryptographic implementations, to reveal their internal states.

For example, a smartcard containing an embedded processor might be subjected to high temperature, unsupported supply voltage or current, excessively high overclocking, strong electric or magnetic fields, or even ionizing radiation to influence the operation of the processor. The processor may begin to output incorrect results due to physical data corruption, which may help a cryptanalyst deduce the instructions that the processor is running, or what its internal data state is.[1][2]

In November 2010 Endre Bangerter, David Gullasch and Stephan Krenn published a paper which described a practical approach to a "near real time" recovery of secret keys from AES-128 without the need for either cipher text or plaintext. The approach also works on AES-128 implementations that use compression tables, such as OpenSSL[90]. Like some earlier attacks this one requires the ability to run unprivileged code on the system performing the AES

encryption, which may be achieved by malware infection far more easily than commandeering the root account [91].

Cracking AES-256 Key Space

The security of this scheme relies on the strength of the advanced encryption standard. The following scenario shows the resistance of the AES to brute force attacks.

Just consider the following:

With each character having two possibilities (1 or 0), there are 2^{256} possible combinations. Typically, only 50% of these need to be exhausted to yield the correct key, so only 2^{255} need to be guessed. How long would it take to flip through each of the possible keys?

When doing routine, repetitive calculations (such as brute-forcing or bitcoin mining), the GPU is better suited than the CPU. A high-end GPU can typically do about 2 billion calculations per second (4 tera flops). So, we'll use GPUs.

Say you had a billion of these, all hooked together in a massively parallel computer system. Together, they could perform at $4*10^{21}$ flops, or $2*10^{21}$ keys per second.

Since there are 31,556,952 seconds in a year,

$2*10^{21}$*keys/second *31,556,952 seconds/year = 6.3113904*10^28 keys per year* (3)

Now if we divide 2^{255} combinations by $6.3113904*10^{25}$ keys per year:

$$2^{255} / 6.3113904*10^{25} = 9.1732631*10^{47} \text{ years} \quad (4)$$

The universe itself only existed for 14 billion ($1.4*10^{10}$) years. It would take ~$6.7*10^{40}$ times longer than the age of the universe to exhaust half of the keyspace of a AES-256 key.

Lets try our experiment while considering power. Most high-end GPUs take around 150 watts of energy to power them selves at full load. This doesn't include cooling systems. *1 billion gpus @ 150 watts = $1.5*10^{11}$ watts.* The largest nuclear power reactors (Kashiwazaki-Kariwa) generate about 1 gigawatt of energy.

Therefore, 1 billion GPUs would require *$1.5*10^{11}$ watts / 1 gigawatt = 150* nuclear power plant reactors to constantly power them, and it would still take longer than the age of the universe to exhaust half of a AES-256 keyspace.

One billion GPUs is kind of unrealistic. How about a supercomputer? The Tianhe-2 Supercomputer is the world's fastest supercomputer located at Sun Yat-sen University,

Guangzhou, China. It clocks in at around 34 petaflops. Applying the same logic, would require 10^{38} Tianhe-2 Supercomputers running for the entirety of the existence of everything to exhaust half of the keyspace of AES-256 key.

This shows that the content of files that are encrypted using AES can not be reveled by brute force attacks. As a result this scheme uses the AES to encrypt any file when applying de-duplication.

Chapter Four
Client Side De-duplication

Client-side deduplication enables remote clients to send data directly to the Storage Server. The client computer deduplicates data and then sends only the unique data directly to a deduplication storage folder. Client-side deduplication may reduce the CPU utilization on the server. On the client however CPU utilization may increase. Since The server is expected to service many clients, sharing some of the server load with client computers is a good example of load balancing.

Figure 5 an example of users trying to upload the same file.

Say user **Tesfa** stores a **file F** and **Michael** requests to store the same **file F.** Instead of storing a second copy of F, the server simply updates metadata associated to **F** to indicate that Tesfa and Michael both stored F. In this way, no file is duplicated on the server.

Figure 6 Example of file de-duplication

4.1 The Challenge of Client Side de-duplication

Figure 7 The Challenge: Encryption vs de-duplication

As it can be seen from the previous figure, it is difficult to apply deduplication on encrypted file. Just storing Tesfa's cipher text, for example, does not work because Michael cannot later decrypt it to recover the file, and vice-versa. De-duplication and encryption, to a great extent, conflicts with each other. De-duplication takes advantage of data similarity in order to achieve storage reduction. While cryptography makes cipher text indistinguishable from theoretically random data, i.e., encrypted data are always distributed randomly, so identical plaintext encrypted by randomly generated cryptographic keys will very likely have different cipher texts which cannot be de-duplicated.

4.2 Convergent Encryption

The conflict between de-duplication and encryption was first discovered by Douceur[26] *et al*. in the Farsite distributed file system [26]. To overcome the conflict, they used the convergent encryption, in which the hash of the data was used as the encryption key, to make identical plaintexts be encrypted to the same cipher text regardless of which user they belonged to. In convergent encryption (CE) where the key is derived deterministically from the file.

User **Tesfa** derives a key K1 = H(X) from his file X and then encrypts the message as C = E(K1;X) = E(H(X),X), where H is a cryptographic hash function and E is a block cipher. (They assume the message is one block long.) Since encryption is deterministic, if user **Michael** starts from the same message he would produce the same key and cipher text.

In this scheme when a user requests to upload a file by submitting the hash value of the file, if the hash is not found on the server, the cipher text is given to the server and the user retains K. The server, for subsequent users, can now manage duplication on the cipher text C, checking, when it receives the hash value of F, whether or not it is already stored. Both **Tesfa** and **Michael** can decrypt C since both have the same key K.

These ideas have been attractive enough to see significant usage. To mention some, Farsite, Permabit, Freenet, MojoNation, GNUnet and flud gained additional visibility in 2011 when cloud storage provider Bitcasa announced they were using convergent encryption to manage duplication of data in their cloud storage service.

Figure 8 Convergent Encryption

Although Convergent encryption [26] and its derivatives have gained visibility through many cloud storage providers, they all have security vulnerabilities. Below are a list of some attack scenarios that demonstrate drawbacks of current implementations of convergent encryption and its derivatives.

Usage of a common hash function: In this case an attacker can get hold of the hash values of files owned by other users. Accepting the hash value as a "proof" for the entire file, allows anyone who gets the hash value to download the entire file.

Learn The Remaining Information Attack: this type of attack applies to the encryption of documents that are only a slight variation of a public document. For example if defender encrypts a bank form including a ten digit bank account number, an attacker that is aware of generic bank form format may extract defender's bank account number by producing bank

form for all possible bank account numbers, encrypt them and then by comparing those encryptions with defender's encrypted file deduce his bank account number.

Using the storage service as an unintended CDN: In this scenario, user Tesfa puts a copy of a file(X) to the storage server and publishes the hash of X on his web page. Now anyone wishing to get the file can present the hash value to the service and to restore the file from the storage server. Hence Tesfa has used the storage service as a content distribution network (CDN).

Poison Attack (Target Collision Attack): As the cloud server is unable to verify if indeed the cipher text is the encryption of the file F by its hash value, a malicious user may substitute the valid cipher text C_F with an equal size poisoned file before uploading it to the cloud.

Divide and Conquer: In this case a file with many pages has sufficient min-entropy to the view of adversaries, however the first page has very low min entropy. If this file is stored in a cloud storage, which supports block-level duplication management. Then the adversary, through brute force search, could recover the unknown information in the first page.

Server Break-In: Consider an attacker that is able to temporarily access to its internal cache, which includes the hash values for all the recently accessed files. The attacker is now not only able to download all these files but also can publish these hash values on the internet, hence enabling anyone to get these files.

Malicious client software: The examination of D. Harnik and Pinkas [32] revealed that the client software of several file storage services, including Dropbox, stores local unencrypted manifest files, written in SQLite format. These files include an entry for each file stored by the service, which includes the hash value of the file. Malicious software can leak to Michael the hash value of this file itself. Michael can use this value to recover the manifest file, and then use the hash values stored in the manifest file to recover all files stored by Tesfa. The implication is that a leakage of a short hash value, which is either 20 or 32 byte long (depending on the service), leaks all of Tesfa's files, in a way which is essentially undetectable.

The convergent encryption gives one deterministic transform from a particular plaintext to the cipher text, which exposes more vulnerability.

Chapter Five
Analysis, Design and Implementation

5.1 Overview of proposed scheme

This scheme, **Secure** and **Efficient** Storage Method for Large Scale Servers, focuses on security (confidentiality) as well as efficiency (de-duplicated storage) of data. In order to achieve this goal, the scheme implements the proof of ownership protocol and the advanced encryption standard.

Here the first user and subsequent users are treated differently by the system. While any user trying to upload a file for the first time is able to store his data on the server, any subsequent user who claims to have the same file which is already stored on the server is required to provide evidence through the proof of ownership protocol.

Case I: First upload.

1. The user computes Hash of the blocks of the file by constructing a Merkle Tree and sends root hash value $\mathbf{H_{ROOT}}$ to the server.
2. As the server has no record of the hash value submitted by the user and hence understands that in fact this is the first upload; the server signals the user to continue.
3. Up on receiving the go command the client software computes the following and sends it to the server
    a. generates a random key **K**;
    b. computes the cipher-text $\mathbf{C_F}$ of the file **F** using the encryption key **K**;
    $\mathbf{C_F = Encrypt(F, K)}$
    c. computes the cipher-text $\mathbf{C_K}$ of the random key **K** using some summary of the file $\mathbf{F_S}$ in this scheme sha 256 was used.
    $\mathbf{C_K = Encrypt(K, F_S)}$
4. Up on receiving $\mathbf{C_F}$, and $\mathbf{C_K}$ the server computes Hash value of $\mathbf{C_F}$, H_{CF}.
5. $\mathbf{H_{ROOT}}$, $\mathbf{C_K}$ and $\mathbf{H_{CF}}$ will be stored in secure database. $\mathbf{C_F}$ will be stored as a file on disk (storage device of the server)

We can say that the server has no idea what the content of the file is; it has only its hash value, the cipher text and the key encrypted by summary of the file. This way the confidentiality of the file can be preserved.

Case II: Subsequent uploads.

1. The user computes Hash of the blocks of the file by constructing a Merkle Tree and sends root hash value $\mathbf{H_{ROOT}}$ to the server.
2. The server searches for the hash value in its lookup database; as the hash value already exists in the server database, the server starts the proof of ownership protocol with the user.
3. Upon receiving the proof of ownership command for a chosen block index B_i from the server, the client software computes the following and sends it to the server
    a. compute the hash value of the block B_i and its siblings $S_i \ldots S_n$ all the way to the root hash;
4. The server then verifies if the submitted values are correct; if so, it sends the cipher key $\mathbf{C_K}$ to the user.
5. Up on receiving the client performs the following
    a. The user decrypts the cipher key $\mathbf{C_K}$ using the file he/she claims to have.
    K= decrypt(C_K, File)
    b. Then the user encrypts the file using K.
    C_F = Encrypt(F, K)
    c. Then the user computes hash of C_F
    H_{CF} = hash(C_F)
6. Upon receiving H_{CF}, the server compares it with the value stored in the database.
    - If the two hash values are different, then with overwhelming high probability, either the first user has launched a poison attack on file F, or the subsequent user is cheating, or both. If the first user is honest, he/she can recover file F from the cloud, and present file F as a proof; if the subsequent is honest, he/she can present her local copy of F as a proof.

- If the two hash values match, the subsequent user may remove the local copy of file F and keep the AES key K safely in local storage. Hence he/she can always recover file F by downloading the cipher-text C_F from the cloud and decrypting it with key.

All the procedures that are described before can be summarized in the following flow chart.

Figure 9 Program flow of the proposed system

As it can be seen from the above flowchart, the proposed scheme has two major components:

- Storage Manager and
- Ownership Verifier

Although the complete source can be found on appendix I to appendix X, the overview of the system and its functionalities is described next.

5.1.1 Storage Manager

The store manager component makes all read and write operations on the server. It updates the database, writes cipher-text on server disk. The upload and the download operations are performed here. The data to be stored is divided into two parts as explained below.

i. Lookup Data

The file server maintains, in a secure small primary storage, a database of hash values of all received files, and looks up the value hash(F) and hash(C_F) in this database.

If there is no match found, then file F is not in the cloud storage yet. The client software will be required to upload C_F to the cloud storage, and the hash value hash(F) will be added into the look-up database.

If there is a match found, then file F is already in the cloud storage, uploaded by other users or even by the same user before. In this case, no uploading of file F from client's computer to the cloud storage is required; as a result network bandwidth will be saved, and the cloud server will allow the user to access the file F in its cloud storage.

The hash value hash(F) and hash(C_F) serve as an index of file F and C_F respectively, used by the cloud server to locate information of C_F among a huge number of files.

ii. Encrypted File

C_F, which may be a large cipher-text, is stored separately in a potentially insecure but large secondary storage. Although this data is kept in relatively insecure storage, the confidentiality will remain assured as the information is already encrypted.

Even if the server is compromised by external adversary, the contents of the file will not be revealed.

5.1.2 Proof of Ownership Verifier

This component uses Merkle tree. A Merkle ree provides a succinct "commitment" to a large buffer, such that it is later possible to "open" and verify individual blocks of the buffer without giving the entire buffer. To construct a Merkle tree we split the input buffer into blocks, then group the blocks in pairs and use a secure hash algorithm to hash each pair. The hash values are then again grouped in pairs and each pair is further hashed, and this process is repeated until only a single hash value remains. This results in a binary tree with the leaves corresponding to the blocks of the input buffer and the root corresponding to the last remaining hash value. (When the number of blocks in the input buffer is 2^h, the resulting tree is a height-h complete binary tree.)

This proof of ownership component lets a client efficiently prove to the server that the client holds a file, rather than just some short information about it. This solution is based on Merkle trees and specific encodings and uses the following basic proof protocol, between a verifier that has the root value of a Merkle tree $MT_{root}(X)$ (and the number of leaves in the tree), and a prover who claims to know the underlying buffer X. The verifier simply chooses some number of leaf indexes and asks the prover for the value of the corresponding leaves. The prover replies with these leaves and with a sibling path for each one of them and the verifier accepts if all these sibling paths are valid.

At this stage the user's request to download/store his/her file on server is performed only if the user provides a valid proof that confirms that he/she owns the file.

Proof-of-ownership is a protocol in two parts between two players on a joint input F (which is the input file). The user and the server engage in an interactive protocol in which the client has F and the server has H_{Root} from previous uploads, at the end of which the server either accepts or rejects.

Hence a proof-of- ownership is could be randomized and takes the input file F and a security parameter, and an interactive two-party protocol (between client and server).

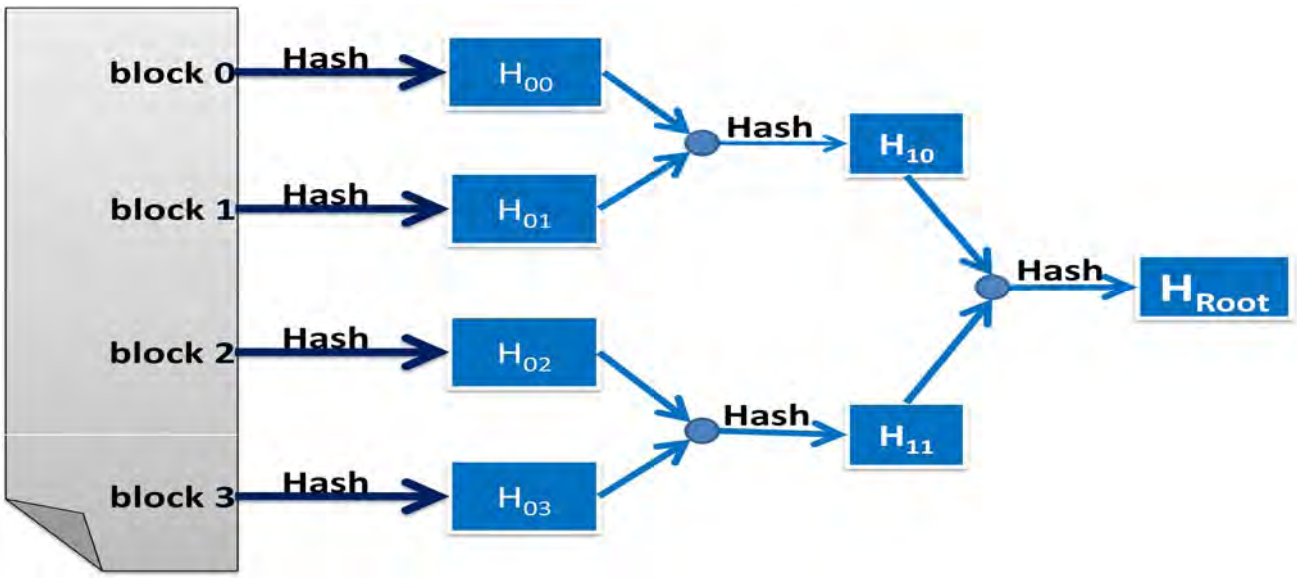


Figure 10 Example of merkle hash tree with four blocks

This proof protocol is conducted between a verifier (server) that has the root value of a Merkle tree H_{Root} (and the number of leaves in the tree, in this example number of leaves = 4), and a user who claims to know the underlying buffer X. The verifier simply chooses some number of leaf indexes and ask the user for the value of the corresponding leaves. The user replies with these leaves and with a sibling path for each one of them, and the verifier accepts if all these sibling paths are valid. Below we use the Merkle-tree protocol (with respect to hash function h) where the verifier knows the root value H_{Root} and number of leaves (4 in this example), and chooses a random leaf number and asks for all the paths of the leaf all the way to the root including the paths of the siblings.

Example: A file with four blocks.

The server has already the root value $\mathbf{H_{Root}}$ and the number of leafs (**4**) from previous upload. the proof of ownership protocol is done in the following manner:

1. A new user requests to store a file and provides $\mathbf{H_{Root}}$ as root hash value;
2. the server chooses a random block index and requests the user for its hash value;
3. up on receiving the request, the user computes the hash value of the index and all its siblings all the way to the root hash value and sends it to the server.

If, for example, the server requests for hash value of index 1, then the user replies with the hash values of H_{01}, H_{00}, H_{10}, H_{11} and H.

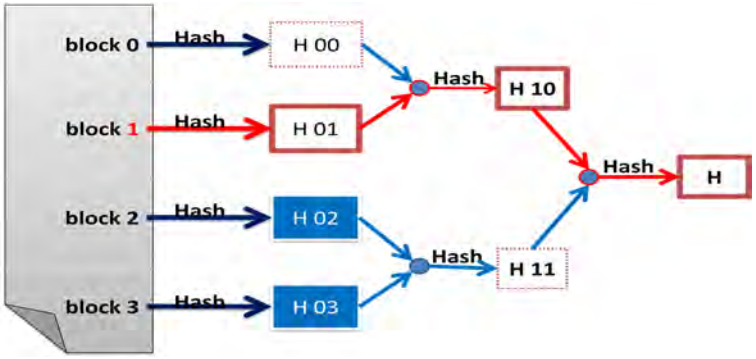


Figure 11 Merkle Tree and Proof of Ownership

1. Server requests proof at index 1;
2. Client replies:

   Hash of all Paths of H01and path of all of its siblings all the way to the root hash H.

Path of H01 and all of its siblings	
Path	Sibling paths
H_{01}	H_{00}
H_{10}	H_{11}
H	

5.2. Implementation of the system

The System has two sides:

1. Server side

2. Client side

Technology

I have chosen the Microsoft visual studio 2013 to develop the system. This technology requires the dot net 4.5 framework be installed on both the server and the client machine. Particularly, I have used the windows communication foundation, WCF, to design the server. Furthermore, The windows presentation foundation, WPF, was used for the implementation of the client component;

5.2.1. The Server Side Application

Server Data Model

The server side application contains a database that maintains users and file pointers to the actual file location. As it can be seen below the **user table** keeps records of registered users along with encrypted passwords and password salts. The **FileData table** keeps records of file pointers to the actual storage location and a summary of the actual file that acts as index for fast access. Specifically, let us see the purpose of all columns in the FileData table.

Column	Function
FileID	A unique ID that represents the actual file
CipherID	A unique ID that represents the encrypted file obtained by computing the hash value of the encrypted file using a random key.
CipherKey	A random key encrypted by summary of the actual file.
PhysicalLocation	A pointer to the encrypted file location on disk.

Table 2 Functions of FileData table columns.

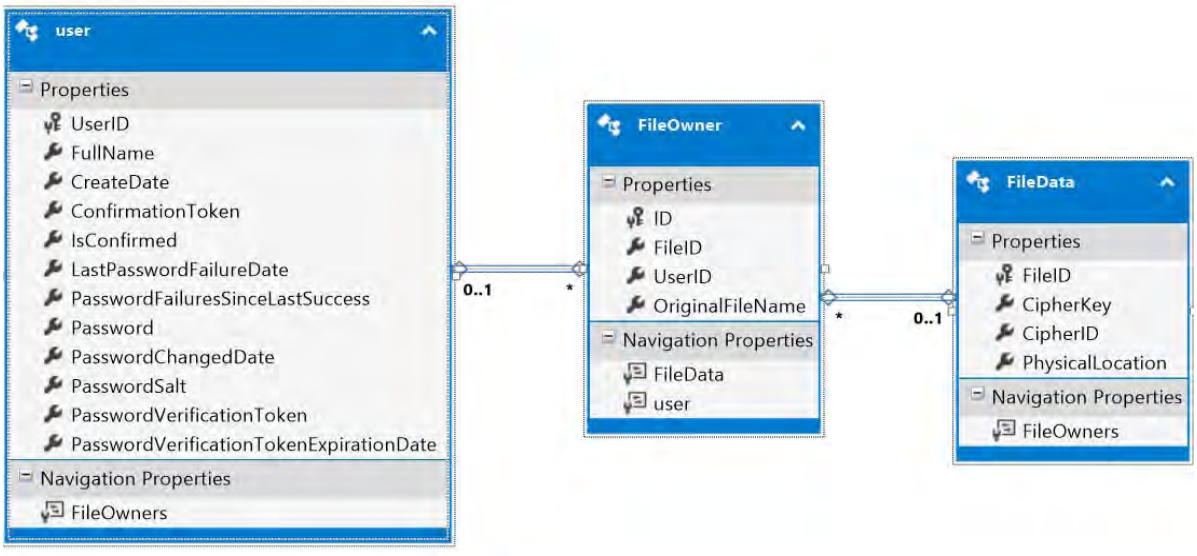


Figure 12 Server Side Data Model

The **FileOwner table** acts as a link between the user and the file pointer to the actual file location.

Server-Side Component

The server listens client requests on three end points, each of which use hypertext transport protocol .

1. **Authentication Service** listens on port 8080 at http:/ServerName:8080/Authenticator

2. **Ownership Verifier** listens on port 8081 at http:/ServerName:8081/Protocol

3. **File Transfer Service** listens on port 8082 at http:/ServerName:8080/FileTransfer

The Server Name is name of the server machine where the service is running.

The server side component is responsible for authenticating users and their files. This component particularly manages the storage device and the database server. While the database helps for storing users and the link to the actual file location, the storage media stores the actual file.

Each service has its own security requirement. For example if we look at the Authentication service users will submit their credientials through the communication channel, the security level at this stage should be very high. As a result the authentication service has its own dedicated port which uses secure communication protocol.

The Protocol service besides having secure communication protocol, it requires continious conversation between the users and the server. Further more it manges user sessions and activities. Hence granting a dedicated pott to this service simplifies programming.

The File transfer service is there to facilitate the transfer of files between client and server. No secure channel is mandatory here. Since the files are ecrypted, it safe to uploiad/download files via insecure channels. Hence this service has its own dedicated port with less security requirement in its communication channel.

Figure 13 Services of the Server Component

The **authenticator** which listens on port 8080 provides the following services:

1. User Registration Service

2. User Login Service

I a user is authenticated by the server, he/she can send his/her proof through the **ownership verifier** at port 8081. If the proof of ownership protocol that is conducted between client and server succeeds, users can upload and download their files using the **file transfer** service on port 8082.

The authentication service implements the ILogIn interface which acts as a service contract between the server and the client. Besides, the authentication service exposes the **Register** method where clients can remotely call to register their credentials and the **LogIn** method which is used by users to log in to the server. The **ListUserFiles** method is automatically invoked upon successful log in. The method returns a list of filenames that are uploaded by the user.

A **dataContext** field which is a reference to the sql-server database tables is maintained along side the **dataSource** field which keeps track of user activities. The **LogIn** method returns a true or false value that indicates success or error. If user name is already taken, the **Register** method returns taken otherwise returns either a success or error.

The complete definition of the ILogIn interface is shown below.

```
[ServiceContract]
  public interface ILogIn
  {
    [OperationContract]
    RegistrationState  Register(UserInfo USER);
    [OperationContract]
    bool LogIn(string username, string password);
    [OperationContract]
    FilesVM ListUserFiles(ListRequest request);
  }
```

Listing 1: ILogIn interface

The following class diagram shows the Authentication Service and the implementation of the ILogIn interface on the server side. The functions of the methods listed in listing 1 are shown below.

LogIn Method: allows a user to login id he/she provides valid credentials.

Register Method: Helps users register their user names and passwords.

ListUserFiles: Lists filenames of a given user.

The input parameter, **UserInfo**, is a class with the following data members:

```
public partial class UserInfo
  {
    public string FullName;
    public string Password;
    public string UserName;
  }
```

Listing 2: UserInfo class

When the client invokes the Register method by providing his information (UserInfo), a RegistrationState will be returned which indicates if the registration was successful.

The return type **RegistrationState** is an enumeration type which looks like the following.

```
  public enum  RegistrationState: int
  {
    Registered = 0,
    Taken = 1,
    Error = 2,
  }
```

Listing 3: RegistrationState **enum**

Whenever a new user requests and for registration and provides his user name and password, the server will first check if the user name is already taken. If so the return value will be **RegistrationState.Taken**. Similarly it will return **RegistrationState.Registered** if the task of registration is completed successfully or **RegistrationState.Error** if an error has occurred.
The sequence diagram that shows the registration process shown in the next diagram

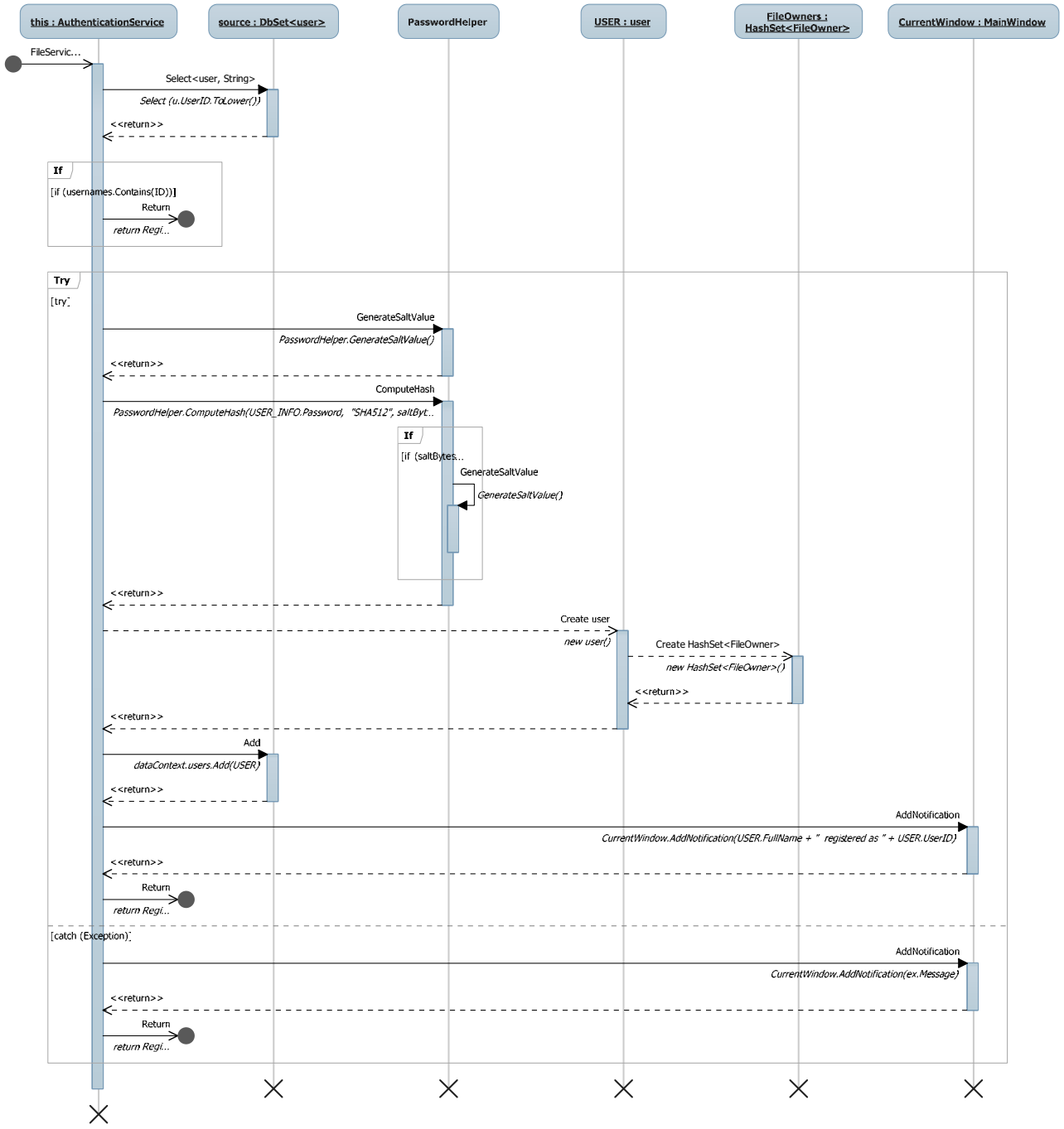


Figure 14 Sequence Diagram of User Registration

Figure 15 Flow Chart of User Registration

Figure 18 shows the program flow of user registration. When a tries to register his/her user name, the algorithm searches for the user name in the database server. If it is already it is found, then TAKEN will be returned. If the user name is not in the database, the user name will be added to the database server. Hence SUCCESS or ERROR will be returned based up on the success of the add operation.

Figure 16 Flow Chart of Login method

Figure 19 show the procedures of the login method. If the username is in database and the input password matches with the stored password in the database, then a new session will be created for the user and authentication will be reported successful.

Error will be returned if the user name is not found in database or if the input password does match with the stored password.

Figure 17 Flow Chart of ListUserFiles Method

As it is shown in figure 20, the ListUserFiles method is designed to return the list of file names that are uploaded by the user. This method first checks if user name is in database and if the user is logged in. This can be done by simply checking if the user has a session created for him/her. If so, it implies that the user has been successfully logged in. The method then updates the session of the user and returns the list of file names that the user has uploaded and the pointers of the actual file location.

Protocol Service

As it can be seen on figure 21, the **protocol** service implements the **IProtocol** interface which defines three methods where the client invokes during initiation of file uploading. All the three methods are invoked in sequence.

The user first invokes the request upload method. If request is accepted by the server, then the client will invoke the send proof method. The server verifies the proof and depending on the outcome of the verification, the client may invoke the send cipher hash method if the proof was correct.

The definition of the IProtocol interface which is implemented by the protocol service is shown below.

```
[ServiceContract]
public interface IProtocol
{
    [OperationContract]
    UploadRequestReply RequestUpload(UploadRequest request);

    [OperationContract]
    UploadRequestReply SendProof(ProofList proof);

    [OperationContract]
    OwnershipRequestReply SendCipherHash(OwnershipRequest CipherHash);
}
```

Listing 4: The IProtocol Interface

During the upload operation, the first command that a user invokes, after successful login, is the request upload method. As it can be seen from figure 23, the server will either grant the request or ask the user for a proof if he/she really posses the file.

Figure 18 Proof of Ownership Protocol

If the proof that users has sent is verified, the server will send the cipher key to the user. If the user really possesses the file he/she will be able to decrypt the cipher key and encrypt the file using the key. The user then sends the hash value of the encrypted file. Having received the cipher hash, the server will compare it with the stored hash value and if the two values match, the user will be granted the privilege to access the services provided by the file transfer service.

File Transfer Service

The **file transfer** service contains the UploadFile and DownloadFile methods. The client invokes these methods to start uploading or downloading files. The definiation of the **IFileTransfer** interface which is implemented by the file transfer service is described below.

```
[ServiceContract]
public interface IFileTransfer
{
    [OperationContract]
    UploadReply UploadFile(UploadInfo request);
    [OperationContract]
    RemoteFileInfo DownloadFile(DownloadRequest request);
}
```

Listing 5: IFileTransfer Interface

The client invokes UploadFile method to upload files and the DownloadFile method to download a file. During this operation, the database is updated to maintain the users and list of files that user owns.

The flow chart of uploading a file is shown on figure 26. As it can clearly be seen from the chart, the upload methods always checks if the user is logged in and has a valid session created for him/her. Once it is verified that the user has a session, the server updates the stage of the user's session to upload started.

The server generates its own new unique GUID (globally unique identifier) as a file name and writes the uploaded bytes to the file. Once the server writes all bytes of the file on a disk, it updates the database to reflect the file owners and file locations.

Figure 19 Flowchart of Upload Method

5.2.2 The Client Component

The client component is created using visual studio 2013. I have used MVVM (Model-View-ViewModel) design pattern. This design pattern is mostly known for separation and encapsulation of responsibilities and it was selected for this scheme as it makes coding easy. The windows presentation foundation is suitable for this design patter as compared to windows forms.

A well-designed application is an application that is easy to develop, test, maintain, and evolve. Creating such an application typically involves some form of separation and encapsulation of responsibilities. A common approach involves separation of the UI (the views) from the business logic (the model). In fact, that approach is so common that there are a number of stock solutions, collectively known as the model-view design patterns:

- Model-View-Controller (MVC) is the grand daddy of them all, dating back to 1979.

  In MVC, the Controller creates the Views and the Model. The Views register with the Controller, which also has a reference to the Model. A View notifies the Controller of user interactions, and the Controller in turn queries the View for its state and updates the Model appropriately. The Controller notifies the Views when there are changes in the Model, and the Views query the controller to update their states.

- Model-View-ViewModel (MVVM) is a derivative of MVC that takes advantage of particular strengths of the Windows Presentation Foundation (WPF) architecture to separate the Model and the View by introducing an abstract layer between them: a “Model of the View,” or ViewModel. The origins of this pattern are obscure. It was adapted for WPF use by the Expression team (Microsoft's employees) as they developed version 1 of Blend (Microsoft Blend for Visual Studio).

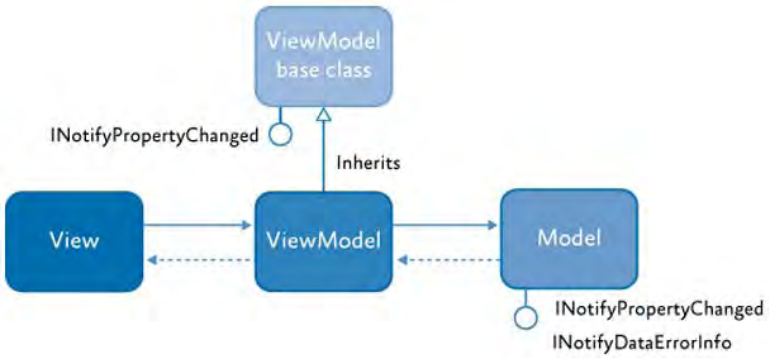


Figure 20 MVVM design Pattern

In MVVM, the View and the ViewModel are typically instantiated by the container application. The View keeps a reference to the ViewModel. The ViewModel exposes commands and observable entities ("bindables") that the View binds to (or to put it in programming terms, view.DataContext = viewModel;). User interactions with the View trigger commands on the ViewModel, and updates in the ViewModel are propagated to the View through data binding.

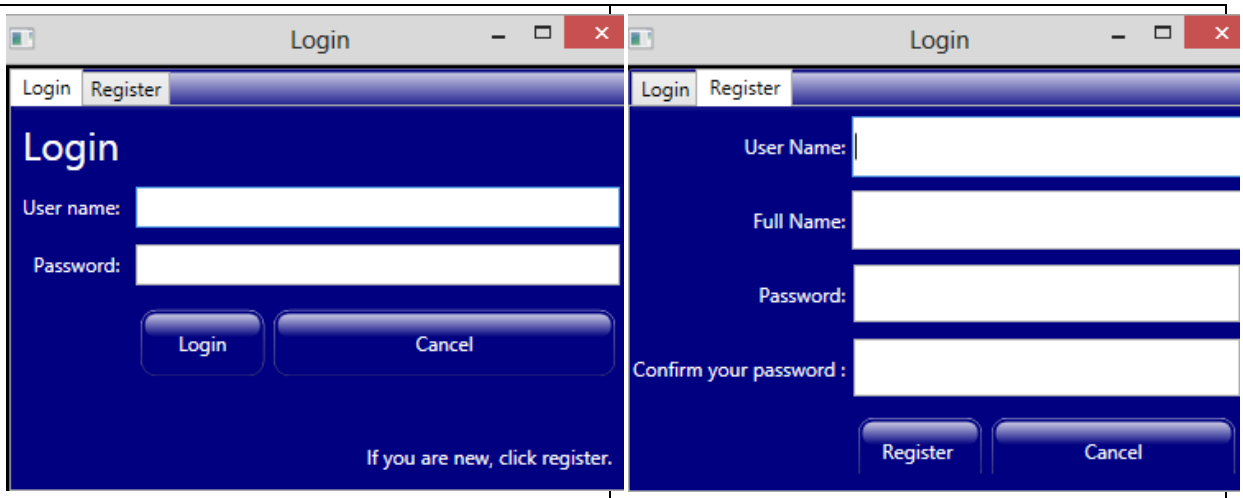


Figure 21 Login/Register Window

Figure 28 shows the first thing that a user sees when trying to access the server. New users can register their credentials while registered users can log in to the server through this window.

Once a user has successfully logged in, the main windows of the client application is displayed. As you can see from figure 29, the user can browse for files and click upload command to start uploading.

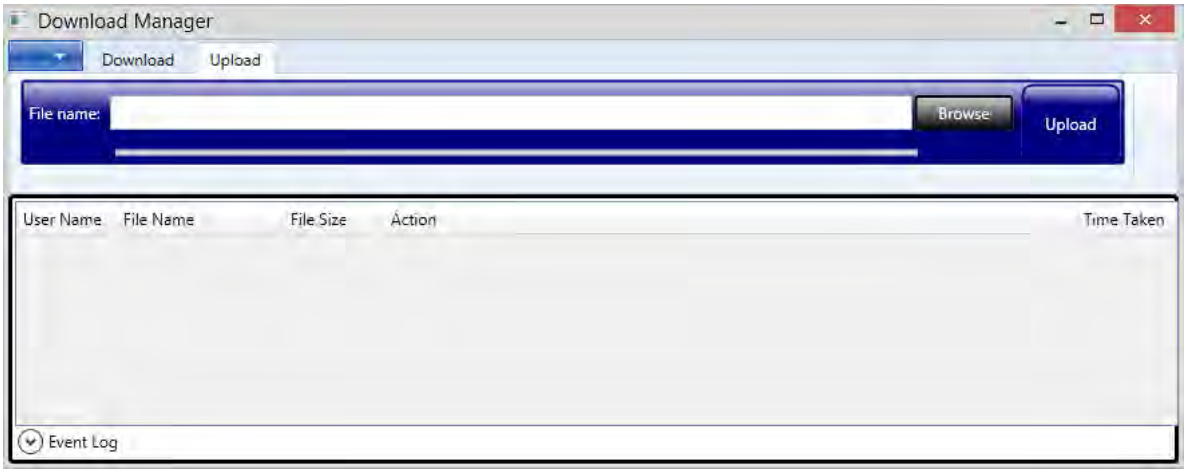


Figure 22 Main window when upload tab is selected

Similarly, figure 30 shows when a user clicks the download tab. the list of files that the user has uploaded to the server are also displayed in order to help the user select the file he/she wants to download.

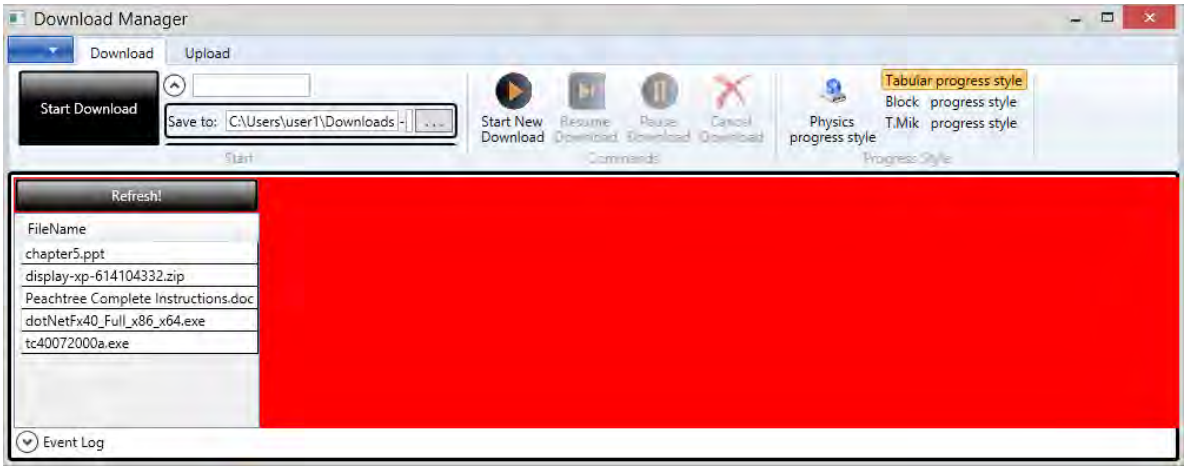


Figure 23 Main window when download tab is selected

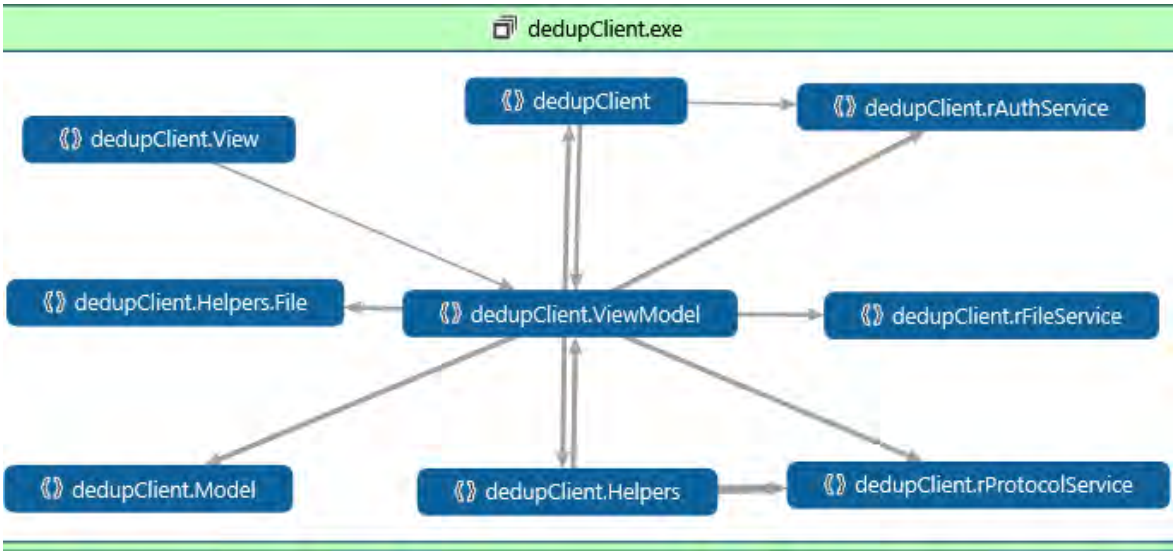


Figure 24 Code-Map of the de-duplication client application

Figure 31 clearly shows that the application follows the MVVM design pattern since all classes are directly connected with the ViewModel other than with each other.

Experiment Setup

The experiment was conducted on the following machines.

	Client Machine	Server Machine
Operating System	Windows 8	Windows Server 2012
Database Server		SQL Server 2008
Manufacturer	Toshiba	Dell
Model	Satellite S955	Optiplex 3010
Processor	Intel Core i5-3317U CPU @1.7GHz 1.7GHz	Intel Core i3 3220 3.3 GHz
RAM	8 GB	2GB
Hard Disk	1 TB	500GB
System Type	64 bit Operating system, x64-based processor	64 bit Operating system, x64-based processor

Table 3 Specification of the experiment machines

Communication Medium

The server and client machine are connected via category 6 UTP cable. The reported data rate of the local area network is 100 Mbps.

Experiment Data

The data used for this experiment was collected from electronic library of the college of engineering and technology of the Jigjiga University. The college comprises of the following departments:

- Electrical and Computer Engineering
- Civil Engineering
- Hydraulics Engineering
- Water Resources and Irrigation Engineering
- Surveying Engineering
- Construction Technology and Management

- Computer Science
- Information Technology
- Software Engineering
- Mechanical Engineering

Experiment Parameters

As you can see below, the download parameter was deliberately ignored as it cannot be optimized. Any user who later wants get his file must download it from the server at a rate of the network speed. There is no other way out. Uploading however can be optimized and I will try to demonstrate how. Below are the lists of parameters that are related to the upload operation.

S/N	Operation	First Upload	Subsequent Uploads	Network Usage
1.	Compute Hash Value	Yes	Yes	No
2.	Encrypt File	Yes	Yes	No
3.	Send Upload-Request	Yes	Yes	Yes
4.	Upload File and receive success/fail reply	Yes	No	Yes
5.	Send Proof of Ownership and receive success/fail reply	No	Yes	Yes

Table 4 Experiment Parameters

It can be noted that activities 1 and 2 are client side operations and do not consume any network band width. Hence, if we are to analyze the benefits of file de-duplication with regard to network bandwidth, the only thing that I should record is the time delay incurred for activities 3, 4 and 5. Since activity 3 is always called by both the first and subsequent users, only activities 4 and 5 would show a difference for the first upload and subsequent uploads.

Hence, only the elapsed times of activities 4 and 5 are measured.

Chapter Six
Result and Discussion

The e-library of the department of electrical and computer engineering with respect to the remaining departments was analyzed and the following results were found.

The total size of these files is 26.88GB out of which only 21GB is unique to the department. The remaining 5.88GB was common to most departments. These common files have consumed a total of 14.28GB of disk space across other departments. As a result 8.4 GB of disk space was wasted.

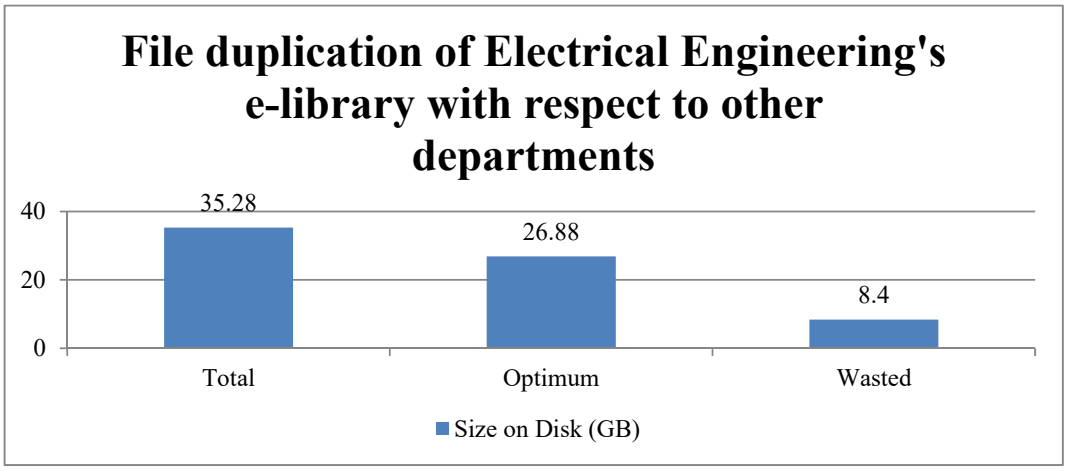


Figure 25 Duplication of Electrical Engineering files with respect to other disciplines

As it can be seen from figure 32 8.4GB which is 31.25% of the optimum size has been wasted.

Similarly, analyzing files of all departments at the same time has produced the following results.

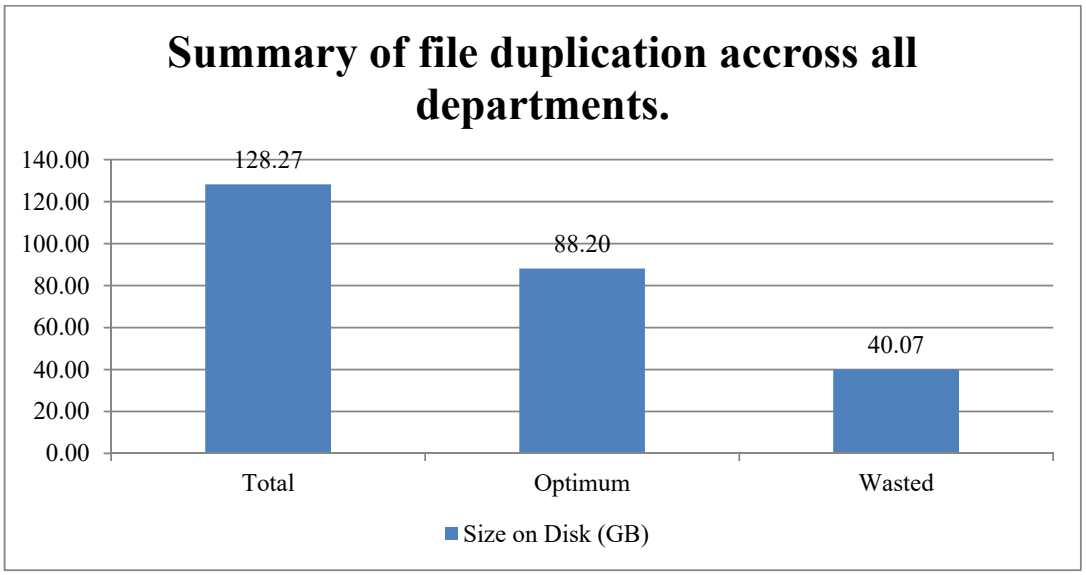


Figure 26 Summary of File duplication across all departments

Overall, 46.15% of duplication was found across all departments. And the results show that applying de-duplication could have saved 40.7 GB of disk space.

No more explanation is required here to convince the advantages of de-duplication with regard to storage efficiency. If we look at the case large scale servers for example cloud storage servers, the size of disk space consumed and the logistics required to run the service is enormous. Imagine the benefits of applying de-duplication in a world where 75% of its digital data are duplicate copies [63].

The next advantage of file de-duplication is reducing the network trafic, reduced transmision time and reducing the server load. In order to verify this claim the following experiment was conducted.

Network usage and server load

File Size	Elapsed Time in Milliseconds		Ratio
	First Upload	Next Upload	First upload/Next Upload
3MB	75.5	12.7	6.0
4.5MB	137.8	15.0	9.2
9MB	225.0	15.0	15.0
49MB	1250.6	19.2	65.0
71MB	1682.3	22.0	76.4

Table 5 Ellapsed time of first upload vs subsequent uploads

As it can be seen from the table above, elapsed time of the first upload increases as the file increases. The time needed to complete next uploads shows very little variation as the size of the file increases.

The difference in elapsed time between the first upload and subsequent uploads can be clearly shown in the following graph.

As it can be seen from Table 6, it is easy to conclude that network trafic can be minimized if de-duplication is applied. For example, let us see the case of 49 Mb file, it has taken 1250.6 milliseconds to complete the upload. However since the next upload request needs only to provide proof, it has taken only 19.2 milliseconds. In this case while the first user uploads at approximately 0.04 Mb per second, subsequent users can complete the same task at a rate of 2.55 Mb per second. This is 6275% increase in performance. Similarly 7975% increase in performance is observed when trying to upload the 71Mb file. The results clearly show that as the size of the file grows using client side de-duplication improves the response time, reduces the network trafic, reduces the transmission time and reduces the server load.

Hence it can be concluded that applying de-duplication improves storage efficiency, minimizes server load and increases network band width.

What about the security aspect of the scheme? Is it securing enough? The possible security threats are discussed below.

Case 1: Server Gets Hacked

Let us assume the server is hacked by external adversary, the actual file is will be accessible to the hacker. However this hacker cannot read the contents of the file as the file is encrypted using Advanced Encryption Algorithm.

The next possibility will be to get the shared key from the database server. Yes the hacker will get it but the key is also encrypted. She/he cannot decrypt the key. This key can only be decrypted only by those who share the file not even by the service provider knows it.

Hence the confidentiality of the file will be preserved.

Case 2: Malicious User.

The outside adversary may obtain some knowledge (e.g. a hash value) of the file of interest via some channels, and plays a role of cloud user to interact with the cloud server.

The proof of ownership protocol uses a binary tree of hash values that make up the root hash. Having knowing the root hash value, it is computationally infeasible find the hash value of the child nodes. This can be due to the fact that hash functions are one way functions.

Case 3: Semi-honest Inside Adversary (Honest-but-Curious Cloud Server).

This could capture at least the following case in real world applications, some technical employee or even the owner of the server tries to access user data due to some reason.

This curious server has no role in generating the encryption key. This shared key is generated by the user. Furthermore the user will always encrypt this key before uploading it to the server so that only those who really posses the file can decrypt it. Hence the confidentiality of the data will be preserved.

Hence it can be concluded that the designed system is **efficient** and **secure method** which can be used by large scale servers. The global leaders in cloud storage as shown in the table below.

SN	Storage service Provider	Number of Users	client side de-duplication
1	DropBox	Over 300 Million	Yes
2	OneDrive(formerly Sky Drive)	Over 250 Million	No
3	Google Drive	Over 190 Million	No

Table 6 Worldwide leaders in cloud storage

Although DropBox implements client side de-duplication, the encryption key is generated by DropBox itself with the consequence that any intruder who gets the key (as well as Dropbox employees) could decrypt any file if they had access to Dropbox's backend storage infrastructure. Even a complaint [62] was filed with the Federal Trade Commission in May 2011 alleging that the popular Dropbox file-sharing service has been misleading users about the security and privacy of their files.

In short, if all big storage service providers implement **secure** and **efficient** de-duplication scheme, the internet will not be over loaded and the services that are provided by these companies will be faster. Because the more the users are the more the more possibility of finding identical files.

Although Google performs server side de-duplication, it is not applicable if users encrypt their files before uploading it to the server as identical files will then appear to be random. Furthermore server side de-duplication does not reduce network trafic and transmission time.

IDC [63] data shows that nearly 75% of our digital world is a copy – in other words, only 25% is unique. Hence applying de-duplication on large scale servers saves can save up to 75% of global storage space. As a result not only huge amount of storage can be optimized, but also the global network traffic can be reduced.

Chapter Seven
Conclusion and Recommendation

I have studied the problem of providing secure outsourced storage that both supports de-duplication and resists brute-force attacks. I design a system, Efficient **and Secure Storage Method** that uses Advanced Encryption Standard with the ability for the client to generate a true random key which is shared amongst a group of clients. The clients interact with the server by a protocol ensuring that the server can request proof of while learning nothing about files stored by clients.

These mechanisms ensure that this scheme provides strong security against external attacks which compromise the server and communication channels. Should the server be compromised, the attacker must still attempt an offline brute-force attack. The substantial increase in security comes at a modest price in terms of performance, and a small increase in storage requirements relative to the base system.

This scheme is much more secure than convergent encryption and its derivatives however this acomplished at a relatively small increment in storage requirements and performance cost.

This scheme is easy to deploy, it can work transparently on top of any server implementing a simple storage interface.

Bibliography

[1] Bitcasa, inifinite storage. "http://www.bitcasa.com" April 2014

[2] Ciphertite data backup. "http://www.ciphertite.com" April 2014

[3] Dropbox, a file-storage and sharing service. "http://www.dropbox.com" April 2014

[4] The Flud backup system. "http://flud.org/wiki/Architecture" April 2014

[5] GNUnet, "a framework for secure peer-to-peer networking" https://gnunet.org/

[6] Google Drive. "http://drive.google.com" April 2014

[7] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. "Farsite: Federated, available, and reliable storage for an incompletely trusted environment" ACM SIGOPS Operating Systems Review, 36(SI):1 14, 2002.

[8] P. Anderson and L. Zhang. "Fast and secure laptop backups with encrypted de-duplication". In Proc. of USENIX LISA, 2010.

[9] Z. Bar-Yossef, O. Reingold, R. Shaltiel, and L. Trevisan. "Streaming computation of combinatorial objects" In Computational Complexity, 2002. Proceedings. 17th IEEE Annual Conference on, pages 133 142. IEEE, 2002.

[10] C. Batten, K. Barr, A. Saraf, and S. Trepetin. p "Store: A secure peer-to-peer backup system" Unpublished report, MIT Laboratory for Computer Science, 2001.

[11] M. Bellare, A. Boldyreva, and A. O'Neill. "Deterministic and efficiently searchable encryption" In A. Menezes, editor, CRYPTO 2007, volume 4622 of LNCS, pages 535 552. Springer, Aug. 2007.

[12] M. Bellare, Z. Brakerski, M. Naor, T. Ristenpart, G. Segev, H. Shacham, and S. Yilek. "Hedged public-key encryption: How to protect against bad randomness" In M. Matsui, editor, ASIACRYPT 2009, volume 5912 of LNCS, pages 232 249. Springer, Dec. 2009.

[13] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. "A concrete security treatment of symmetric encryption" In 38th FOCS, pages 394 403. IEEE Computer Society Press, Oct. 1997.

[14] M. Bellare, M. Fischlin, A. O'Neill, and T. Ristenpart. "Deterministic encryption: Definitional equivalences and constructions without random oracles" In D. Wagner, editor, CRYPTO 2008, volume 5157 of LNCS, pages 360 378. Springer, Aug. 2008.

[15] M. Bellare, O. Goldreich, and S. Goldwasser. "Incremental cryptography: The case of hashing and signing" In Y. Desmedt, editor, CRYPTO'94, volume 839 of LNCS, pages 216 233. Springer, Aug. 1994.

[16] M. Bellare and P. Rogaway. "Random oracles are practical: A paradigm for designing e ficient protocols" In V. Ashby, editor, ACM CCS 93, pages 62-73. ACM Press, Nov. 1993.

[17] M. Bellare and P. Rogaway. "The security of triple encryption and a framework for code-based gameplaying proofs" In S. Vaudenay, editor, EUROCRYPT 2006, volume 4004 of LNCS, pages 409-426. Springer, May / June 2006.

[18 M. Blum, P. Feldman, and S. Micali. "Non-interactive zero-knowledge and its applications" In Proceedings of the twentieth annual ACM symposium on Theory of computing, pages 103-112. ACM, 1988.

[19] A. Boldyreva, S. Fehr, and A. O'Neill. "On notions of security for deterministic encryption, and e ficient constructions without random oracles" In D. Wagner, editor, CRYPTO 2008, volume 5157 of LNCS, pages 335-359. Springer, Aug. 2008.

[20] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. "Public key encryption with keyword search" In C. Cachin and J. Camenisch, editors, EUROCRYPT 2004, volume 3027 of LNCS, pages 506-522. Springer, May 2004.

[21] Z. Brakerski and G. Segev. "Better security for deterministic public-key encryption: The auxiliary-input setting" In P. Rogaway, editor, CRYPTO 2011, volume 6841 of LNCS, pages 543-560. Springer, Aug. 2011.

[22] J. Cooley, C. Taylor, and A. Peacock. "ABS: the apportioned backup system" MIT Laboratory for Computer Science, 2004.

[23] L. P. Cox, C. D. Murray, and B. D. Noble. "Pastiche: making backup cheap and easy" SIGOPS Oper. Syst. Rev., 36:285 298, Dec. 2002.

[24] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith. "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data" SIAM Journal on Computing, 38(1):97 139, 2008.

[25] Y. Dodis, T. Ristenpart, and T. Shrimpton. "Salvaging Merkle-Damg fiard for practical applications" In A. Joux, editor, EUROCRYPT 2009, volume 5479 of LNCS, pages 371 388. Springer, Apr. 2009.

[26] J. Douceur, A. Adya, W. Bolosky, D. Simon, and M. Theimer. "Reclaiming space from duplicate files in a serverless distributed file system" In Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on, pages 617 624. IEEE, 2002.

[27] B. Fuller, A. O'Neill, and L. Reyzin. "A uni fied approach to deterministic encryption: New constructions and a connection to computational entropy" In R. Cramer, editor, TCC 2012, volume 7194 of LNCS, pages 582 599.Springer, Mar. 2012.

[28] S. Goldwasser and S. Micali. Probabilistic encryption. "Journal of Computer and System Sciences" 28(2):270-299, 1984.

[29] V. Goyal, A. O'Neill, and V. Rao. "Correlated-input secure hash functions" In Y. Ishai, editor, TCC 2011, volume 6597 of LNCS, pages 182-200. Springer, Mar. 2011.

[30] S. Gueron. Advanced encryption standard (AES) instructions set. Intel Corporation, 25, 2008.

[31] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. "Proofs of ownership in remote storage systems" In Y. Chen, G. Danezis, and V. Shmatikov, editors, ACM CCS 11, pages 491-500. ACM Press, Oct. 2011.

[32] D. Harnik, B. Pinkas, and A. Shulman-Peleg. "Side channels in cloud services: Deduplication in cloud storage" Security & Privacy, IEEE, 8(6):40-47, 2010.

[33] J. H fiastad, R. Impagliazzo, L. A. Levin, and M. Luby. "A pseudorandom generator from any one-way function". SIAM Journal on Computing, 28(4):1364-1396, 1999.

[34] J. Kilian and P. Rogaway. "How to protect DES against exhaustive key search (an analysis of DESX)" Journal of Cryptology, 14(1):17-35, 2001.

[35] M. Killijian, L. Court fies, D. Powell, et al. "A survey of cooperative backup mechanisms" XRDS, 2006.

[36] C.J. Lu. "Hyper-encryption against space-bounded adversaries from on-line strong extractors" In M. Yung, editor, CRYPTO 2002, volume 2442 of LNCS, pages 257-271. Springer, Aug. 2002.

[37] L. Marques and C. Costa. "Secure de-duplication on mobile devices" In Proceedings of the 2011 Workshop on Open Source and Design of Communication, pages 19-26. ACM, 2011.

[38] I. Mironov, O. Pandey, O. Reingold, and G. Segev. "Incremental deterministic public-key encryption" In D. Pointcheval and T. Johansson, editors, EUROCRYPT 2012, volume 7237 of LNCS, pages 628-644. Springer,Apr. 2012.

[39] N. Nisan and D. Zuckerman. "Randomness is linear in space" Journal of Computer and System Sciences, 52(1):43-52, 1996.

[40] J. Pettitt. Content based hashing. "http://cypherpunks.venona.com/date/1996/02/msg02013.html" Sep. 2015

[41] A. Rahumed, H. Chen, Y. Tang, P. Lee, and J. Lui. "A secure cloud backup system with assured deletion and version control" In Parallel Processing Workshops (ICPPW), 2011 40th International Conference on, pages 160-167. IEEE, 2011.

[42] T. Ristenpart, H. Shacham, and T. Shrimpton. "Careful with composition: Limitations of the indi fierentiability framework" In K. G. Paterson, editor, EUROCRYPT 2011, volume 6632 of LNCS, pages 487-506. Springer, May 2011.

[43] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. "OCB: A block-cipher mode of operation for efficient authenticated encryption" In ACM CCS 01, pages 196-205. ACM Press, Nov. 2001.

[44] D. X. Song, D. Wagner, and A. Perrig. "Practical techniques for searches on encrypted data" In 2000 IEEE Symposium on Security and Privacy, pages 44-55. IEEE Computer Society Press, May 2000.

[45] M. Storer, K. Greenan, D. Long, and E. Miller. "Secure data de-duplication" In Proceedings of the 4th ACM international workshop on Storage security and survivability, pages 1-10. ACM, 2008.

[46] S. P. Vadhan. "On constructing locally computable extractors and cryptosystems in the bounded storage model" In D. Boneh, editor, CRYPTO 2003, volume 2729 of LNCS, pages 61-77. Springer, Aug. 2003.

[47] D. Wichs. "Barriers in cryptography with weak, correlated and leaky sources" In Proceedings of the 4th conference on Innovations in Theoretical Computer Science, ITCS '13, pages 111-126, New York, NY, USA,2013. ACM.

[48] Z. Wilcox-O'Hearn. "Convergent encryption reconsidered" 2011. http://www.mail-archive.com/cryptography@metzdowd.com/msg08949.html.

[49] Z. Wilcox-O'Hearn and B. Warner. "Tahoe: The least-authority file system" In Proceedings of the 4th ACM international workshop on Storage security and survivability, pages 21-26. ACM, 2008.

[50] J. Xu, E.-C. Chang, and J. Zhou. "Leakage-resilient client-side de-duplication of encrypted data in cloud storage" Cryptology ePrint Archive, Report 2011/538, 2011. http://eprint.iacr.org/.

[51] Wee Kenong Ng, Yonggang Wen, and Huafei Zhu. "Private data de-duplication protocol"In SAC 12': ACM Symposium on Applied Computing, 2012.

[52] SpiderOak-Blog. "Why SpiderOak doesn't de-duplicate data across users" September 2014.

[53] Roberto Di Pietro and Alessandro Sorniotti. "Boosting Efficiency and Security in Proof of Ownership for eduplication" In ASIACCS '12:ACM Symposium on Information, Computer and Communications Security (Full Paper), 2012.

[54] Mihir Bellare, Sriram Keelveedhi and Thomas Ristenpart "DupLESS: Server-Aided Encryption for Deduplicated Storage" 22nd USENIX Security Symposium, 2013

[55] Rogaway, P., AND Shrimpton, T. "A provable-security treatment of the key-wrap problem" In EUROCRYPT 2006 (St. Petersburg, Russia,May 28 – June 1, 2006), S. Vaudenay, Ed., vol. 4004 of LNCS, Springer, Berlin, Germany, pp. 373–390.

[56] Goh, E., Shacham, H., Modadugu, N., AND Boneh, D. Sirius "Securing remote un trusted storage" NDSS.

[57] Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., AND Fu, K. Plutus "Scalable secure file sharing on un trusted storage" In Proceedings of the 2nd USENIX Conference on File and Storage Technologies (2003), pp. 29–42.

[58] Kamara, S., Papamanthou, C., AND Roeder, T. Cs2: "A searchable cryptographic in cloud storage system" Tech. rep., Technical Report MSR-TR-2011-58, Microsoft, 2011.

[59] LI, J., Krohn, M., Mazi`ERES, D., AND Shasha, D. "Secure un trusted data repository (SUNDR)" Defense Technical Information Center, 2003.

[60] Wang, W., LI, Z., Owens, R., AND Bhargava, B. "Secure and efficient access to outsourced data" In Proceedings of the 2009 ACM workshop on Cloud computing security (2009), ACM,pp. 55–66.

[61] NIST. "Information Technology Laboratory-National Institute of Standards and Technology" Gaithersburg, MD 20899-8900. March 2012

[62] Mathew J. Schwartz. "InformationWeek: Dropbox Accused Of Misleading Customers On Security. www.darkreading.com" May 17, 2011.

[63] John Gantz and David Reinsel. "The Digital Universe Decade – Are You Ready?" May 2010.

[64] National Institute of Standards and Technology (NIST). "Federal Information Processing Standards Publications (FIPS PUB 197)" November 26, 2001

[65] Diffie, Whitfield, Hellman, Martin E. "Exhaustive Cryptanalysis of the NBS Data Encryption Standard" June 1977.

[66] Wikipedia. "http://en.wikipedia.org/wiki/United_States_Secretary_of_Commerce" June 2013.

[67] ISO/IEC 18033-3:2010. "Security techniques -- Encryption algorithms" 2013

[68] Wikipedia. "http://en.wikipedia.org/wiki/Side_channel_attack" June 2014.

[69] Ou, George. "Is encryption really crackable?" Ziff-Davis. April 30, 2006.

[70] Bruce Schneier. "AES News, Crypto-Gram Newsletter" September 15, 2002.

[71] Niels Ferguson, Richard Schroeppel, Doug Whiting (2001). "Proceedings of Selected Areas in Cryptography" 2001.

[72] Lynn Hathaway. "National Policy on the Use of the Advanced Encryption Standard (AES) to Protect National Security Systems and National Security Information" June 2003.

[73] "Lecture Notes in Computer Science (PDF/PostScript)" Springer-Verlag. pp. 103–111. November 2006.

[74] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson. "The Twofish Encryption Algorithm (PDF/PostScript)" June 1998.

[75] Niels Ferguson. "Impossible differentials in Twofish (PDF)" October 1999.

[76] Biryukov, Alex, Khovratovich, Dmitry (2009-12-04). "Related-key Cryptanalysis of the Full AES-192 and AES-256" March 2010.

[77] Nikolić, Ivica. "Distinguisher and Related-Key Attack on the Full AES-256. Advances in Cryptology – CRYPTO 2009. Springer Berlin / Heidelberg" pp. 231–249. doi:10.1007/978-3-642-03356-8_14. ISBN 978-3-642-03355-1.

[78] Bruce Schneier. "Another New AES Attack. Schneier on Security, A blog covering security and security technology" August 2009

[79] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich and Adi Shamir. "Key Recovery Attacks of Practical Complexity on AES Variants With Up To 10 Rounds" September 2009.

[80] Henri Gilbert,Thomas Peyrin. "Super-Sbox Cryptanalysis: Improved Attacks for AES-like permutations" November 2009.

[81] Vincent Rijmen. "Practical-Titled Attack on AES-128 Using Chosen-Text Relations" 2010.

[82] Vincent Rijmen. "Practical-Titled Attack on AES-128 Using Chosen-Text Relations" 2010.

[83] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. "Biclique Cryptanalysis of the Full AES" 2011.

[83] 1Password, Knox, security. "AES Encryption isn't cracked." December 2014.

[84] Snowden documents. "http://www.spiegel.de/international/germany/inside-the-nsa-s-war-on-internet-security-a-1010361.html" April 2014.

[85] "Index of formal scientific papers" Cr.yp.to; November 2008.

[86] Bruce Schneier. "AES Timing Attack" February 2007.

[87] "Index of formal scientific papers" Cr.yp.to. November 2008.

[88] Dag Arne Osvik, Adi Shamir and Eran Tromer. "Cache Attacks and Countermeasures: the Case of AES" Nov 2005.

[89] Dhiman Saha, Debdeep Mukhopadhyay and Dipanwita RoyChowdhury. "A Diagonal Fault Attack on the Advanced Encryption Standard" December 2009.

[90] Endre Bangerter, David Gullasch and Stephan Krenn. Cache Games "Bringing Access-Based Cache Attacks on AES to Practice" 2010.

[91] "Breaking AES-128 in real-time, no cipher-text required" Hacker News. News.ycombinator.com.December 2012

Submitted by

Student: sig. _____ date: _____

_____

Approved by

Advisor: sig. _____ date: _____

_____

Chairman, Dept's Graduate Committee: sig. _____ date: _____

_____

Chairman, College's Graduate Commission: sig. _____ date: _____

Dean, Graduate School: sig. _____ date: _____

_____

Appendix I - Source code of server UI

```
public partial class MainWindow : Window
{
   private UsersVM dataSource =new UsersVM();
   public MainWindow()
   {
      InitializeComponent();
      this.DataContext = dataSource;
   }
   ServiceHost FileServiceHost;
   ServiceHost AuthenticationServiceHost;
   ServiceHost ProtocolServiceHost;
   private void Start(object sender, RoutedEventArgs e)
   {
      try
      {
         AuthenticationServiceHost = new ServiceHost(typeof(AuthenticationService));
         AuthenticationServiceHost.Open();
         foreach (Uri address in AuthenticationServiceHost.BaseAddresses
                  lstActivity.Items.Add("Authentication Service Listening on: " + address);

         FileServiceHost = new ServiceHost(typeof(FileTransferService));
         FileServiceHost.Open();

         foreach (Uri address in FileServiceHost.BaseAddresses)
            lstActivity.Items.Add("Fle Service Listening on: " + address);

         ProtocolServiceHost = new ServiceHost(typeof(ProtocolService));
         ProtocolServiceHost.Open();

         foreach (Uri address in ProtocolServiceHost.BaseAddresses)
            lstActivity.Items.Add("Protocol Service Listening on: " + address);
      }
      catch (Exception ex)
      {
         log.Text += "Service Error:\n" + ex.Message;
         Debug.Print(ex.Message);
      }
```

```
    }
    private void Stop(object sender, RoutedEventArgs e)
    {
      try
      {
        if (AuthenticationServiceHost != null)
        {
          AuthenticationServiceHost.Close();
          lstActivity.Items.Add("Authentication Service Stoped.");
        }
        if (FileServiceHost != null)
        {
          FileServiceHost.Close();
          lstActivity.Items.Add("File Service Stoped.");
        }
        if (ProtocolServiceHost != null)
        {
          ProtocolServiceHost.Close();
          lstActivity.Items.Add("File Service Stoped.");
        }
      }
      catch (Exception ex)
      {
        log.Text+="Error while stopping service: \n" + ex.Message;
        Debug.Print(ex.Message);
      }
    }
    public void AddNotification(string notification)
    {
      lstActivity.Items.Add(notification);
    }
    public UsersVM DataSource
    {
      get
      {
        return dataSource;
      }
    }
  }
```

Appendix II
Source code of the authentication service of server component implementation

```
[ServiceContract]
public interface ILogIn
{
    [OperationContract]
    bool LogIn(string username, string password);
    [OperationContract]
    RegistrationState  Register(UserInfo USER);
    [OperationContract]
    FilesVM ListUserFiles(ListRequest request);
}

public class UserInfo
{
    public string FullName;
    public string Password;
    public string UserName;
}
public enum  RegistrationState: int
{
    Registered = 0,
    Taken = 1,
    Error = 2,
}
 [MessageContract]
public class ListRequest
{
    [MessageBodyMember]
    public string UserID;
}
[MessageContract]
public class FilesVM
{
    [MessageHeader]
    public string FullName { get; set; }
    [MessageBodyMember]
    public List<FileInfoVM> Files { get; set; }
}
```

```
public class UsersVM:ViewModel
{
    dedupEntities dataContext = new dedupEntities();
    private ObservableCollection<ProtocolVM> protocolSession;
    public ObservableCollection<ProtocolVM> ProtocolSession
    {
        get
        {
            if (protocolSession == null)
                protocolSession = new ObservableCollection<ProtocolVM>();
            return protocolSession;
        }
        set
        {
            protocolSession = value;
            NotifyPropertyChanged("ProtocolSession");
        }
    }
}
[MessageContract]
public class FileInfoVM
{
    public string FileID { get; set; }
    public string CipherID { get; set; }
    public string OriginalFileName { get; set; }
    public string PhysicalLocation { get; set; }
}
public partial class AuthenticationService : ILogIn
{
    dedupEntities dataContext;
    MainWindow CurrentWindow;
    private UsersVM dataSource;
    public AuthenticationService()
    {
        dataSource = ((MainWindow)Application.Current.MainWindow).DataSource;
        CurrentWindow = (MainWindow)Application.Current.MainWindow;
        dataContext = new dedupEntities();
    }


bool ILogIn.LogIn(string username, string password)
    {
```

```
List<string> loggedUsers = dataSource.ProtocolSession.Select(u => u.UserID.ToLower()).ToList();

if (!(loggedUsers==null) && loggedUsers.Contains(username))
{
    CurrentWindow.AddNotification(username + " is requested for signing in again.");
}
else
{
   CurrentWindow.AddNotification("user log in " + username);
}

ProtocolVM session = new ProtocolVM();
session.UserID = username;
session.Stage = Stage.LogInRequest;
dataSource.ProtocolSession.Add(session);
user tmp = dataContext.users.Where((u) =>   u.UserID.ToLower() == username.ToLower()).FirstOrDefault();

if (tmp != null)
{
   if (PasswordHelper.VerifyHash(password, "SHA512", tmp.Password, tmp.PasswordSalt))
   {
      CurrentWindow.AddNotification("user signed in: " + username);
      session.Stage = Stage.LogedIn;
      return true;
   }
   CurrentWindow.AddNotification("user log in failed for " + username);
   session.Stage = Stage.LogInFailed;
   return false;
}
CurrentWindow.AddNotification("user log in failed for " + username);
session.Stage = Stage.LogInFailed;
return false;
}

RegistrationState  ILogIn.Register(UserInfo USER_INFO)
{
   MainWindow CurrentWindow = (MainWindow)Application.Current.MainWindow;
   string ID = USER_INFO.UserName.ToLower();
   List<string> usernames = dataContext.users.Select(u => u.UserID.ToLower()).ToList();
   if (usernames.Contains(ID))
   {
      return RegistrationState.Taken ;
   }
   try
```

```
        {
            byte[] saltBytes = PasswordHelper.GenerateSaltValue();
            string saltText = Convert.ToBase64String(saltBytes);
            string hashPassword = PasswordHelper.ComputeHash(USER_INFO.Password, "SHA512", saltBytes);

            user USER = new user();

            USER.UserID = ID;
            USER.FullName = USER_INFO.FullName;
            USER.PasswordSalt = saltText;
            USER.Password = hashPassword;
            USER.CreateDate = DateTime.Now;
            USER.PasswordChangedDate = DateTime.Now;

            dataContext.users.Add(USER);
            dataContext.SaveChanges();
            CurrentWindow.AddNotification(USER.FullName + "  registered as " + USER.UserID);
            return RegistrationState.Registered ;
        }
        catch (Exception ex)
        {
            CurrentWindow.AddNotification(ex.Message);
            return RegistrationState.Error ;
        }
    }

    public FilesVM ListUserFiles(ListRequest request)
    {
        user u = dataContext.users.Where(x => x.UserID == request.UserID).FirstOrDefault();
        if (u==null)
        {
            Debug.Print("null user for " + request.UserID);
            return null;
        }
        FilesVM file = new FilesVM()
        {
            FullName = u.FullName
        };

        file.Files = new List<FileInfoVM>
                (
                    u.FileOwners.Select
                    (f =>
                        new FileInfoVM()
```

```
                    {
                      CipherID = f.FileData.CipherID,
                      FileID = f.FileID,
                      OriginalFileName = f.OriginalFileName,
                      PhysicalLocation = f.FileData.PhysicalLocation
                    }
                  )
                );

      if (file  == null)
        return null;
      else
        return file;
    }
  }
```

Appendix III source code of the implementation of the proof service

```
[ServiceContract]
public interface IProtocol
{
    [OperationContract]
    UploadRequestReply RequestUpload(UploadRequest request);
    [OperationContract]
    UploadRequestReply SendProof(ProofList proof);
    [OperationContract]
    OwnershipRequestReply SendCipherHash(OwnershipRequest CipherHash);
}

[MessageContract]
public class UploadRequest
{
    [MessageHeader]
    public string username { get; set; }
    [MessageBodyMember]
    public string FileHash { get; set; }
}
[MessageContract]
public class UploadRequestReply
{
    [MessageHeader]
    public bool UploadGranted;
    [MessageHeader]
    public int ProofIndex;
    [MessageHeader]
    public string Message;
}
[MessageContract]
public class ProofList
{
    [MessageHeader]
    public string UserName;
    [MessageHeader]
```

```
    public string FileName;
    [MessageBodyMember]
    public List<HashItem> Proof;
}


public class HashItem
{
    public HashItem()
    {
        _Children = new List<HashItem>();
    }

    public HashItem(string id)
    {
        _ID = id;
        _Children = new List<HashItem>();
    }

    private string _ID;
    private string _PID;
    private List<HashItem> _Children;

    public string ID
    {
        get
        {
            return _ID;
        }
        set
        {
            _ID = value;
        }
    }
    public string PID
    {
        get
        {
            return _PID;
```

```
        }
        set
        {
            _PID = value;
        }
    }
    public List<HashItem> Children
    {
        get { return _Children; }
        set
        {
            _Children = value ;
        }
    }
}

[MessageContract]
public class OwnershipRequest
{
    [MessageHeader]
    public string username { get; set; }
    [MessageHeader]
    public string FileName { get; set; }

    [MessageBodyMember]
    public string FileHash { get; set; }
}
[MessageContract]
public class OwnershipRequestReply
{

    [MessageHeader]
    public string Message;
}
public partial class ProtocolService : IProtocol
{
    private UsersVM dataSource;
    dedupEntities dataContext;
    MainWindow CurrentWindow;
```

```
        public ProtocolService()
        {
            dataSource = ((MainWindow)Application.Current.MainWindow).DataSource;
            CurrentWindow = (MainWindow)Application.Current.MainWindow;
            dataContext = new dedupEntities();
        }
#region Tree
        private List<HashItem> shashItems;
        private void BindTree()
        {
            if (CurrentWindow.NavigationTree != null)
            {
                Debug.Print("ServerSide ..... List<HashItem> l");
                List<HashItem> l;
                Debug.Print("Adding Anccestors ...");
                l = shashItems.Where(t => string.IsNullOrWhiteSpace(t.PID)).ToList();
                if (l != null && l.Count > 0)
                {
                    CurrentWindow.NavigationTree.Items.Clear();
                    foreach (HashItem x in l)
                    {
                        TreeViewItem tvi = new TreeViewItem()
                        {
                            Header = x.ID,
                            Tag = x.PID
                        };
                        tvi.Expanded += OnChildExpand;
                        tvi.Items.Add(new object());
                        CurrentWindow.NavigationTree.Items.Add(tvi);
                    }
                }
            }
            else
            {
                Debug.Print("NavigationTree == null");
            }
        }
        private void OnChildExpand(object obj, RoutedEventArgs e)
        {
```

```
        TreeViewItem node = e.OriginalSource as TreeViewItem;
        if (node != null)
        {
            ExpandTree(node);
        }
    }

    private void ExpandTree(TreeViewItem node)
    {
        node.Items.Clear();
        List<HashItem> l;
        string ID = node.Header.ToString().ToLower();
        Debug.Print("Adding Children of " + ID.ToUpper() + " ....");
        l = shashItems.Where(x => !string.IsNullOrWhiteSpace(x.PID)).Where(t => (t.PID.ToLower() ==
(string)ID)).ToList();
        if (l != null && l.Count > 0)
        {
            Debug.Print(l.Count.ToString() + " Items found ....");
            foreach (HashItem x in l)
            {
                Debug.Print("            Item: " + x.ID);
                TreeViewItem tvi = new TreeViewItem()
                {
                    Header = x.ID,
                    Tag = x.PID
                };
                tvi.Expanded += OnChildExpand;
                tvi.Items.Add(new object());
                node.Items.Add(tvi);
            }
        }
        else
            Debug.Print("Not Children found");
    }

    private bool ProofVerfied(string fileID)
    {
        List<HashItem> l;
        l = shashItems.Where(t => string.IsNullOrWhiteSpace(t.PID)).ToList();
```

```
        string ID;
        if (l != null && l.Count == 1)
        {
          if (l.Count==1)
          {
            HashItem item = l[0];
            ID = item.ID;
            List<HashItem> children;
            children = shashItems.Where(x => !string.IsNullOrWhiteSpace(x.PID)).Where(t => (t.PID ==
(string)ID)).ToList();

            if (children.Count != 2)
            {
              CurrentWindow.AddNotification("Null children.");
              return false;
            }

            HashItem leftChild = children[0];
            HashItem rightChild = children[1];

            CurrentWindow.AddNotification(leftChild.ID);
            CurrentWindow.AddNotification(rightChild.ID);

            byte[] leftBytes = Convert.FromBase64String(leftChild.ID);
            byte[] rightBytes = Convert.FromBase64String(rightChild.ID);

            Helpers.MerkleHash mh = new MerkleHash();

            string newID = Convert.ToBase64String(mh.GetIntermideateHash(leftBytes, rightBytes));

            CurrentWindow.AddNotification(fileID);
            CurrentWindow.AddNotification(newID);
            CurrentWindow.AddNotification(ID);
            return (newID == fileID);
          }
        }
        CurrentWindow.AddNotification("Null tree");
        return false;
      }
```

```
        #endregion

        #region IProtocol Members

        UploadRequestReply IProtocol.RequestUpload(UploadRequest request)
        {
            List<string> sessionUsers = dataSource.ProtocolSession.Select(u => u.UserID).ToList();
            UploadRequestReply tmp = new UploadRequestReply();

            if (!sessionUsers.Contains(request.username))
            {
                tmp.UploadGranted = false;
                tmp.ProofIndex = -1;
                return tmp;
            }

            ProtocolVM currentSession = dataSource.ProtocolSession.Where(s => s.UserID ==
request.username).FirstOrDefault();
            if (currentSession == null)
            {
                tmp.UploadGranted = false;
                tmp.ProofIndex = -1;
                return tmp;
            }

            string fileID = request.FileHash;
            FileData fileInfo;
            fileInfo = dataContext.FileDatas.Where((file) => file.FileID.ToLower() ==
fileID.ToLower()).FirstOrDefault();

            if (fileInfo == null)
            {
                CurrentWindow.AddNotification("Server grannted upload right!\nFileID added to Session of user: " +
fileID);
                tmp.UploadGranted = true;
                currentSession.Stage = Stage.UploadGranted;
                currentSession.FileHash = fileID;
                return tmp;
```

```
        }
        else
        {
            int sec = DateTime.Now.Second % 7;
            currentSession.Stage = Stage.ProofRequested;
            currentSession.FileHash = fileID;
            CurrentWindow.AddNotification("Server Needs prooft!\nFileID added to Session of user: " + fileID);

            tmp.UploadGranted = false;
            tmp.ProofIndex = sec;
            return tmp;

        }

    }

UploadRequestReply IProtocol.SendProof(ProofList request)
    {
        List<string> sessionUsers = dataSource.ProtocolSession.Select(u => u.UserID).ToList();
        UploadRequestReply tmp = new UploadRequestReply();
        tmp.Message = "";
        if (!sessionUsers.Contains(request.UserName))
        {
            tmp.UploadGranted = false;
            tmp.ProofIndex = -1;
            return tmp;
        }

        ProtocolVM currentSession = dataSource.ProtocolSession.Where(s => s.UserID ==
request.UserName).FirstOrDefault();
        if (currentSession == null)
        {
            tmp.UploadGranted = false;
            tmp.ProofIndex = -1;
            return tmp;
        }
        currentSession.Stage = Stage.ProofSent;

        Debug.Print("ServerSide .....received List<HashItem> = " + request.Proof.Count);
```

```
            shashItems = request.Proof;
            string fileID = currentSession.FileHash;
            if (!ProofVerfied(fileID))
            {
                tmp.UploadGranted = false;
                tmp.ProofIndex = -1;
                tmp.Message = "Error";
                CurrentWindow.AddNotification("mismatch in hash");
                return tmp;
            }
            currentSession.Stage = Stage.OwnershipVerified;
            CurrentWindow.AddNotification("OwnershipVerified!");
            user USER = dataContext.users.Where((u) => u.UserID.ToLower() ==
request.UserName.ToLower()).FirstOrDefault();
            if (USER != null)
            {
                List<string> fileowners = dataContext.FileOwners.Where(u => u.FileID
==fileID).Select(s=>s.UserID).ToList();
                if (fileowners.Contains(USER.UserID))
                {
                    tmp.UploadGranted = false;
                    tmp.ProofIndex = -1;
                    tmp.Message = "File already uploaded.";
                    CurrentWindow.AddNotification("user already owns this file");
                    return tmp;
                }
                FileData fileData = dataContext.FileDatas.Find(fileID);
                tmp.UploadGranted = true;
                tmp.Message = fileData.CipherKey;
                return tmp;
            }

            tmp.UploadGranted = false;
            tmp.ProofIndex = -1;
            return tmp;
        }

        public OwnershipRequestReply SendCipherHash(OwnershipRequest request)
        {
```

```
        List<string> sessionUsers = dataSource.ProtocolSession.Select(u => u.UserID).ToList();
        OwnershipRequestReply tmp = new OwnershipRequestReply();
        tmp.Message = "Error!";

        user USER = dataContext.users.Find(request.username);

        Debug.Print("In SendCipherHash");
        if (USER == null)
        {
           Debug.Print("USER == null");
           return tmp;
        }

        if (!sessionUsers.Contains(request.username))
        {
           Debug.Print("USER not Session");
           return tmp;
        }

        ProtocolVM currentSession = dataSource.ProtocolSession.Where(s => s.UserID ==
request.username).FirstOrDefault();
        if (currentSession == null)
        {
           Debug.Print("Session == null");
           return tmp;
        }

        if (currentSession.Stage == Stage.OwnershipVerified)
        {
           currentSession.Stage = Stage.CipherHashSent;
           string fileID = currentSession.FileHash;

           FileData fileData = dataContext.FileDatas.Find(fileID);
           if (fileData == null)
           {
              Debug.Print("FileID == null");
              return tmp;
           }
           if (fileData.CipherID == request.FileHash)
```

```
            {
                FileOwner fileOwner = new FileOwner()
                {
                    FileID = fileID,
                    UserID = request.username,
                    OriginalFileName = request.FileName
                };

                dataContext.FileOwners.Add(fileOwner);
                dataContext.SaveChanges();
                currentSession.Stage = Stage.ProofVerified;
                Debug.Print("Success!");
                tmp.Message = "Done!";
                return tmp;
            }
            return tmp;
        }
        Debug.Print("Unknown error");
        return tmp;
    }

    #endregion
}
```

Appendix IV source code of the implementation of File Transfer Service

```
[ServiceContract]
public interface IFileTransfer
{
    [OperationContract]
    UploadReply UploadFile(UploadInfo request);
    [OperationContract]
    RemoteFileInfo DownloadFile(DownloadRequest request);
}
[MessageContract]
public class UploadInfo : IDisposable
{
   [MessageHeader(MustUnderstand = true)]
   public string UserName;
   [MessageHeader(MustUnderstand = true)]
   public string CipherHash;
   [MessageHeader(MustUnderstand = true)]
   public string CipherKey;
   [MessageHeader(MustUnderstand = true)]
   public string FileName;
   [MessageHeader(MustUnderstand = true)]
   public long Length;
   [MessageBodyMember(Order = 1)]
   public System.IO.Stream FileByteStream;
   public void Dispose()
   {
      if (FileByteStream != null)
      {
         FileByteStream.Close();
         FileByteStream = null;
      }
   }
}
[MessageContract]
```

```
public class UploadReply
{
   [MessageHeader]
   public bool UploadCompleted;
   [MessageHeader]
   public string UploadedMessage;
}
[MessageContract]
public class DownloadRequest
{
   [MessageBodyMember]
   public string FileName;
}
[MessageContract]
public class RemoteFileInfo : IDisposable
{
   [MessageHeader(MustUnderstand = true)]
   public string FileName;

   [MessageHeader(MustUnderstand = true)]
   public long Length;

   [MessageBodyMember(Order = 1)]
   public System.IO.Stream FileByteStream;

   public void Dispose()
   {
      if (FileByteStream != null)
      {
         FileByteStream.Close();
         FileByteStream = null;
      }
   }
}


public partial class FileTransferService :IFileTransfer
{
```

```
    private UsersVM dataSource;

    dedupEntities dataContext;
    MainWindow CurrentWindow;
    public FileTransferService()
    {
      dataSource = ((MainWindow)Application.Current.MainWindow).DataSource;
      CurrentWindow = (MainWindow)Application.Current.MainWindow;
      dataContext = new dedupEntities();
    }

    UploadReply IFileTransfer.UploadFile(UploadInfo request)
    {
      UploadReply tmp = new UploadReply();
      try
        {
          List<string> sessionUsers = dataSource.ProtocolSession.Select(u => u.UserID).ToList();
          if (!sessionUsers.Contains(request.UserName))
          {
            tmp.UploadCompleted = false;
            tmp.UploadedMessage = "Error!";
            return tmp;
          }
          ProtocolVM currentSession = dataSource.ProtocolSession.Where(s => s.UserID ==
request.UserName).FirstOrDefault();
          if (currentSession == null)
          {
            tmp.UploadCompleted  = false;
            tmp.UploadedMessage = "Error!";
            return tmp;
          }

          currentSession.Stage = Stage.UploadStarted;

          string txt = request.UserName  + " Started uploading " + request.FileName + "\tSize: " + request.Length;
          CurrentWindow.AddNotification(txt);

          string t = "Current Session FileID: " + currentSession.FileHash;
          CurrentWindow.AddNotification(t);
```

```
            Guid g = Guid.NewGuid();
            CurrentWindow.AddNotification(g.ToString());
            string filePath = @"C:\JJU\" + g.ToString();

            int chunkSize = 2048;
            byte[] buffer = new byte[chunkSize];

            using (System.IO.FileStream writeStream = new System.IO.FileStream(filePath,
System.IO.FileMode.CreateNew, System.IO.FileAccess.Write))
            {
               do
               {
                  int bytesRead = request.FileByteStream.Read(buffer, 0, chunkSize);
                  if (bytesRead == 0) break;
                  writeStream.Write(buffer, 0, bytesRead);
               } while (true);
               CurrentWindow.AddNotification("Uploading completed!");
            }

         user USER = dataContext.users.Where((u) => u.UserID.ToLower() ==
request.UserName.ToLower()).FirstOrDefault();
         string fileID = currentSession.FileHash;
         if (USER != null)
         {
            FileData fileData = new FileData()
            {
               FileID = fileID,
               CipherID = request.CipherHash,
               CipherKey = request.CipherKey,
               PhysicalLocation = filePath
            };

            FileOwner fileOwner = new FileOwner()
            {
               FileID = fileID,
               UserID = request.UserName,
               OriginalFileName = request.FileName
            };
```

```
                dataContext.FileDatas.Add(fileData);
                dataContext.FileOwners.Add(fileOwner);
                dataContext.SaveChanges();
                tmp.UploadCompleted = true;
                tmp.UploadedMessage = "Done!";
                return tmp;
            }
                tmp.UploadCompleted = true;
                tmp.UploadedMessage  = "Error";
                return tmp;
            }
        catch (Exception ex)
        {
            CurrentWindow.AddNotification("Error while uploading: \n" + ex.Message);
            tmp.UploadCompleted = false;
            tmp.UploadedMessage  = "Error";
            return tmp;
        }
    }
}

    public RemoteFileInfo DownloadFile(DownloadRequest request)
    {
        System.IO.FileInfo fileInfo = new System.IO.FileInfo(request.FileName);

        CurrentWindow.AddNotification("Sending stream " + request.FileName + " to client");
        CurrentWindow.AddNotification("Size " + fileInfo.Length);

        if (!fileInfo.Exists)
        {
            CurrentWindow.AddNotification("File not found " + request.FileName);
            return new RemoteFileInfo();
        };
        System.IO.FileStream stream = new System.IO.FileStream(request.FileName, System.IO.FileMode.Open,
System.IO.FileAccess.Read);
        RemoteFileInfo result = new RemoteFileInfo();
        result.FileName = request.FileName;
        result.Length = fileInfo.Length;
```

```
        result.FileByteStream = stream;
        return result;

    }
```

Appendix V

Source code Helper classes of the implementation of server component

```
public static partial class PasswordHelper
{

    public static byte[] GenerateSaltValue()
    {
        byte[] saltBytes;

        int minSaltSize = 16;
        int maxSaltSize = 32;

        Random random = new Random();
        int saltSize = random.Next(minSaltSize, maxSaltSize);

        saltBytes = new byte[saltSize];

        RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();

        rng.GetNonZeroBytes(saltBytes);

        return saltBytes;
    }


    public static string ComputeHash(string plainText, string hashAlgorithm, byte[] saltBytes)
    {
        if (saltBytes == null)
        {
            saltBytes = GenerateSaltValue();
        }

        byte[] plainTextBytes = Encoding.UTF8.GetBytes(plainText);
```

```
byte[] plainTextWithSaltBytes = new byte[plainTextBytes.Length + saltBytes.Length];

for (int i = 0; i < plainTextBytes.Length; i++)
   plainTextWithSaltBytes[i] = plainTextBytes[i];

for (int i = 0; i < saltBytes.Length; i++)
   plainTextWithSaltBytes[plainTextBytes.Length + i] = saltBytes[i];

HashAlgorithm hash;

if (hashAlgorithm == null)
   hashAlgorithm = "";

switch (hashAlgorithm.ToUpper())
{
   case "SHA1":
      hash = new SHA1Managed();
      break;

   case "SHA256":
      hash = new SHA256Managed();
      break;

   case "SHA384":
      hash = new SHA384Managed();
      break;

   case "SHA512":
      hash = new SHA512Managed();
      break;

   default:
      hash = new MD5CryptoServiceProvider();
      break;
}

byte[] hashBytes = hash.ComputeHash(plainTextWithSaltBytes);
```

```
        byte[] hashWithSaltBytes = new byte[hashBytes.Length +
                                           saltBytes.Length];

        for (int i = 0; i < hashBytes.Length; i++)
            hashWithSaltBytes[i] = hashBytes[i];

        for (int i = 0; i < saltBytes.Length; i++)
            hashWithSaltBytes[hashBytes.Length + i] = saltBytes[i];

        string hashValue = Convert.ToBase64String(hashWithSaltBytes);

        return hashValue;
    }

    public static bool VerifyHash(string plainText, string hashAlgorithm, string hashValue, string saltText)
    {
        Debug.Print(saltText);
        int hashSizeInBits, hashSizeInBytes;

        byte[] hashWithSaltBytes = Convert.FromBase64String(hashValue);
        byte[] saltBytes = Convert.FromBase64String(saltText);

        if (hashAlgorithm == null)
            hashAlgorithm = "";

        switch (hashAlgorithm.ToUpper())
        {
            case "SHA1":
                hashSizeInBits = 160;
                break;

            case "SHA256":
                hashSizeInBits = 256;
                break;

            case "SHA384":
                hashSizeInBits = 384;
```

```
            break;

        case "SHA512":
            hashSizeInBits = 512;
            break;

        default: // Must be MD5
            hashSizeInBits = 128;
            break;
    }

    hashSizeInBytes = hashSizeInBits / 8;

    if (hashWithSaltBytes.Length < hashSizeInBytes)
        return false;

    string expectedHashString = ComputeHash(plainText, hashAlgorithm, saltBytes);
    string tos = Convert.ToBase64String(saltBytes);
    Debug.Print(tos);

    return (hashValue == expectedHashString);
  }

}
public partial class MerkleHash
{
  const int Block_Size = 1024 * 1024;
  const int ZERO_BYTE_FILE = 0;
  private int Leaf_Count;
  private ArrayList LeafCollection;
  private FileStream FilePtr;
  private HashAlgorithm hashAlgorith = HashAlgorithm.Create("SHA512");

  private struct HashHolder
  {
    public byte[] HashValue;

    public HashHolder(byte[] HashValue)
```

```
        {
            this.HashValue = HashValue;
        }
    }

    public byte[] GetMerkleRootHash(string Filename)
    {
        HashHolder Result;
        try
        {
            FilePtr = new FileStream(Filename, FileMode.Open, FileAccess.Read, FileShare.ReadWrite);
            if (FilePtr.Length == ZERO_BYTE_FILE)
            {
                hashAlgorith.Initialize();
                Result.HashValue = hashAlgorith.ComputeHash(new byte[] { 0 });
            }
            else
            {
                Leaf_Count = (int)FilePtr.Length / Block_Size;
                if (FilePtr.Length % Block_Size > 0)
                    Leaf_Count++;
                LoadLeafHash();
                Result = GetRootHash();
            }
            FilePtr.Close();
            return Result.HashValue;
        }
        catch (Exception e)
        {
            System.Diagnostics.Debug.WriteLine("error while trying to get TTH for file: " +
                Filename + ". (" + e.Message + ")");
            if (FilePtr != null)
                FilePtr.Close();

            return null;
        }
    }
```

```
private Stack<List<HashItem>> lst;
public Stack<List<HashItem>> HashData
{
   get
   {
      return lst;
   }
}

List<HashItem> _leafHashes = new List<HashItem>();
public string ComputeRootHash(string filename, out List<HashItem> HData)
{
   FilePtr = new FileStream(filename, FileMode.Open, FileAccess.Read, FileShare.ReadWrite);
   Leaf_Count = (int)FilePtr.Length / Block_Size;
   if (FilePtr.Length % Block_Size > 0)
      Leaf_Count++;

   LoadLeafHash();
   FilePtr.Close();
   HData = new List<HashItem>();
   FillStackOfHashValues();

   if (lst.Peek().Count == 1)
   {
      HashItem root = lst.Peek()[0];
      lst.Pop();
      do
      {
         HData.AddRange(lst.Pop());
      } while (lst.Count != 0);
      return root.ID;
   }
   HData = null;
   return "null means error";

}

private void LoadLeafHash()
```

```
    {
        LeafCollection = new ArrayList();

        for (int i = 0; i < (int)Leaf_Count / 2; i++)
        {
            byte[] BlockA = new byte[Block_Size], BlockB = new byte[Block_Size];

            FilePtr.Read(BlockA, 0, 1024);
            int DataSize = FilePtr.Read(BlockB, 0, 1024);

            if (DataSize < Block_Size)
                BlockB = ByteExtract(BlockB, DataSize);

            leafHashValues.Add(new HashItem()
            {
                ID = Convert.ToBase64String(GetLeafHash(BlockA)),
                PID = null
            });

            leafHashValues.Add(new HashItem()
            {
                ID = Convert.ToBase64String(GetLeafHash(BlockB)),
                PID = null
            });

            LeafCollection.Add(new HashHolder(GetIntermideateHash(GetLeafHash(BlockA),
GetLeafHash(BlockB))));
        }

        if (Leaf_Count % 2 != 0)
        {
            byte[] Block = new byte[Block_Size];
            int DataSize = FilePtr.Read(Block, 0, 1024);

            if (DataSize < 1024)
                Block = ByteExtract(Block, DataSize);

            leafHashValues.Add(new HashItem()
```

```
            {
                ID = Convert.ToBase64String(GetLeafHash(Block)),
                PID=null
            });

            LeafCollection.Add(new HashHolder(GetLeafHash(Block)));
        }
    }

    private HashHolder GetRootHash()
    {
        ArrayList InternalCollection = new ArrayList();
        do
        {
            InternalCollection = new ArrayList(LeafCollection);
            LeafCollection.Clear();

            while (InternalCollection.Count > 1)
            {
                byte[] HashA = ((HashHolder)InternalCollection[0]).HashValue;
                byte[] HashB = ((HashHolder)InternalCollection[1]).HashValue;

                HashHolder IH = new HashHolder(GetIntermideateHash(HashA, HashB));

                LeafCollection.Add(IH);

                InternalCollection.RemoveAt(0);
                InternalCollection.RemoveAt(0);
            }

            if (InternalCollection.Count > 0)
                LeafCollection.Add(InternalCollection[0]);
        } while (LeafCollection.Count > 1);

        return (HashHolder)LeafCollection[0];
    }

    internal byte[] GetIntermideateHash(byte[] LeafA, byte[] LeafB)
```

```
    {
        byte[] Data = new byte[LeafA.Length + LeafB.Length];

        LeafA.CopyTo(Data, 0);
        LeafB.CopyTo(Data, LeafA.Length);

        hashAlgorith.Initialize();
        return hashAlgorith.ComputeHash(Data);
    }

    internal byte[] GetLeafHash(byte[] Raw_Data)
    {
        byte[] Data = new byte[Raw_Data.Length];

        Raw_Data.CopyTo(Data, 0);

        hashAlgorith.Initialize();
        return hashAlgorith.ComputeHash(Data);
    }

    private byte[] ByteExtract(byte[] Raw_Data, int Data_Length)
    {
        byte[] Data = new byte[Data_Length];

        for (int i = 0; i < Data_Length; i++)
            Data[i] = Raw_Data[i];

        return Data;
    }

    public ArrayList ListLeafHash(string Filename)
    {
        FileStream F = new FileStream(Filename, FileMode.Open, FileAccess.Read, FileShare.ReadWrite);
        int iCnt;
        iCnt = (int)F.Length / Block_Size;
        if (F.Length % Block_Size > 0)
            iCnt++;
```

```
        ArrayList lc = new ArrayList();

        for (int i = 0; i < (int)iCnt / 2; i++)
        {
            byte[] BlockA = new byte[Block_Size], BlockB = new byte[Block_Size];

            F.Read(BlockA, 0, 1024);
            int DataSize = F.Read(BlockB, 0, 1024);

            if (DataSize < Block_Size)
                BlockB = ByteExtract(BlockB, DataSize);

            BlockA = GetLeafHash(BlockA);
            BlockB = GetLeafHash(BlockB);

            HashHolder ih = new HashHolder(GetIntermideateHash(BlockA, BlockB));
            lc.Add(Convert.ToBase64String(ih.HashValue));
        }

        if (iCnt % 2 != 0)
        {
            byte[] Block = new byte[Block_Size];
            int DataSize = F.Read(Block, 0, 1024);

            if (DataSize < 1024)
                Block = ByteExtract(Block, DataSize);

            lc.Add(new HashHolder(GetLeafHash(Block)));
        }
        F.Close();
        return lc;
    }

    private byte[] GetCopy(byte[] Raw_Data)
    {
        byte[] Data = new byte[Raw_Data.Length];

        for (int i = 0; i < Raw_Data.Length; i++)
```

```
            Data[i] = Raw_Data[i];

        return Data;
    }

}
```

Appendix VI source code configuration of the server component

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>

 <configSections>
  <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework,
Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false" />
 </configSections>

 <system.serviceModel>
  <bindings>
   <basicHttpBinding>
    <binding name="FileTransferServicesBinding"
                    transferMode="Streamed" messageEncoding="Mtom" maxReceivedMessageSize="10067108864"/>
   </basicHttpBinding>
   <wsHttpBinding>
    <binding name="ProtocolServicesBinding"
                    messageEncoding="Mtom" maxReceivedMessageSize="2147483647"/>
   </wsHttpBinding>
  </bindings>
  <services>

  <service name="FileServiceHost.AuthenticationService">
    <endpoint address="" binding="wsHttpBinding" contract="FileServiceHost.ILogIn">
     <identity>
      <dns value="localhost" />
     </identity>
    </endpoint>
    <endpoint address="mex" binding="wsHttpBinding" contract="IMetadataExchange" />
    <host>
     <baseAddresses>
      <add baseAddress="http://localhost:8080/Authenticator" />
     </baseAddresses>
    </host>
   </service>

   <service name="FileServiceHost.ProtocolService">
    <endpoint address="" binding="wsHttpBinding"
                    contract="FileServiceHost.IProtocol" bindingConfiguration="ProtocolServicesBinding" >
     <identity>
      <dns value="localhost" />
     </identity>
```

```
      </endpoint>
      <endpoint address="mex" binding ="wsHttpBinding" contract="IMetadataExchange"/>
      <host>
       <baseAddresses>
        <add baseAddress="http://localhost:8081/Proof" />
       </baseAddresses>
      </host>
     </service>

     <service name="FileServiceHost.FileTransferService">
      <endpoint address="" binding="basicHttpBinding"
                          contract="FileServiceHost.IFileTransfer"
                          bindingConfiguration="FileTransferServicesBinding">
       <identity>
        <dns value="localhost" />
       </identity>
      </endpoint>
      <endpoint address="mex" binding="basicHttpBinding" contract="IMetadataExchange" />
      <host>
       <baseAddresses>
        <add baseAddress="http://localhost:8082/FileTransfer" />
       </baseAddresses>
      </host>
     </service>
    </services>

    <behaviors>
     <serviceBehaviors>
      <behavior name="MyServiceTypeBehaviors">
       <serviceMetadata httpGetEnabled="true" />
       <serviceDebug includeExceptionDetailInFaults="true" />
      </behavior>
      <behavior name="">
       <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true" />
       <serviceDebug includeExceptionDetailInFaults="true" />
      </behavior>
     </serviceBehaviors>
    </behaviors>
   </system.serviceModel>
   <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
   </startup>
</configuration>
```

Appendix VII

Source code of the implementation ILogIn interface on the client

```
public partial class LogInClient : System.ServiceModel.ClientBase<dedupClient.rAuthService.ILogIn>,
dedupClient.rAuthService.ILogIn {

    public LogInClient() {
    }

    public LogInClient(string endpointConfigurationName) :
        base(endpointConfigurationName) {
    }

    public LogInClient(string endpointConfigurationName, string remoteAddress) :
        base(endpointConfigurationName, remoteAddress) {
    }

    public LogInClient(string endpointConfigurationName, System.ServiceModel.EndpointAddress remoteAddress) :
        base(endpointConfigurationName, remoteAddress) {
    }

    public LogInClient(System.ServiceModel.Channels.Binding binding, System.ServiceModel.EndpointAddress remoteAddress) :
        base(binding, remoteAddress) {
    }

    public bool LogIn(string username, string password) {
      return base.Channel.LogIn(username, password);
    }

    public System.Threading.Tasks.Task<bool> LogInAsync(string username, string password) {
      return base.Channel.LogInAsync(username, password);
    }

    public dedupClient.rAuthService.RegistrationState Register(dedupClient.rAuthService.UserInfo USER) {
      return base.Channel.Register(USER);
    }

    public System.Threading.Tasks.Task<dedupClient.rAuthService.RegistrationState>
RegisterAsync(dedupClient.rAuthService.UserInfo USER) {
        return base.Channel.RegisterAsync(USER);
    }
```

```
        dedupClient.rAuthService.FilesVM dedupClient.rAuthService.ILogIn.ListUserFiles(dedupClient.rAuthService.ListRequest
request) {
            return base.Channel.ListUserFiles(request);
        }

        public string ListUserFiles(string UserID, out System.Collections.Generic.List<dedupClient.rAuthService.FileInfoVM> Files)
{
            dedupClient.rAuthService.ListRequest inValue = new dedupClient.rAuthService.ListRequest();
            inValue.UserID = UserID;
            dedupClient.rAuthService.FilesVM retVal = ((dedupClient.rAuthService.ILogIn)(this)).ListUserFiles(inValue);
            Files = retVal.Files;
            return retVal.FullName;
        }


        System.Threading.Tasks.Task<dedupClient.rAuthService.FilesVM>
dedupClient.rAuthService.ILogIn.ListUserFilesAsync(dedupClient.rAuthService.ListRequest request) {
            return base.Channel.ListUserFilesAsync(request);
        }

        public System.Threading.Tasks.Task<dedupClient.rAuthService.FilesVM> ListUserFilesAsync(string UserID) {
            dedupClient.rAuthService.ListRequest inValue = new dedupClient.rAuthService.ListRequest();
            inValue.UserID = UserID;
            return ((dedupClient.rAuthService.ILogIn)(this)).ListUserFilesAsync(inValue);
        }
    }
```

Appendix VIII

Source code of the implementation IProtocol interface on the client

```
   public partial class ProtocolClient : System.ServiceModel.ClientBase<dedupClient.rProtocolService.IProtocol>,
dedupClient.rProtocolService.IProtocol {

      public ProtocolClient() {
      }

      public ProtocolClient(string endpointConfigurationName) :
            base(endpointConfigurationName) {
      }

      public ProtocolClient(string endpointConfigurationName, string remoteAddress) :
            base(endpointConfigurationName, remoteAddress) {
      }

      public ProtocolClient(string endpointConfigurationName, System.ServiceModel.EndpointAddress
remoteAddress) :
            base(endpointConfigurationName, remoteAddress) {
      }

      public ProtocolClient(System.ServiceModel.Channels.Binding binding, System.ServiceModel.EndpointAddress
remoteAddress) :
            base(binding, remoteAddress) {
      }


      dedupClient.rProtocolService.UploadRequestReply
dedupClient.rProtocolService.IProtocol.RequestUpload(dedupClient.rProtocolService.UploadRequest request) {
         return base.Channel.RequestUpload(request);
      }

      public string RequestUpload(string username, string FileHash, out int ProofIndex, out bool UploadGranted) {
         dedupClient.rProtocolService.UploadRequest inValue = new dedupClient.rProtocolService.UploadRequest();
         inValue.username = username;
         inValue.FileHash = FileHash;
         dedupClient.rProtocolService.UploadRequestReply retVal =
((dedupClient.rProtocolService.IProtocol)(this)).RequestUpload(inValue);
```

```
            ProofIndex = retVal.ProofIndex;
            UploadGranted = retVal.UploadGranted;
            return retVal.Message;
        }


        System.Threading.Tasks.Task<dedupClient.rProtocolService.UploadRequestReply>
dedupClient.rProtocolService.IProtocol.RequestUploadAsync(dedupClient.rProtocolService.UploadRequest request) {
            return base.Channel.RequestUploadAsync(request);
        }

        public System.Threading.Tasks.Task<dedupClient.rProtocolService.UploadRequestReply>
RequestUploadAsync(string username, string FileHash) {
            dedupClient.rProtocolService.UploadRequest inValue = new dedupClient.rProtocolService.UploadRequest();
            inValue.username = username;
            inValue.FileHash = FileHash;
            return ((dedupClient.rProtocolService.IProtocol)(this)).RequestUploadAsync(inValue);
        }


        dedupClient.rProtocolService.UploadRequestReply
dedupClient.rProtocolService.IProtocol.SendProof(dedupClient.rProtocolService.ProofList request) {
            return base.Channel.SendProof(request);
        }

        public string SendProof(string FileName, string UserName,
System.Collections.Generic.List<dedupClient.rProtocolService.HashItem> Proof, out int ProofIndex, out bool
UploadGranted) {
            dedupClient.rProtocolService.ProofList inValue = new dedupClient.rProtocolService.ProofList();
            inValue.FileName = FileName;
            inValue.UserName = UserName;
            inValue.Proof = Proof;
            dedupClient.rProtocolService.UploadRequestReply retVal =
((dedupClient.rProtocolService.IProtocol)(this)).SendProof(inValue);
            ProofIndex = retVal.ProofIndex;
            UploadGranted = retVal.UploadGranted;
            return retVal.Message;
        }
```

```
        System.Threading.Tasks.Task<dedupClient.rProtocolService.UploadRequestReply>
dedupClient.rProtocolService.IProtocol.SendProofAsync(dedupClient.rProtocolService.ProofList request) {
            return base.Channel.SendProofAsync(request);
        }

        public System.Threading.Tasks.Task<dedupClient.rProtocolService.UploadRequestReply>
SendProofAsync(string FileName, string UserName,
System.Collections.Generic.List<dedupClient.rProtocolService.HashItem> Proof) {
            dedupClient.rProtocolService.ProofList inValue = new dedupClient.rProtocolService.ProofList();
            inValue.FileName = FileName;
            inValue.UserName = UserName;
            inValue.Proof = Proof;
            return ((dedupClient.rProtocolService.IProtocol)(this)).SendProofAsync(inValue);
        }

        dedupClient.rProtocolService.OwnershipRequestReply
dedupClient.rProtocolService.IProtocol.SendCipherHash(dedupClient.rProtocolService.OwnershipRequest request) {
            return base.Channel.SendCipherHash(request);
        }

        public string SendCipherHash(string FileName, string username, string FileHash) {
            dedupClient.rProtocolService.OwnershipRequest inValue = new
dedupClient.rProtocolService.OwnershipRequest();
            inValue.FileName = FileName;
            inValue.username = username;
            inValue.FileHash = FileHash;
            dedupClient.rProtocolService.OwnershipRequestReply retVal =
((dedupClient.rProtocolService.IProtocol)(this)).SendCipherHash(inValue);
            return retVal.Message;
        }

        System.Threading.Tasks.Task<dedupClient.rProtocolService.OwnershipRequestReply>
dedupClient.rProtocolService.IProtocol.SendCipherHashAsync(dedupClient.rProtocolService.OwnershipRequest
request) {
            return base.Channel.SendCipherHashAsync(request);
        }
```

```
        public System.Threading.Tasks.Task<dedupClient.rProtocolService.OwnershipRequestReply>
SendCipherHashAsync(string FileName, string username, string FileHash) {
            dedupClient.rProtocolService.OwnershipRequest inValue = new
dedupClient.rProtocolService.OwnershipRequest();
            inValue.FileName = FileName;
            inValue.username = username;
            inValue.FileHash = FileHash;
            return ((dedupClient.rProtocolService.IProtocol)(this)).SendCipherHashAsync(inValue);
        }
    }
```

Appendix IX

Source code of the implementation IFileTransfer interface on the client

```
public partial class FileTransferClient : System.ServiceModel.ClientBase<dedupClient.rFileService.IFileTransfer>,
dedupClient.rFileService.IFileTransfer {

    public FileTransferClient() {
    }

    public FileTransferClient(string endpointConfigurationName) :
        base(endpointConfigurationName) {
    }

    public FileTransferClient(string endpointConfigurationName, string remoteAddress) :
        base(endpointConfigurationName, remoteAddress) {
    }

    public FileTransferClient(string endpointConfigurationName, System.ServiceModel.EndpointAddress
remoteAddress) :
        base(endpointConfigurationName, remoteAddress) {
    }

    public FileTransferClient(System.ServiceModel.Channels.Binding binding,
System.ServiceModel.EndpointAddress remoteAddress) :
        base(binding, remoteAddress) {
    }


    dedupClient.rFileService.UploadReply
dedupClient.rFileService.IFileTransfer.UploadFile(dedupClient.rFileService.UploadInfo request) {
        return base.Channel.UploadFile(request);
    }

    public bool UploadFile(string CipherHash, string CipherKey, string FileName, long Length, string UserName,
System.IO.Stream FileByteStream, out string UploadedMessage) {
        dedupClient.rFileService.UploadInfo inValue = new dedupClient.rFileService.UploadInfo();
```

```
            inValue.CipherHash = CipherHash;
            inValue.CipherKey = CipherKey;
            inValue.FileName = FileName;
            inValue.Length = Length;
            inValue.UserName = UserName;
            inValue.FileByteStream = FileByteStream;
            dedupClient.rFileService.UploadReply retVal =
((dedupClient.rFileService.IFileTransfer)(this)).UploadFile(inValue);
            UploadedMessage = retVal.UploadedMessage;
            return retVal.UploadCompleted;
        }


        System.Threading.Tasks.Task<dedupClient.rFileService.UploadReply>
dedupClient.rFileService.IFileTransfer.UploadFileAsync(dedupClient.rFileService.UploadInfo request) {
            return base.Channel.UploadFileAsync(request);
        }

        public System.Threading.Tasks.Task<dedupClient.rFileService.UploadReply> UploadFileAsync(string
CipherHash, string CipherKey, string FileName, long Length, string UserName, System.IO.Stream FileByteStream) {
            dedupClient.rFileService.UploadInfo inValue = new dedupClient.rFileService.UploadInfo();
            inValue.CipherHash = CipherHash;
            inValue.CipherKey = CipherKey;
            inValue.FileName = FileName;
            inValue.Length = Length;
            inValue.UserName = UserName;
            inValue.FileByteStream = FileByteStream;
            return ((dedupClient.rFileService.IFileTransfer)(this)).UploadFileAsync(inValue);
        }


        dedupClient.rFileService.RemoteFileInfo
dedupClient.rFileService.IFileTransfer.DownloadFile(dedupClient.rFileService.DownloadRequest request) {
            return base.Channel.DownloadFile(request);
        }

        public long DownloadFile(ref string FileName, out System.IO.Stream FileByteStream) {
            dedupClient.rFileService.DownloadRequest inValue = new dedupClient.rFileService.DownloadRequest();
            inValue.FileName = FileName;
```

```
            dedupClient.rFileService.RemoteFileInfo retVal =
((dedupClient.rFileService.IFileTransfer)(this)).DownloadFile(inValue);
            FileName = retVal.FileName;
            FileByteStream = retVal.FileByteStream;
            return retVal.Length;
        }


        System.Threading.Tasks.Task<dedupClient.rFileService.RemoteFileInfo>
dedupClient.rFileService.IFileTransfer.DownloadFileAsync(dedupClient.rFileService.DownloadRequest request) {
            return base.Channel.DownloadFileAsync(request);
        }

        public System.Threading.Tasks.Task<dedupClient.rFileService.RemoteFileInfo> DownloadFileAsync(string
FileName) {
            dedupClient.rFileService.DownloadRequest inValue = new dedupClient.rFileService.DownloadRequest();
            inValue.FileName = FileName;
            return ((dedupClient.rFileService.IFileTransfer)(this)).DownloadFileAsync(inValue);
        }
    }
```

DECLARATION

I, the undersigned, hereby declare that this thesis is my original work performed under the supervision of Dr. Srineevas Nune, has not been presented as a thesis for a degree program in any other university and all sources of materials used for the thesis are duly acknowledged.

Tesfamichael Gebegziabher

February 2015.