Current State of Cryptography and Design of an Online Storage System

Tommy Jonsson

December 14, 2009
Master's Thesis in Computing Science, 30 ECTS-credits
Supervisor at CS-UmU: Thomas Johansson
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

The rapid growth of Internet usage has resulted in an increasing demand for storing files on an online system as opposed to the traditional way of storing them on the local hard drive. Since the online system is exposed to public access, more requirements are imposed on the storage system, such as secure access to files and secure storing of files. A control for both these problems can be cryptography. Common symmetric algorithms are DES, Triple-DES and AES, and one popular asymmetric algorithm is RSA.

Benchmarks show that symmetric algorithms outperform asymmetric algorithms in both encryption and decryption. AES is the fastest, using any key size, among the tested algorithms. Cryptanalysis in academic papers conclude that the only algorithm among those examined in this paper that is considered to be insecure is DES, due to usage of short keys.

This thesis proposes an online storage system fully implemented in .NET 3.5, featuring storage of files as well as additional file-type dependent functionality. PDF documents can be split, merged and previewed using this system, and it can be extended to support other functionality on any file types. Encryption of files is supported using the 128-bit AES algorithm.

Contents

1 Introduction **1**
1.1 Acino 1
1.2 Outline 1
1.3 Problem Statement 2
1.4 Goals 2
1.5 Related Work 2

2 Cryptography **3**
2.1 Purpose of Cryptography 3
2.1.1 Terminology 3
2.1.2 Confusion and Diffusion 4
2.2 Cryptanalysis 4
2.2.1 Breakable algorithms 4
2.2.2 Trustworthy Cryptosystems 5
2.3 Symmetric Cryptosystems 5
2.4 Asymmetric Cryptosystems 6
2.5 Common Algorithms 6
2.5.1 Data Encryption Standard 6
2.5.2 Advanced Encryption Standard 8
2.5.3 RSA 9

3 Evaluation of Common Cryptosystems **11**
3.1 Performance 11
3.1.1 Test program 11
3.1.2 Procedure 12
3.1.3 Results 12
3.2 Security 13
3.3 Conclusion 15

4 Implementation **17**
4.1 Database 17

4.1.1 Files and Directories . . . 17
4.1.2 Representing the file trees . . . 18
4.1.3 Access Control . . . 18
4.1.4 User Information . . . 18
4.2 Library . . . 19
4.2.1 Design Principles . . . 19
4.2.2 File Management . . . 19
4.2.3 User Management . . . 21
4.2.4 Password Management . . . 21
4.2.5 Encryption . . . 22
4.2.6 File Sharing . . . 23
4.2.7 PDF Functionality . . . 24

5 Results 27
5.1 Class Overview . . . 27
5.2 Basic Usage . . . 28
5.3 Configuration . . . 29
5.4 Code Examples . . . 29
5.4.1 Authenticate a user . . . 30
5.4.2 Listing Files . . . 30
5.4.3 Using PDFFilter . . . 31
5.4.4 Encrypting a file . . . 32

6 Conclusions 33
6.1 Limitations . . . 33
6.2 Future Work . . . 34

7 Acknowledgements 35

References 37

A User's Guide 39
A.1 Overview . . . 39
A.2 Technology . . . 39
A.3 Basic Usage . . . 40
A.4 Configuration . . . 40
A.5 Code Examples . . . 40
A.5.1 Authenticate a user . . . 40
A.5.2 Listing Files . . . 41
A.5.3 Using PDFFilter . . . 42
A.5.4 Encrypting a file . . . 42
A.6 Class Information . . . 43

A.6.1 FileInfo 43
A.6.2 DirectoryInfo 44
A.6.3 UserInfo 44
A.6.4 AccessInfo 44
A.6.5 MediaStorage 45
A.6.6 PDFFilter 46
A.7 Internal Classes 46
A.7.1 CryptoTools 46
A.7.2 UserDBConnector 47
A.7.3 FileDBConnector 47
A.7.4 FileManager 47
A.7.5 GSWrapper 47

B Database Design 49
B.1 Schema 49
B.1.1 Users 49
B.1.2 LoginInfo 49
B.1.3 Files 50
B.1.4 AccessList 50
B.2 Stored Procedures 51

List of Figures

2.1 A symmetric cryptosystem. 5
2.2 An asymmetric cryptosystem. 6

3.1 Encryption and decryption of a photo. 13
3.2 Encryption and decryption of a CD Image. 14

4.1 Encryption header. 22

5.1 Library datastructures. 27
5.2 Class overview. 28
5.3 Database datastructures. 29

List of Tables

3.1 Execution times of encryption and decryption of a textfile (ms). 12
3.2 Execution times of encryption and decryption of a photo (ms). 13
3.3 Execution times of encryption and decryption of a CD image (s). 13

B.1 Attributes of the table Users. 49
B.2 Attributes of the table LoginInfo. 50
B.3 Attributes of the table Files. 50
B.4 Attributes of the table AccessList. 51

Chapter 1

Introduction

The rapid growth of Internet infrastructure with high capacity has made it possible to provide online storage to users. Instead of storing files on a local computer, people can use of online storage to access the content from any computer as well as be safe against threats such as disc failures. Such a service must fulfill requirements on providing good performance, easy usage as well as a secure storing. This thesis describes the process of designing and implementing an online storage system that addresses these requirements. This system will be referred to as *MediaStore* in this report.

1.1 Acino

This thesis is based on a proposal offered by the company Acino, based in Umeå. Acino works with product development as well as provides consulting services to the telecom industry and to companies in the bank and insurance businesses. One of Acino's largest products, *Fullmaktskontoret*, is currently using another online storage system, *Mediaarkivet*, for storing files but Acino want to investigate the possibility of developing a new system on their own.

1.2 Outline

The remaining part of this chapter formulates the problem statement, the goals this project is set to accomplish and which restrictions have been imposed on the project in order for it to be completed within the specified time frame.

Chapter 2 presents a general study of cryptography and what the current state of field is today. The chapter describes the goal and procedure of cryptography and studies the use of both symmetric- and asymetric cryptosystems. Three popular symmetric cryptosystems are examined more in detail: the Data Encryption Standard (DES) and the newer variant, 3-DES, and the replacement for DES, the Advanced Encryption Standard (AES). Further asymetric, or public key, cryptosystems are examined with a study of the algorithm RSA.

Chapter 3 evaluates the algorithms examined in Chapter 2 based on performance and provided security. Performance is evaluated using a small program written in Java that makes use of the algorithms mentioned above and measures encryption and decryption performance on various files.

The implementation and design of the proposed storage system is described in Chapter 4, beginning with the design of the database used for storing metadata and following up with a description of the code library that constitutes the main part of the solution. Chapter 5 describes the resulting solution and how to use some of its features.

Chapter 6 discusses the conclusions of the thesis, the limitations of the project and how further work may improve it.

1.3 Problem Statement

The proprietary system Mediaarkivet is currently used as the main solution for storing files by several systems developed by Acino. Mediaarkivet can store any type of file but the systems in question mainly use it for storing Pdf documents. Acino would like to investigate the feasibility of developing a replacement for that storage system, with some additional functionality. A storage solution must be scalable to store a large number of files and provide a modular design to simplify further development. The system should also be able to store files in a secure manner and provide encryption and decryption capabilities.

The purpose of the Master's Thesis is to design a system that stores and provides access to any types of files. The system should be able to provide additional functionality that is filetype dependent, with a modular design so that additional functionality for more filetypes can be added to the system when needed.

1.4 Goals

The primary goal is to create a prototype of a system that may function as a replacement of Mediaarkivet. Thus the system will support the primary functionality of Mediaarkivet for managing files, furthermore providing a secure storage by featuring encryption of files. In detail, the goals of the thesis are the following:

– The system contains a module that provides the same functionality for Pdf documents as Mediaarkivet, as well as the capability to create previews of any page of a Pdf document.

– Users are able to make files available to other users, in order to simplify collaborative work.

– Perform a study of the current state of cryptography and examine which cryptosystems are suitable for usage in the system.

1.5 Related Work

There exists many commercial systems providing online storage. Mediaarkivet [med] is currently in use at Acino. Box [box] has features such as file sharing, online workspaces and version management. [AGRS04] presents a multimedia content management system for library applications.

Chapter 2

Cryptography

Cryptography is the art of disguising data so that it can not easily be read, modified or fabricated easily. It is a fundamental tool for controlling many security threats. This chapter will cover some of the research on cryptography and describe some common algorithms that are in use today.

2.1 Purpose of Cryptography

The main purpose of cryptography is to hide the original meaning of some information in order to prevent unauthorized access to it. For example, consider the procedure of sending a message M from a sender S to the recipient R. S gives the message to the transmission medium T which delivers the message to R. If an attacker O want to access the message, either to read, change or destroy it, O might try to access the message in any of the following ways:

- *Blocking it*: O may try to block the message so it never reaches R.
- *Intercepting it*: The confidentiality of the message is affected by O reading or listening to the message.
- *Modifying it*: O intercepts and modifies the message in some way, thus affecting the integrity of the message.
- *Fabrication*: O fabricates a message that appears as if coming from S, and arranges so it is delivered to R, thus affecting the integrity of the message.

Encrypting the message is a way to control all these threats.

2.1.1 Terminology

Encryption is the process of transforming a message according to an algorithm so that its meaning is no longer obvious. The encrypted form of a message is called **ciphertext**. Decryption is the reverse process, transforming an encrypted message back to its original form. This form is called **plaintext**.

If a plaintext P is encrypted by an encryption rule E it becomes a ciphertext C, and we write $C = E(P)$. Decrypting C with a decryption rule D is written $P = D(C)$. A system that performs both encryption and decryption is called **cryptosystem**, and

represents the function $P = D(E(P))$. That is, the system can encrypt a plaintext into a form that can not be understood by an outsider, as well as be able to convert the encrypted form back into plaintext.

Some algorithms makes use of a **key** K so that a ciphertext will depend on the plaintext that is used, the algorithm and the key. An algorithm that uses the same key for both encryption and decryption is called a **symmetric algorithm**, since encryption and decryption are mirror-image processes. This type of algorithms are described further in Section 2.3. Another class of algorithms uses two different keys, one for encryption and one for decryption. This class is called **asymmetric algorithms** and is described in Section 2.4.

2.1.2 Confusion and Diffusion

As stated previously, encryption is the process of transforming a plaintext so that its meaning is hidden to an attacker. The attacker should not be able to predict how the encrypted message, the ciphertext, will change if one character in the plaintext is changed. This characteristic is called **confusion**. An algorithm with a high degree of confusion makes it harder for an attacker to determine the relationship between plaintext, key and ciphertext.

Another important concept is **diffusion**. A cipher should spread the information from the plaintext over the ciphertext so that small changes in the plaintext results in many changes in the ciphertext. A high degree of diffusion means that an attacker must have access to a large portion of the ciphertext to determine the algorithm [PP07].

2.2 Cryptanalysis

Cryptanalysis is the process of breaking an encryption. A cryptanalyst want to deduce the original meaning of a ciphertext. This may also include finding the specific cryptosystem that was used and deducing a key, so that subsequent messages can easily be broken. An analyst can use a wide range of methods for breaking a message. Encrypted messages, known algorithms, intercepted plaintext, properties of languages and mathematical and statistical tools are means which can be used to deduce the original meaning of a message.

2.2.1 Breakable algorithms

An encryption algorithm is called **breakable** when an analyst can deduce how the algorithm works, just given enough time and data. However, this does not necessarily mean finding a practical way to recover the original meaning of a ciphertext. In academic cryptography, breaking an encryption simply means finding a weakness so that the complexity for deducing the original meaning is less than a brute force attack. A brute force attack, also called extensive search, is an attack when the attacker tries every possible key to find the correct one. The number of possible keys is determined by the length of the key. For example, a brute force attack on an encryption algorithm using 128-bit keys involves at most 2^{128} encryptions. But if there exist an attack involving just 2^{110} encryptions the algorithm is considered broken, even though the attack may still be practically infeasible. But such an attack do show that the algorithm does not perform as advertised [Sch00].

2.2.2 Trustworthy Cryptosystems

Since the purpose of using cryptography is to keep information secret from outsiders, there is a problem regarding trust when considering which cryptosystem to use. How can the user be sure that the cryptosystem is secure and have no unwanted properties such as a back-door? [PP07] identifies three important properties of a cryptosystem that can be trusted:

- It is based on sound mathematics.
- It has been analyzed by competent experts and found to be sound.
- It has stood the "test of time".

2.3 Symmetric Cryptosystems

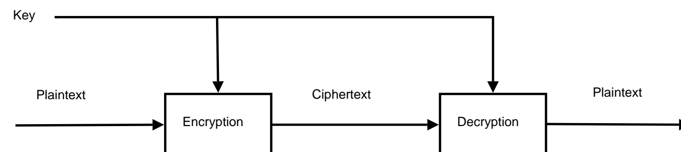

Figure 2.1: A symmetric cryptosystem.

Symmetric cryptosystems, also called "secret key", makes use of one key for both encryption and decryption. Figure 2.1 shows the process of symmetric encryption. Decryption is usually the inverse process of encryption, thus applying the steps of the encryption backwards. The users shares a secret key, with which all users can encrypt messages to send to all other, and all users can decrypt messages received from the other users.

A symmetric system also provides authentication as long as the key remain secret, since only the legitimate sender can produce a message that decrypts properly.

The symmetry of these kind of systems do however also lead to a problem concerning key distribution. How do two users that wish to communicate messages securely obtain the shared key? And if there exists a secure channel with which the key can be securely shared, why can not this channel be used for communicating the messages as well? These concerns were addressed in 1975 when public key cryptosystems were discovered [Whi88]. Examples of symmetric cryptosystems are DES and AES, covered in Sections 2.5.1 and 2.5.2, respectively.

2.4 Asymmetric Cryptosystems

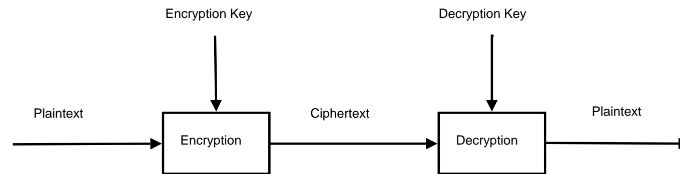

Figure 2.2: An asymmetric cryptosystem.

Asymmetric, or public key cryptosystems separates the capacities of encryption and decryption. These systems makes use of two keys, a private key that should be kept secret, and a public key that can be published publically. Each pair of keys has the following properties:

– Anything encrypted with one key can be decrypted with the other key.

– Given the public key it is computationally infeasible to discover the private key.

Figure 2.2 shows the process of encryption and decryption in an asymmetric system. In order for a user A to send a private message to another user B, A has to look up the public key of B and use it to encrypt the message. The message can only be decrypted with the corresponding private key, which is only known by the user B. With two keys, one secret and one public, a major problem with symmetric cryptosystems can be solved, namely key distribution.

Asymmetric cryptosystem can also be used for authentication. Any user can sign a message by encrypting it with his own private key. Any recipient can decrypt the message with the corresponding public key and verify the sender[Whi88].

RSA is an example of an asymmetric cryptosystem and is explained further in Section 2.5.3.

2.5 Common Algorithms

This section describe some popular cryptosystems that are in use today. The section describes a bit of the history of each cryptosystem and how the algorithm works in general, and concludes with an evaluation of the security each algorithm provides.

2.5.1 Data Encryption Standard

The Data Encryption Standard was a widely used symmetric encryption technique, but is now regarded as insecure because a brute force attack is practically feasible.

Overview

In the early 1970's the U.S National Bureau of Standards (NBS) recognized the need for protecting sensitive data. Cryptography was historically studied extensively at the Department of Defense and the Department of State but the very nature of the information they were encrypting made it difficult to release any material used there to the general

public. Therefore NBS was assigned the responsibility of developing an encryption algorithm that would be standardized and released publically. NBS contracted IBM, which continued their work on the algorithm Lucifer, and in 1976 the algorithm was standardized and named Data Encryption Standard, DES. During development, IBM worked with the National Security Agency (NSA) which was responsible for analyzing the strength of the algorithm [PP07].

Algorithm

DES is a block cipher using blocks of 64 bits and uses substitution and permutation to encrypt data. These two techniques are applied in sequence and then the procedure is repeated for 16 rounds, or cycles.

The algorithm makes use of a 64 bit key although only 56 bits are actually used for encryption, the remaining 8 bits are often used as a check sum or may even be discarded.

Strength of the Algorithm

Due to the secrecy imposed on the development by NSA, details regarding the design of the algorithm was not publically released for many years. This led to speculations of that the NSA had included "back doors" in the algorithm in order for the agency to easily decrypt messages. But the design principles were released as **differential cryptanalysis** was developed and published in 1990 in a paper by Eli Biham and Adi Shamir [BS90], and even extensive public scrutiny have not found anything that could indicate a back door [And01].

The differential cryptanalysis is a technique that can be applied to algorithms using substitution and permutation, and investigates how the algorithmic strength changes when an algorithm is changed in some way. As Biham and Shamir applied the technique on DES, they concluded that almost any change to the algorithm would only weaken it. Shortening the algorithm to 15 rounds allows a key to be determined in 2^{56} searches as well as 2^{35} when using 10 rounds. The results also shows that when the full 16 rounds are used, 2^{58} searches are needed, which is four times more than performing an extensive search for the key [BS90]. In fact this technique was known to IBM and the NSA when the DES was developed, and the algorithm was specifically designed to defeat such an attack [PP07].

The most serious question regarding the security of the DES is the length of the key used in the algorithm. As mentioned in Section 2.2.1, the length of a key determines how many possible keys there are. DES uses a key with the fixed size of 56 bits, thus there exists 2^{56} possible keys. Diffie and Hellman argued in a paper published in 1977 [WH77] that a 56 bit key was too short. With the hardware available in 1977 it was unfeasible to conduct an extensive search but they argued that over time more powerful hardware would eventually surpass the strength of DES. The authors suggested a machine with parallell design and estimated the cost to $20 million [PP07].

20 years later, a team of researchers won a contest sponsored by the organisation RSA Security, as they broke a message encrypted with DES using extensive search. They used a distributed system consisting of at most 14000 machines on the Internet to find the key in about four months.

In 1998 the Electronic Frontier Foundation built a custom DES keysearch machine with a budget of approximately $210 000 that broke an encrypted message in four days. The machine consisted of 1536 chips running at 40 MHz, each chip containing 24 search units that could perform a decryption in 16 cycles [ISB].

These two efforts have concluded that the original DES is not considered secure anymore due to the limits of the 56-bit key. However, in order to deal with this problem one can use the algorithm multiple times, with different keys. In [MH81] the authors show that two encryptions are not much better than one, it only doubles the work of an attacker. Triple DES, or 3DES, makes use of three keys. The plain text is encrypted with one key, decrypted with the second key and finally encrypted again with a third key, according to the following procedure

$$C = E(k_3, D(k_2, E(k_1, P)))$$

This process gives the encryption a strength as if using a key of 112 bits, since the strength of one of the keys is defeated as of the double DES attack [PP07].

2.5.2 Advanced Encryption Standard

The Advanced Encryption Standard, AES, is a block cipher that uses the algorithm Rijndael and has replaced DES as the cryptosystem used by the U.S government.

Overview

In the late 1990's, the U.S National Institute of Standards and Technology (NIST) saw the need for a new encryption standard to replace the aging DES. NIST concluded that unlike the development of DES, the selection process would be open so that anyone could propose an algorithm for the new encryption standard [DR02]. Each proposed algorithm had to be

- Unclassified
- Publicly disclosed
- Available royalty-free for use worldwide
- Symmetric block cipher algorithms, for blocks of 128 bits
- Usable with key sizes of 128, 192 and 256 bits

The selection process was announced in 1997 and in 1999 the field of candidates was narrowed down to five candidates. The cryptology community were invited to analyze and mount attacks on these candidates in order to find weaknesses in the algorithms. Although no apparent flaws were found in any of the five, the Rijndael algorithm were selected as the winner due to good overall efficiency and performance [DR02]. In 2001 NIST formally released the specification of AES [NIS01], and the encryption standard was adopted by the U.S government for encrypting government data transmissions and storage.

Algorithm

AES is a a block cipher using blocks of 128 bits. It can use keys of 128, 192 and 256 bits, although the Rijndael algorithm can be extended to use any key length that is a multiple of 64.

The algorithm primarily uses substitution, permutation, shifts and exclusive OR to perform the encryption. As in the case with DES, this AES uses repeat rounds, and

each round contains the steps described below. The number of rounds depends on the length of the key, where 10, 12 or 14 rounds are used for keys of length 128, 192 and 256 bits, respectively.

The steps conducted in each round are

- Byte substitution
- Shift row
- Mix column
- Add subkey

Strength of the algorithm

Although AES is relatively new compared to the DES, it has been subjected to extensive cryptanalysis since 1997 and no serious flaws has been discovered so far. Unlike the DES, the algorithm was developed by independent Belgian cryptographers with no connection to the NSA or any part of the U.S government, so there is no suspicion regarding hidden back doors or that the government has in some way weakened the algorithm [PP07].

In [FKL+00] the authors describe attacks on Rijndael when the algorithm is reduced to 6, 7 and 8 rounds. It is concluded that their attack on 8 round Rijndael reduces the complexity from 2^{192} and 2^{256} when using 192 and 256 bit keys to between 2^{128} and 2^{119}. Although this attack manage to find the key without an extensive search, the complexity is still too large for a practical attack. It should be noted that no attacks have been discovered so far on the algorithm when it uses the full number of rounds.

As stated in Section 2.5.1, DES encryption lasted for about 20 years. It is hard to speculate how long AES will last, but since it starts with a key almost twice as long as the key DES uses, and that AES can use up to 256 bit keys, makes brute force attacks unlikely to break the algorithm for the coming years. Another difference when compared to DES is that the underlying algorithm of AES allows for much larger keys than 256 bits, which means the standard could be extended to even larger keys if deemed necessary [PP07].

2.5.3 RSA

The Rivest-Shamir-Adelman Encryption cryptosystem is an asymmetric encryption algorithm, named after its three inventors. The algorithm was published in 1978 and is based on an underlying hard problem in order to encrypt data, as opposed to substitution and permutation used by AES and DES.

Overview

The RSA cryptosystem is a block cipher that is based on number theory and the difficulty of determining the prime factors of a number. RSA was published as an implementation of the public key cryptosystem that was described by Diffie and Hellman in [DH76].

Algorithm

RSA makes use of a public key e' and a private key d'. These keys are in fact interchangeable and d' could be used as the public key and e' as the private key. However,

in this text e' will be chosen as the public key. The public key will be published so that it is freely available and the private key must be kept secret.

In RSA each plain text block P is treated as a number. P is encrypted by calculating

$$C = P^{e'} \mod n$$

Decrypting C is performed by

$$P = C^{d'} \mod n$$

If d' is not known, an attacker must factor $P^{e'}$ which is difficult and this problem constitutes the basis of the security of RSA.

More in detail, the encryption key e' is a pair of positive integers, (e, n). Similarly the decryption key is the pair of the positive integers (d, n). The selection of these integers are vital for the function of the algorithm. The value of n should be large and is the product of two primes p and q:

$$n = p * q$$

These primes should also be large, typically 100 digits each, so that n is approximately 200 digits. Although n will be made public as part of the key, it has to be large enough to prohibit factoring of n to infer p and q. Then d has to be chosen as a large, random integer which is relatively prime to $(p - 1) * (q - 1)$. That is, d satisfies the following, where *gcd* stands for greatest common divisor:

$$gcd(d, (p - 1) * (q - 1)) = 1$$

Finally e is computed from p, q and d to be the multiplicative inverse of d, modulo $(p - 1) * (q - 1)$:

$$e * d = 1 \mod (p - 1) * (q - 1)$$

Strength of the Algorithm

The underlying problem of factoring large numbers is not known or even believed to be NP-complete, but it is regarded as unlikely that an algorithm would be developed that can solve the problem easily. The problem is well known and has been studied extensively for the past 300 years[RSA78]. The RSA algorithm has been scrutinized intensely since its release and so far cryptologists have found no serious problems with it.

RSA Laboratories has published challenges for breaking RSA encryption. The challenges involve factoring n of a particular size to find the two factors p and q. The intent of the challenges was to track the cutting edge in integer factorization and thus to give an indication of which key sizes that are still safe. The largest number to be factored consisted of 200 decimal digits, or 663 bits, and took 18 months with an unspecified number of computers[PP07].

Chapter 3

Evaluation of Common Cryptosystems

This chapter evaluates the four common cryptography algorithms described in Section 2.5, namely DES, 3DES, AES and RSA. Three aspects are important when deciding which cryptosystem to use for a task: The nature of the task, such as protecting files or key distribution to many users, the performance of encryption and decryption and how secure the cryptosystem is. The evaluation focuses on the performance of the algorithms on home user workstation as well as the security provided.

3.1 Performance

A small program was written to evaluate the performance of the Java implementations of the algorithms. Java was chosen for two main reasons. Java is platform independent and can therefore be the language of choice when encryption is involved in web applications. The second reason was that its cryptography API is well-established and has been in use since 2002.

3.1.1 Test program

The test program was written using Java 1.6. The program allows the user to select a number of algorithms and key sizes which will be applied to a chosen file. The algorithms and key sizes that are implemented are:

- DES - 56 bit key
- 3DES - 168 bit key
- AES - 128 bit key
- AES - 256 bit key
- RSA - 1024 bit key
- RSA - 2048 bit key

Due to U.S export restrictions large key sizes are disabled in the regular Java Runtime Environment but can be enabled by installing appropriate policy files. The algorithms that are affected by this are AES using a 256 bit key and RSA using a 2048 bit key.

3.1.2 Procedure

The algorithms were tested on three common filetypes of various file sizes. Each algorithm was run 10 times for each file in order to produce a consistent measurement. For each file and iteration the run time for encryption and decryption was measured. The median of the measured values are presented in the results of this chapter since the first iteration of each algorithm was almost allways slower than the remaining iterations. The reason for this is probably that Java allocates resources the first time the program is run, imposing a penalty on the performance for the first iteration. The resources is reused for later runs which improves the performance of those iterations.

Initialization and key generation is not included in the measured time. DES, 3DES and AES are tested using Cipher-Block-Chaining (CBC). RSA can not use CBC and is therefore tested using Electronic Codebook (ECB). The following files were used for testing:

- A text file of 59,1 KB.
- A photo of 4,7 MB.
- An Ubuntu CD image of 698,8 MB.

All tests were performed using an AMD Turion 64 with 2 GB of memory. RSA was not tested on the photo and the CD image because of the large size of those files.

3.1.3 Results

Textfile

	DES	3DES	AES-128	AES-256	RSA-1024	RSA-2048
Encryption	16	31	15	15,5	304	1100
Decryption	16	31	15,5	15,5	5538	37736,5

Table 3.1: Execution times of encryption and decryption of a textfile (ms).

Table 3.1 shows the median of the execution times from testing the algorithms on the textfile. DES and the two variants of AES perform almost the same while 3DES takes roughly twice as long time as those. The results may not be accurate since the measured execution times are very low. RSA is many times slower than the symmetric algorithms and it can be seen that decryption requires much more work than encryption.

Photo

The results from applying the algorithms on the photo are displayed in Figure 3.1. The fastest algorithm is AES with a 128 bit key, with AES-256 not far behind. The variants of DES are slower, with 3DES more than twice as slow as the original DES. The results also show that decryption is somewhat slower than encryption for all algorithms. The medians of the test results is displayed in Table 3.2.

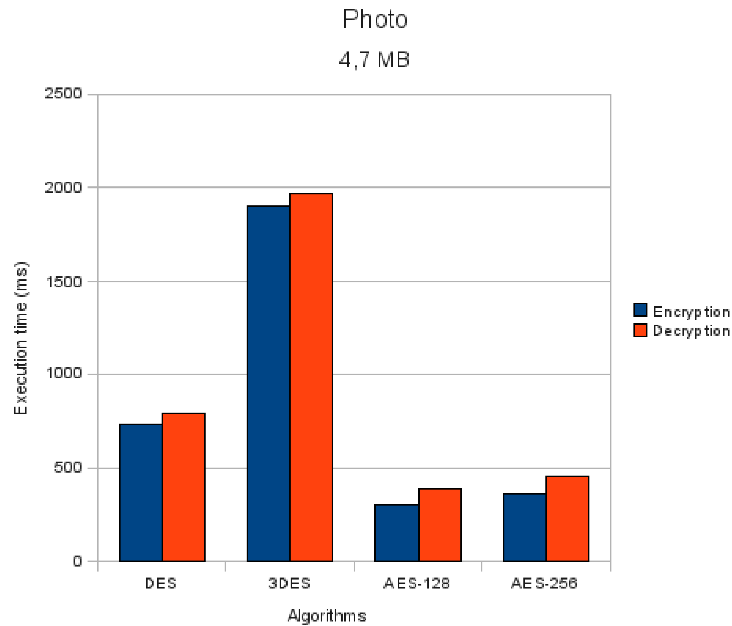

Figure 3.1: Encryption and decryption of a photo.

	DES	3DES	AES-128	AES-256
Encryption	733,5	1903,5	304,5	366,5
Decryption	795	1965	390	452

Table 3.2: Execution times of encryption and decryption of a photo (ms).

CD image

The last test involved encryption and decryption of a large file. The results can be seen in Figure 3.2 and as with the photo, the two versions of AES are the fastest and 3DES is the slowest. Similarly it can be seen that decryption is slower than encryption for all algorithms. Table 3.3 shows the median of the test results. Note that the results are in seconds.

	DES	3DES	AES-128	AES-256
Encryption	131,5	286	76	85
Decryption	140	293	94,5	100,5

Table 3.3: Execution times of encryption and decryption of a CD image (s).

3.2 Security

Of the four cryptosystems that are described in Section 2.5, DES is the oldest cryptosystem and the one that has been subject to the most research. The involvement of

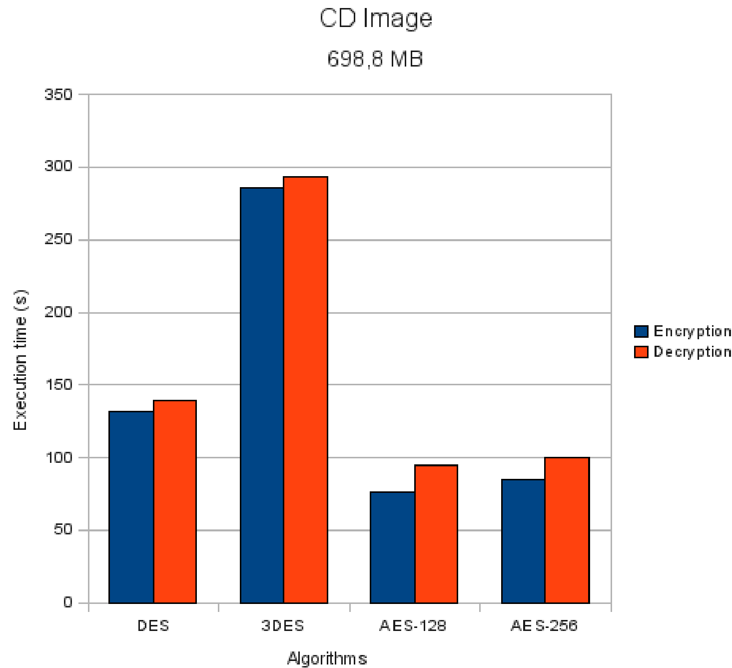

Figure 3.2: Encryption and decryption of a CD Image.

the NSA in the development of the algorithm have for long worried researchers that the algorithm contains a back door to make it easier for NSA to decrypt messages, but so far no such problem has been found. The design documents behind the DES has since been released and nothing indicates that the algorithm contains any weaknesses.

However, DES is still regarded as insecure due the feasibility to simply try all keys to find the correct one in a reasonable amount of time. The attack, performed by the Electronic Frontier Foundation (EFF), on custom built hardware, took only four days to infer the key and showed that a 56 bit key is too short to provide security. This is by far the most serious problem with DES since the key length is fixed and longer keys can not be used. 3DES is a temporary remedy for this problem since it has an effective key length of 112 bits. This length provides enough security to withstand a brute force attack in the near future, but the length is still fixed so it can not be extended should the need arise.

The more modern cryptosystem AES is superior DES regarding key lengths, as the standard constitutes that either 128, 192 or 256 bit keys can be used. Furthermore, the algorithm Rijndael upon which the AES is built, allows for even longer keys, as long as they are multiples of 64. This overcomes the problem with brute force attacks since one could simply choose a larger key if deemed necessary, even if a 128 bit key is large enough to withstand a brute force attack in the foreseeable future. A drawback for AES is that it has not been around for a long time, and thus not been subjected to such extensive cryptanalys as with the DES. But unlike the DES, the specification of the algorithm is publically available and the research that has been performed so far do

show that there are no apparent weaknesses in the algorithm.

The research regarding RSA has so far not found any serious weaknesses of the algorithm. It can not directly be compared to the other cryptosystems in this evaluation since it is based on the difficulty of factoring large numbers and not substitution and permutation. The most prominent danger for RSA is if new methods are discovered for factoring large numbers, but researchers find this unlikely since the topic has been studied extensively for a long time.

A problem for all cryptosystems is the use of keys. There will always be the problem of key distribution, even with public key cryptosystems like RSA. Symmetric cryptosystems rely on another secure channel with which the key can be delivered to both the encryptor and the decryptor. For this asymmetric cryptosystems are used, but there will always be the problem with knowing who actually has control over the key that is used, i.e. a man in the middle attack.

3.3 Conclusion

In this chapter four common cryptosystems have been evaluated by their performance and the security that is provided. Performance tests showed that the fastest algorithm on the test hardware is AES using 128 bit keys, with the same algorithm using 256 bit keys being the second fastest. The difference between these two can be explained by the fact that the algorithm uses four more rounds when encrypting with the larger key. 3DES is about two times slower than the original DES, which is explained by the fact that 3DES is just the original DES applied three times, although optimizations makes the algorithm faster than that. RSA is by far the slowest algorithm and is barely practically usable even with the smallest file of 59,1 KB. However, the intended usage is different for RSA than for the symmetric algorithms. The strength of public key encryption lies in key distribution, which is what RSA should be used for, and not as encryption of files as in the tests. The power of RSA is best utilized if a symmetric algorithm is used to encrypt data and RSA is used to distribute the symmetric key.

Considering the research about these algorithms it can be concluded that DES should not be used anymore since it can be broken by a brute force attack. 3DES adds the needed key length to be secure, but the inability to extend the length and the slow performance makes AES a better candidate for file encryption. Even the smallest key length of AES provides enough security for protecting any data. Another important advantage of AES is that larger keys can be used, making the algorithm a good choice even if computing performance would make 128 bit keys insecure. A minor disadvantage of AES is that it is a rather new algorithm that have not yet stood the test of time, and, however unlikely, it can not be ruled out that a weakness of the algorithm can still be discovered.

A problem that is often not considered when using encryption is selection of keys. The key is often derived from a pass phrase or password provided by the user, since a sentence or a word is much easier to remember than a long series of numbers. This pass phrase must be chosen with care so that an attacker can not guess it easily. It does not matter how secure the algorithm is if an attacker can simply guess the pass phrase and thus bypassing what makes an encryption hard to break; finding the correct key among all that are available. In order to avoid this problem, security policies should be enforced so that the users must choose pass phrases that are hard to guess for an attacker.

Chapter 4

Implementation

The design of `MediaStore` is derived from the functionality provided by the previous system, Mediaarkivet [med], with focus on storage and management of files. Similarly MediaStore is designed to provide the same functionality for PDF documents, with the addition of supporting image previews of a document. Some ideas from the in-depth study, in Chapter 2, are incorporated in the design, such as the ability to encrypt files and the management of passwords.

The design is split into two correlated parts, a code library that contains the functionality of the system, and a database used by the library for storing metadata. This chapter describes the design of the database in Section 4.1 and the design and functionality of the code library in Section 4.2.

4.1 Database

The database in *MediaStore* is used for storage of metadata about files, directories and users. The database server that is currently used is *Microsoft SQL Server 2008*, chosen since it is already in use at Acino, and it integrates well with *Visual Studio 2008*.

The main purpose of the database is to keep track of the files in the system and the users that own the files. Although only one database is used in *MediaStore*, the idea was to keep user information such as usernames and passwords in an additional database in order to separate such information from file information. But to simplify implementation all information is stored in the same database in *MediaStore*.

4.1.1 Files and Directories

Metadata about all files and directories of the system are stored in the table `Files`, which is fully explained in Appendix B.1.3. Each file is given an unique identification number upon insertion.

A similar concept as Unix systems employs is used, where not much difference is made between directories and regular files. A directory is a file that can contain other files and a regular file can be an image or a Pdf document. A file in the database is an entity created and owned by a user, stored somewhere in a filesystem. The boolean attribute `isDir` indicates whether a file is a directory or not. The purpose of making no real distinction between files and directories, such as having two different tables is

to make retrieval of entire sub trees easy and fast. As described in Section 4.1.2, this enables retrieval of all files of a sub tree with only one query.

The database stores a minimal amount of information regarding files as only the name and the file path is stored. Additional information is which user who owns it, and information to represent the file tree. The latter is explained further below, in Section 4.1.2.

4.1.2 Representing the file trees

A common task is to retrieve a full or partial file tree to list the files of a user or the contents of a certain directory. In order to complete such a task in one query the file tree of each user is represented in the database as well. The table `Files` has two additional attributes other than file information, `lineage` and `depth`. The former is the path identification numbers for each directory from a root node to the specified file, separated by a forward slash.

For example, the path /1/4/3/, shows that the file with ID 3 is a child of the file with ID 4, which in turn is a child of the file with ID 1.

The second attribute, `depth`, represents the level in the tree the actual file is situated in, starting with 0 for a root node.

These two attributes are calculated and maintained by the database through triggers, and is enough to represent the filetree. Retrieval of a tree is performed with the stored procedure `FilesGetTree(int ID)`, which is shown in Listing 4.1. The ID of a directory is provided and the procedure returns all files and sub directories contained in it. The result is ordered by the lineage and name attribute, in order to make a depth-first traversal easy when the tree structure is recreated by the library (see Section 4.2.2).

```
SELECT * FROM Files
  WHERE lineage LIKE (SELECT lineage
    FROM Files
    WHERE id = @id) + '%'
ORDER BY lineage, name
```

Listing 4.1: Retrieve a tree or sub tree of files.

4.1.3 Access Control

Access control to files and directories is governed by the table `AccessList`. Each entry represent what access permissions a certain user has on a certain file. The access control model is a simplified version of the model employed in a Unix system, as a user may have write- and read permission on a file. No execute permission is needed since the system is a mean of storing files, not running programs. This table allows for flexible file sharing since any user may have access to any file, but this has been constrained somewhat in MediaStore, as is explained further in Section 4.2.6.

4.1.4 User Information

Originally the idea was to store user- and file information in separate databases. These two were however merged into a single database in order to simplify implementation and usage, and thus user information is kept in two tables, `Users` and `LoginInfo`. Only a minimal amount of information is stored regarding users in order to avoid keeping

information that is not needed for storing files. Each user is assigned a unique identification number upon insertion into the table `Users`. Apart from the ID the table only contain a username for each user. The table `LoginInfo` contain information needed for authentication of a user, such as a 256 bit hash of the user's password and a string used as a password salt. This table is not linked to `Users` by a foreign key relationship since, as mentioned, the original plan was to store this information in a separate database. Identification of a user in `LoginInfo` is made based on username instead of ID.

4.2 Library

The library is the largest part of MediaStore. It is written in C# using LINQ as the means to communicate with the database. This section describes the functionality provided by the library as well as how it is implemented and why a certain approach was chosen.

4.2.1 Design Principles

The library is designed to be mostly focused on file management and leave user- and session management to other systems. User management is restricted to creating users and listing users. The functionality for files is designed to reflect common operations available in file management such as creating, moving and deleting files. A common function not part of the interface is *copy*, not implemented mostly due to time restrictions. If requested there would however not be difficult to incorporate it.

All interaction with files and users is conducted with the identification numbers of those. When calling a typical method of the library the caller has to provide the ID of the current user and the ID:s of any files involved in the operation. The reason to base interaction on identification numbers instead on using the path of files as a string was mainly to improve performance. All relevant tables in the database use integers as primary keys to improve performance, as described in Section 4.1, and thus it was easiest to use the identification numbers in the library as well.

Exceptions are used extensively to signal errors or when a method can not complete successfully, for example when a user tries to access a file without the proper permission or if the file does not exist. Existing exception types in the .NET framework are used if applicable, but on some occasions custom exceptions are employed.

4.2.2 File Management

The largest part of the library is functionality for managing files. Common for all functionality is that the ID number of the current user must be provided in order to control access to files and directories. The database is queried regarding metadata about files and directories and the filesystem is only accessed when a file is to be retrieved, created or modified. The file path of a file is split in two parts, one base path specifying the root directory in which all users' files are stored. The database stores each file's relative path to the base path, so when a file is accessed the full path is created by the library. The purpose of this separation is to make it easy to move all files to a new location, since only the base directory has to be changed and the information in the database can be left intact. The path to the base directory is specified in the configuration file `app.config`.

The general operations for file management are described below, the behavior on shared directories are described more in detail in Section 4.2.6, encryption and decryption of files in Section 4.2.5 and operations on PDF documents is described in Section 4.2.7. The following operations are supported by the library:

Creating directories

A user can create a directory in any directory the user owns, or in any shared directory where the user has write permission. This operation inserts an entry into the file database and creates a directory in the filesystem.

Upload a file

A file can be uploaded to any directory where the user has write permission. The operation is implemented by using streams in order to allow files of any size to be uploaded without using a large amount of memory. An entry is inserted into the table `Files` in the database and then the file is written to the filesystem. Upon insertion into the database the file will be assigned an identification number which will be used in subsequent use of the file.

Download a file

A user may get any file the user has access to. As with uploading of files, this is implemented using streams in order to minimize memory allocation. The library first queries the file database to find the location of the file in the filesystem after which a read only file stream is opened to the file and returned to the caller.

Listing files

The library has methods for listing the files and directories of a certain user, as well as listing shared files and directories available to the user. These methods queries the file database and does not access the filesystem. The methods return objects of the classes `DirectoryInfo` and `FileInfo` which contain relevant information about the files, such as name, ID number and the relative path to the file.

When listing a user's own files the result is a tree structure of the directories, where each directory lists the files and directories contained in it. The decision to return a tree instead of a list was made due to the organization of files and directories in the database, which made it easy to return the result as a tree. Another reason is that the tree structure is a natural way of representing the relationship between directories and files, and is therefore easiest to navigate when presented in such a structure.

When listing files and directories not owned by, but readable by the user, the result is an array of available directories as opposed to the tree structure obtained by listing a user's own files. The reason for this is that the sharing model allows independent sharing of any directory, which results in that there may be no apparent tree structure in the result. Although possible, it was decided not to implement any functionality for recovering tree structures among shared directories due to time constraints.

Moving and renaming

A file or directory can be moved to any other directory where the user has write permission. The same method is used for moving both regular files and directories. Optionally

the user may specify a new name for the object to move, which has the effect that the method can be used to rename objects as well.

The database is queried before the file is moved or renamed to make sure that the new name does not already exist in the specified directory.

Delete

The same method is used for deleting either regular files or directories. In case of a directory all files and directories contained in it are deleted as well. As opposed to the other functions for managing files, delete may only be applied to the files and directories owned by the user, and not to shared files. This is further explained in Section 4.2.6.

Before the actual deletion of a directory from the file database and from the filesystem, all entries referencing the directory in the table `AccessList` are removed. Likewise, all subdirectories is checked in the same manner to make sure that a deleted directory is not listed as shared.

4.2.3 User Management

The library keeps track of its users through the table `Users` in the database. A user has a unique username and is assigned a unique identification number upon creation. This ID is used when interacting with most methods of the library. A user is created by calling the method `createUser` in the class `MediaStorage`. The caller must provide an unused username and a pass phrase from which the password hash will be derived, as described in Section 4.2.4. A directory is created with the same name as the username which will represent the home directory of the user. This directory can not be removed and all files or directories created by the user will be stored in this directory.

The library does not handle user sessions, for example for keeping track of the users that are logged in and those who are not. Session management must be handled by some other software using this library or by some other means. The reason for not implementing session management was that it would unnecessarily constrain the usage of the library and it would take more time than was available to implement.

Currently there exists no methods for modifying or removing users, as that functionality was deemed unnecessary for an prototype. The library does not have any measures for controlling disk quota for users, as the library should only store a minimum of information about users. Disk quota control could instead be delegated to the filesystem or any system that uses the library.

4.2.4 Password Management

Passwords are the main way to authenticate an existing user that want to access the system. As mentioned in Section 4.2.3 a username and a password has to be provided when a new user is created. The password is not stored in clear text in the database, but rather as a 256-bit hash.

Upon creation of the new user the library generates a password salt, a random string of 20 characters, which is appended to the password. The password salt is generated with the class `RandomNumberGenerator`, supplied by the .NET framework. The resulting string consisting of the password and the salt, is then hashed using the algorithm SHA256. The hash is finally stored in the database along with the username and the password salt.

Authentication is performed by providing the username and password to the method `checkPassword` in the class `MediaStorage`. The method retrieves the password salt from the database and creates a hash of the salt appended to the password. If this hash is found to be equal to the hash stored in the database the user has provided the correct password and is considered authenticated.

It was decided to not store the passwords in clear text due to a number of reasons. SHA256 was chosen because it is a well known and well tested algorithm. Although there exists academical attacks on the algorithm the 256-bit hash is sufficiently large to be practically secure [NIS]. The usage of the password salt prevents dictionary attacks against the password. The drawback of not storing the passwords in clear text is that a lost password can not be recovered, but must instead be changed. This was considered a minor drawback and does not motivate clear text storage of passwords.

4.2.5 Encryption

The library support server side encryption and decryption of files, with the methods `encryptFile` and `decryptFile` in the class `MediaStorage`. Based on the evaluation in Section 3, the Advanced Encryption Standard has been chosen as the cryptosystem used in the library. AES is a well tested algorithm that surpass DES and 3DES in both performance and provided security. The library uses 128-bit keys. This size of the key was chosen since the evaluation in Chapter 3 showed that it was faster than using 192- or 256-bit keys and because a brute force attack is still practically infeasible. The performance of the algorithm is important since the system is able to store files of any size, and encryption must be practical even when applied to large files.

An encrypted file gets the extension `.mcrf`. The extension is not needed for the actual decryption but is a way to easily indicate that the file is encrypted. Although encryption and decryption could be performed in place using only the file in question, the library makes use of a temporary file during the process. The original file is deleted as soon as the cryptography has successfully finished. This prevents loss of data if either encryption or decryption would fail during the process. The drawback is that twice the amount of disk space is consumed during the cryptography, which could pose a problem if encrypting very large files.

The library does not store keys in any way and delegates key management to surrounding systems. Encryption and decryption derives a key from a pass phrase which is specified in the call to the methods.

Encryption Header

Upon encryption a 16-byte header is added to the file before the encryption is applied. The layout of the header can be seen in Figure 4.1.

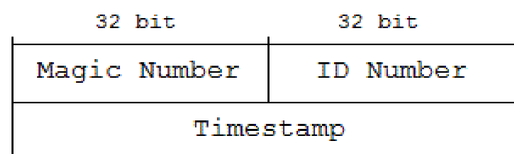

Figure 4.1: Encryption header.

The header consists of a four byte magic number, a four byte number indicating the ID of the file, and an eight byte timestamp. The main benefit of the header is encountered upon decryption. If the first four bytes of the file does not decrypt to the specified magic number it is clear that either the pass phrase is wrong or the file is damaged. This is useful for large files since the system does not have to spend time decrypting data that can not be used anyway. It is also convenient for the user who does not have to wait for the decryption to finish in order to find out that the wrong pass phrase was provided.

Weaknesses

Although the AES cryptosystem is regarded as secure, other parts of the system may introduce weaknesses. The strength of the pass phrase from which the key is derived is not controlled in any way by the library, allowing users to encrypt with phrases that can easily be guessed. This poses the same problem as with passwords, where the system is vulnerable to a dictionary attack. The vulnerability could be controlled with the same measures as described in Section 4.2.4, by concatenating a salt with the pass phrase. This would however require that the library stores these salts somehow, and it was decided that this kind of information should not be the responsibility of the library.

Currently encryption is only applicable on files that are already stored in the system. A user has to upload a file first and then encrypt it. This leaves an attacker an opportunity to access the file after the file has been saved to disk and before the file is encrypted. The methods responsible for cryptography are however designed to be easily integrated with the methods for uploading and downloading files, should the need arise.

4.2.6 File Sharing

The library implements a simple model for file sharing between users to simplify collaborative work. Sharing is governed by an Access Control List (ACL) which is maintained in the database. A user may specify another user's read and write permissions on a directory that is owned by the user.

Sharing model

Three models for file sharing was considered for implementation:

All files: Read and write permissions can be set individually for each file and directory. This model provides the best flexibility but may also requires a very large ACL in the worst case. It would also simplify implementation since no difference is made between directories and regular files.

Subtrees: Access permissions always apply recursively, resulting in read and write permissions set on a directory will apply to all files and directories contained in the directory. This model would be the easiest to implement and would minimize the size of the ACL, but would also provide a minimum of flexibility.

Directories: Permissions can be set on each directory but not on regular files. This is similar to the second model but does not enforce recursive permissions, as each directory is independent. This model maintains an acceptable tradeoff between flexibility and size of the ACL, but would require more time to implement than the other two models.

The third model was chosen for implementation, even though it would require more work than the others. Any access permission set on a directory applies on all regular files contained in the directory. Access permissions are set individually for each directory so that sub directories does not necessarily have the same permissions as its parent directory.

Access permission can only be set to apply for another user, a user can not set which access rights he has on his own directories. A user always has read and write permissions on his own directories and files.

Read Permission

If a user has read permission on another user's directory the user is allowed to list and download any regular file contained in the directory. Some of the functionality for PDF documents may also be applied to a file with read permission, such as creating previews of pages. Note that even if a user do not have write permission on a file there is nothing stopping him from modifying the file once downloaded, since the file is no longer managed by MediaStore.

Write Permission

Write permission in a directory means that the user may create directories and upload files to the shared directory. In order to maintain the integrity of a user's file tree, created directories and uploaded files to a shared directory will automatically be owned by the user that owns the shared directory. Thus the owner of the directory may restrict access to the new directory for the user who actually created it.

Write permission is needed in both the source and destination directory when moving files or directories. Deletion of files and directories, which can be regarded as allowed by a write permission, is however not allowed in the implementation. Only the owner of a file or directory may delete it.

4.2.7 PDF Functionality

The library contains a module for functionality on PDF documents. The functionality is contained in the class `PDFFilter`, and an instance of that class must be obtained by calling the method `getPDFFilter` in `MediaStorage`. PDF functionality is provided by the open source library `itextSharp`. This library was chosen as it is used in Acino's current system, Mediaarkivet [med]. `itextSharp` implements many functions for creating and modifying documents, but does not have capabilities for rendering documents. As this functionality is needed in order to create previews of documents, other libraries were considered. However, no free libraries with easy rendering functions was found satisfactory and it was decided to use the program *GhostScript* for creating previews. This is further described below.

Currently the following functionality is supported by MediaStore:

Joining Documents

This operation joins several documents into one new document. Any number of documents can be joined together and the location of the documents does not matter, as long as the user has permission to read the files. The user must specify in which directory

the new document is to be stored, and the user must have permission to write in that directory.

Splitting Documents

A document can be split into one new document for each page. The new documents may be stored in any directory where the user has permission to write. The new documents are named according to the following:

$[filename]_[pageNr].pdf$

where `filename` is the name of the file and `pageNr` is the number of the page. If the file already exists another $_[pageNr]$ is appended to the filename until an available name is found.

The source document may optionally be moved to another directory where the user has write permission.

Preview

`PDFFilter` can create image previews of any page of a PDF document. The caller specifies the ID of the document and the page number to preview and the image is returned as a byte array. As `itextSharp` does not handle rendering, the open source program GhostScript was chosen for creating previews. When the method is called, MediaStore sets up a string containing the arguments to GhostScript and executes the program as a new process. The resulting image is saved to a temporary file by GhostScript. When the image is created it immediately read into memory by MediaStore and is returned to the caller as a byte array. `PDFFilter` can create previews in either JPG or PNG format, at different resolutions. Changing these settings does however require a recompilation, and currently the previews are in JPG format with 72 dpi resolution.

Although a bit cumbersome, the solution accomplishes its task. Using a dedicated library would be a more clean solution and would probably improve performance somewhat. It is possible to use the GhostScript library and make calls to the `dll` directly, but this approach was disregarded due to a number of reasons. GhostScript is not written in .NET and it would complicate the design and implementation of MediaStore if managed and unmanaged code would be mixed. Secondly, using the `dll` would require some work with licenses. GhostScript uses the GPL license when included in open source projects. If the project is closed in any way, or if the project is a commercial product another license is used, which has to be applied for. Linking against the `dll` would require some work to ensure that the license is not violated, while executing the program as an independent process avoids any issues with licenses.

Get Number Of Pages

It is also possible to retrieve the number of pages a certain document consists of.

Chapter 5

Results

This chapter describes the design of MediaStore in general and how the classes of the system relate to each other. It continues to explain how to configure the system and finishes with a few examples of how to use the functionality of the system in practice.

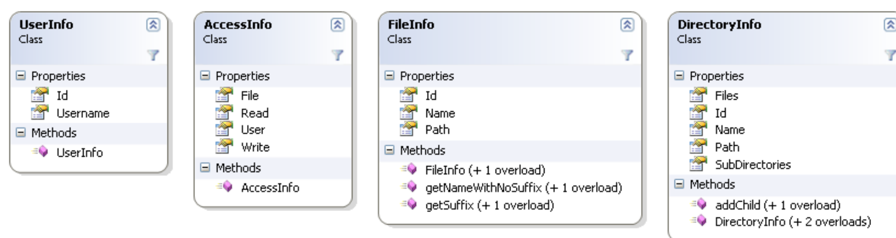

Figure 5.1: Library datastructures.

5.1 Class Overview

Figure 5.2 displays the classes of MediaStore and how they are related. The class `MediaStorage` has public methods that provides access to all functionality of the system. The connection to the database is managed by `FileDBConnector` and `UserDBConnector`, where the former manages all queries regarding files and access permissions and the latter is used for looking up and inserting users. Both these classes connect to the database using `MediaStoreDatabaseContext`, which is a class generated by Visual Studio for accessing the database using LINQ.

Figure 5.3 displays the data structures used by the system. `File, AccessList` and `User` are classes generated, like `MediaStoreDatabaseContext`, by Visual Studio when using LINQ. These classes are data structures that represents one row of data from the database tables `Files, AccessList` and `Users`, respectively. The purpose of these classes is to represent data that is either read or will be written to the database, and is only used internally by MediaStore. When data is presented outside of MediaStore it will be converted to the data structures displayed in Figure 5.1. Classes and public methods are described in detail in Appendix A.

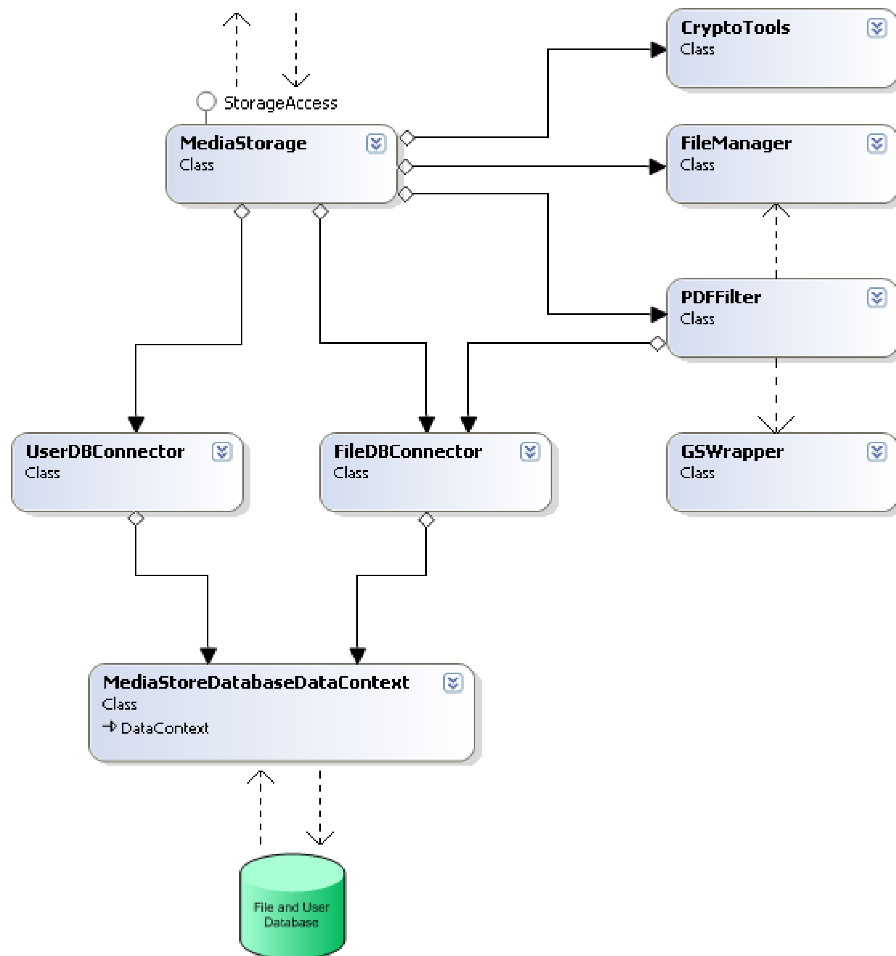

Figure 5.2: Class overview.

5.2 Basic Usage

All functionality of MediaStore is available through two classes, `MediaStorage` and `PDFFilter`. The former class has all functionality for working with users and files in general and the latter class has functionality for PDF documents. In addition to those two there are custom classes for representing information about files, directories and users. The general pattern for using the library is to create an instance of the class `MediaStorage`, retrieve the identification number of the current user with the method `checkPassword` and then perform the desired tasks. Note that MediaStore has no management of sessions, with the result that any method may be run by any user. It is the responsibility of the system using the MediaStore library to ensure that the methods are used properly, for example to ensure that users does not impersonate other users by supplying the wrong identification number.

All methods of `MediaStorage` are independent and no operations explicitly require

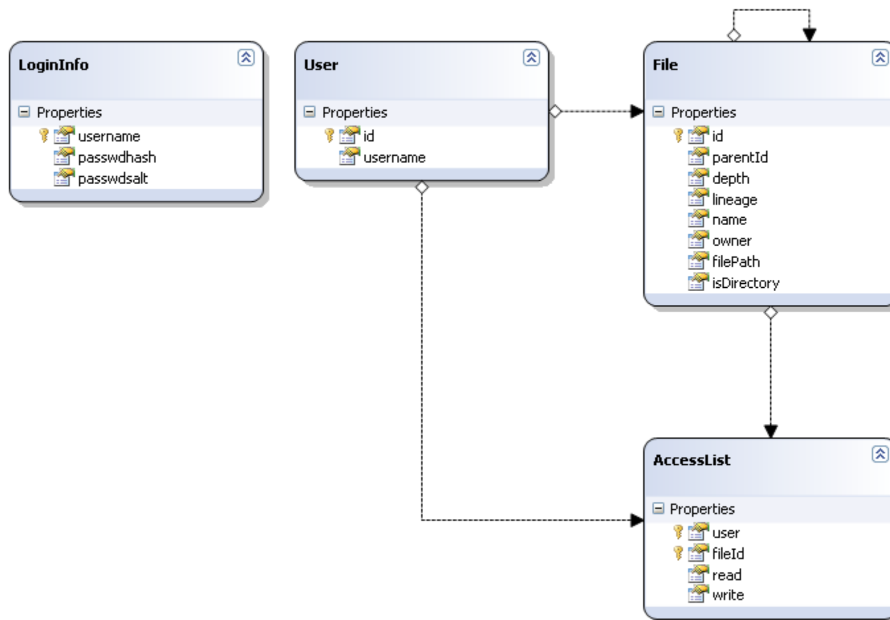

Figure 5.3: Database datastructures.

that another method has been executed before.

5.3 Configuration

MediaStore is configured using the file `app.config`. The following properties must be specified in order for MediaStore to function properly:

Connection String - The connection string specifying which database that is used by the system.

Base File Path - The file path to the directory where MediaStore will store the files of each user.

GhostScript Path - The file path to the directory where GhostScript is located, which is used for creating previews of PDF documents.

GhostScript Binary Name - The name of the binary of GhostScript, typically "gswin32c.exe" on a 32-bit Windows system.

5.4 Code Examples

This sections displays a number of examples of how to use the functionality of MediaStore in practice. Some examples require that necessary data such as users and files already have been inserted into the system. Further are not all exceptions explicitly handled to make the examples shorter.

5.4.1 Authenticate a user

User authentication is handled with the method `checkPassword` in `MediaStorage`. The calling entity supplies a username and a passphrase and receives a `UserInfo` in return upon successful login. Exceptions will be thrown if the username does not exist or if the password does not match. Listing 5.1 shows how to check a user's password and list the user's files.

5.4.2 Listing Files

Most of the methods in the library require the identification number of a file or user as a parameter. The common way to retrieve the ID of an entity is to list the files accessible to a specific user. The ID of a user is retrieved either by the method `getUsers`, which lists all users, or `checkPassword` which checks the password of a certain user and thus serves as a login authentication. Listing 5.1 shows the intended procedure to list a user's files.

```
string username = Console.ReadLine();
string password = Console.ReadLine();
UserInfo ui = null;
DirectoryInfo files = null;
DirectoryInfo[] sharedFiles = null;

MediaStorage m = new MediaStorage();
try {
  // Check the password and username.
  ui = m.checkPassword(username, password);

  // List the files.
  files = m.listFiles(ui.Id);
  sharedFiles = m.listSharedFiles(ui.Id);

} catch(InvalidUsernameException ex) {
  Console.WriteLine("The username is invalid.");
} catch(InvalidPassphraseException ex {
  Console.WriteLine("The password is invalid.");
}
```

Listing 5.1: Checking a the password of a user and listing available files.

5.4.3 Using PDFFilter

Instances of `PDFFilter` is retrieved through `MediaStorage` and each instance can only operate on the files of a certain user. The example in Listing 5.2 shows how to upload a PDF file to the system and then use `PDFFilter` to get a JPG preview of a the first page the document. Then the file is split into a new file for each page in the original document.

```
byte[] preview = null;
string filename = "myPDF.pdf";
FileInfo[] newFiles;
System.IO.FileStream input = System.IO.File.OpenRead(filename);

MediaStorage m = new MediaStorage();
PDFFilter filter = m.getPdfFilter(BOB_ID);

try {
  // Upload the PDF file to Bob's root directory.
  FileInfo file = m.uploadFile(BOB_ID, filename, input, BOB_ROOT_DIR);

  // Get the preview of the first page.
  preview = filter.getPreview(file.Id, 1);

  // Split the document
  newFiles = f.split(file.Id, BOB_ROOT_DIR);
}
catch (ArgumentException ex) {
  Console.WriteLine("The file is not a PDF-document.");
}
catch (NoSuchFileException ex) {
  Console.WriteLine("The file does not exist.");
}
catch (AccessDeniedException ex) {
  Console.WriteLine("Can not access the file.");
}
```

Listing 5.2: Using PDF functionality.

5.4.4 Encrypting a file

Files stored in the system may be encrypted and decrypted using the methods `encryptFile` and `decryptFile` in `MediaStorage`. Listing 5.3 shows how to encrypt a file and copy it to a new file on the hard drive. The example then proceeds to decrypt the file in Listing 5.4.

```
string KEY = "encryption_key";

MediaStorage m = new MediaStorage();
try {
  // Encrypt the file
  m.encryptFile(BOB_ID, MY_FILE_ID, KEY);

  // Get the encrypted file and copy it to a new file
  System.IO.FileStream fileStream = m.getFile(BOB_ID, MY_FILE_ID);
  System.IO.FileStream outStream = System.IO.File.Create("
      my_encrypted_file");

  byte[] data = new byte[fileStream.Length];

  fileStream.Read(data, 0, data.Length);
  outStream.Write(data, 0, data.Length);

  fileStream.Close();
  outStream.Close();

}
catch (Exception ex) {
  Console.WriteLine(ex.Message);
}
```

Listing 5.3: Encryption of a file.

```
try {
  m.decryptFile(BOB_ID, MY_FILE_ID, KEY);
}
catch (System.Security.Cryptography.CryptographicException ex) {
  Console.WriteLine("File is not encrypted.");
}
catch (InvalidPassphraseException ex) {
  Console.WriteLine("Wrong passphrase.");
}
```

Listing 5.4: Decryption of a file.

Chapter 6

Conclusions

This project has resulted in a prototype for an online storage system, called MediaStore. It is designed as an alternative to the system *Mediaarkivet*. MediaStore provides essential functionality for managing files but also the same functionality for PDF documents as currently used by Acino. As specified in the goals for the project (see Section 1.4), MediaStore can create previews of any page of a document as well, although the implementation did not go according to plan. The use of the external program *GhostScript* is a bit cumbersome, and using a dedicated library for creating previews would be preferable. The design of MediaStore allows for other modules capable of creating previews of other types of files as well but currently only PDF documents can be previewed.

MediaStore can encrypt and decrypt stored files using AES with 128-bit keys, as this cryptosystem proved to provide the best combination of performance and security.

Using Microsoft programs for all parts of the system has been a new experience, and the integration between the IDE Visual Studio and the database made development and testing easy. LINQ is used as the means to communicate with the database and has mostly been a positive experience. LINQ eliminates the possibility of runtime errors regarding SQL queries which makes interaction with the database more stable. Using LINQ also prevents SQL injection attacks.

Overall the work has progressed according to plan although the report consumed a lot more time than planned. Some redesigning of both database and the library has occurred several times, as the original design made MediaStore a bit to extensive and would require much more time than allotted to a master's thesis. As the project progressed the design became more simplified than originally planned, which focused the functionality of MediaStore on file management.

6.1 Limitations

Many limitations of MediaStore was introduced as the design progressed towards a more simple and specialized software, mainly focused on the management of files. A large limitation of MediaStore is the lack of session management, which was decided to be left out in an early stage. Although this decision simplified the design and functionality of MediaStore, the management of sessions is then effectively delegated to any wrapping system using the MediaStore library. An effect of the lack of sessions is that the method for authenticating the password of a user does not have to be called, allowing any user to access the system without being authenticated.

However, the lack of session management do make MediaStore more flexible as it can be customized to be used by whatever methods that are preferred by any system using it, for example as a web service or in a desktop application.

Another limitation is the lack of user management which is also a consequence of focusing on files. The library has the ability to create users, but can not remove or modify existing users. There are no means for controlling disk quotas and disk usage in MediaStore, as such functionality is delegated to either the filesystem or any wrapping system.

A common function not included in the interface of the library is the ability to make copies of files and directories. Although that functionality is not currently used by Acino's systems, it is a common and important function.

MediaStore has no concept of transactions when changes are made to both the database and the filesystem, for example when moving a file. Thus the system will be inconsistent if a crash occurs after the changes are written to the database and before the file is moved to the new location in the filesystem. Another limitation imposed by the separation of metadata in the database and the actual files in the filesystem is that files must be inserted using the library. Files can not be inserted to the filesystem using FTP or any other means that bypasses the MediaStore library since the system will be inconsistent. If FTP access is deemed necessary one solution might be to have a separate program running on the file server that automatically controls that new files are actually in the database, and if not inserts the correct entry into the database.

MediaStore is limited to Windows platforms as it is written entirely using C# .NET. There exists open source projects that implements and complies with the .NET framework running on other platforms than Windows, but MediaStore is currently using Microsoft's framework. A limitation imposed by using of LINQ To SQL is that MediaStore must eploy a supported database manager, which is currently only Microsoft SQL Server. MediaStore must be rewritten to use LINQ To Datasets in order to communicate with other database managers.

Regarding the database schema, it can always be argued whether large enough datatypes are used in the tables. The unique identification number of a file is a signed 32-bit integer, which restricts the number of files to 2^{32-1}. Note that this is not the number of files allowed at a single moment, but is the total number of files that has been added to the system. As this identification is incremented for each file added to the system the upper limit will eventually be reached. Another example is that the name or the path of a file can not be longer than 255 characters. The solution to these problems are of course to increase the size of the datatypes, but knowing what is large enough without wasting storage space is a difficult problem.

6.2 Future Work

Several of the limitations discussed in Section 6.1 can be subject to future work, where user management is the most pressing issue. MediaStore can not be expected to be used in a live environment without proper functionality for managing users, such as removing or modifying user, or controlling disk quotas. Another approach may be to remove current functionality from MediaStore and have all user management in a separate project.

Another important issue for future work is to implement some concept of transactions to avoid inconsistencies in the event of a crash. An alternative to transactions could be some recovery procedure after a crash, correcting any inconsistencies.

Chapter 7

Acknowledgements

I want to thank my supervisor at CS, Thomas Johansson, for his constructive criticism on the project and for his advice on the structure and contents of this report. I also want to thank the personnel at Acino, Dan Viktorsson for the offer of doing the master's thesis at Acino, my supervisors, Robert Bjuhr and Magnus Stålåker, for their ideas for defining the thesis as well as their help throughout the project.

I want to acknowledge the help of Annika Jonsson and Emelia Stening for proof reading this report and lastly my family for numerous reasons.

References

[AGRS04] Giuseppe Amato, Claudio Gennaro, Fausto Rabitti, and Pasquale Savino. *Milos: A Multimedia Content Management System for Digital Library Applications*, volume 3232/2004 of *Lecture Notes in Computer Science*, chapter 2, pages 14–25. Springer Berlin / Heidelberg, 2004.

[And01] Ross Anderson. *Security Engineering*. John Wiley & Sons, Inc, first edition, 2001.

[box] Box.net - Online Storage. `http://www.box.net`. Accessed 2009-03-16.

[BS90] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. In *Journal of Cryptology*, volume 4, pages 3–72, January 1990.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer-Verlag New York, LLC, 2002.

[FKL+00] Niels Feguson, John Kelsey, Stefan Lucks, Bruce Schneier, Mike Stay, David Wagner, and Doug Whiting. *Improved Cryptanalysis of Rijndael*. Springer-Verlag, 2000.

[ISB] Cracking DES: Secrets of Encryption Research, Wiretap Politics, and Chip Design. `http://jya.com/cracking-des/cracking-des.htm`. Accessed 2009-03-24.

[med] Mediaarkivet. `https://www.mediaarkivet.se`. Accessed 2009-03-16.

[MH81] Ralph Merkle and Martin Hellman. On the security of multiple encryption. In *Communications of the ACM*, volume 24, pages 465–467, July 1981.

[NIS] NIST. NIST comments on cryptanalytic attacks on SHA-1. Accessed 2009-11-30 `http://csrc.nist.gov/groups/ST/hash/statement.html`.

[NIS01] NIST. *Specification for the Advanced Encryption Standard*, 2001. FIPS 197.

[PP07] Charles P. Pfleeger and Shari Lawrence Pfleeger. *Security in Computing*. Upper Saddle River, NJ : Prentice Hall, fourth edition, 2007.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[Sch00] Bruce Schneier. Self-study course in block cipher cryptanalysis. *Cryptologia*, 24(1):18–34, January 2000.

[WH77] Diffie Whitfield and Martin Hellman. Exhaustive Cryptanalysis of the NBS Data Encryption Standard. In *Computer*, volume 10, pages 74–84, June 1977.

[Whi88] Diffie Whitfield. The first ten years of public-key cryptography. In *Proceedings of the IEEE*, volume 76, may 1988.

Appendix A

User's Guide

This document describes how to use the functionality of MediaStore. Classes and interfaces are described as well as examples on how to use the functionality programmatically.

A.1 Overview

This document is a simple userguide to MediaStore, a library of functions that enables storage and access to any type of files. The library has basic management of users, common operations on files such as create, move and delete, as well as some more specialized functionality on PDF documents. Users are able to share files with other users, by specifying write and read permissions on directories.

Metadata about files and users are stored in a database, currently SQL Server 2008. Access to the database from the library is frequent and necessary for many of the methods in the interface, but is completely hidden from any system using MediaStore.

MediaStore is a plain library and does not involve session- or user management other than creation of users and authentication of user passwords. The library is also thread safe.

The library is designed to not be limited to a certain type of client program, but could be used by either a desktop program or part of a web service.

A.2 Technology

- The library is implemented in C# using .NET 3.5.
- MediaStore uses the database management system SQL Server 2008 Express to store information. Files are stored in the filesystem and not in the database.
- LINQ is used by the library for communicating with the database.
- iTextSharp is used for managing Pdf documents, while GhostScript renders previews of pages.

A.3 Basic Usage

All functionality of MediaStore is available through two classes, `MediaStorage` and `PDFFilter`. The former class has all functionality for working with users and files in general and the latter class has functionality for PDF documents. In addition to those two there are custom classes for representing information about files, directories and users. The general pattern for using the library is to create an instance of the class `MediaStorage`, retrieve the identification number of the current user with the method `checkPassword` and then perform the desired tasks. Note that MediaStore has no management of sessions, with the result that any method may be run by any user. It is the responsibility of the system using the MediaStore library to ensure that the methods are user properly, for example to ensure that users does not impersonate other users by supplying the wrong identification number.

All methods of `MediaStorage` are independent and no operations explicitly requires that another method has been executed before.

A.4 Configuration

MediaStore is configured using the file `app.config`. The following properties must be specified in order for MediaStore to function properly:

Connection String - The connection string specifying which database that is used by the system.

Base File Path - The file path to the directory where MediaStore will store the files of each user.

GhostScript Path - The file path to the directory where GhostScript is located, which is used for creating previews of PDF documents.

GhostScript Binary Name - The name of the binary of GhostScript, typically "gswin32c.exe" on a 32-bit Windows system.

A.5 Code Examples

This sections displays a number of examples of how to use the functionality of MediaStore in practice. Some examples require that necessary data such as users and files already have been inserted into the system. Further are not all exceptions explicitly handled to make the examples shorter.

A.5.1 Authenticate a user

User authentication is handled with the method `checkPassword` in `MediaStorage`. The calling entity supplies a username and a passphrase and receives a `UserInfo` in return upon successful login. Exceptions will be thrown if the username does not exist or if the password does not match. Listing A.1 shows how to check a user's password and list the user's files.

A.5.2 Listing Files

Most of the methods in the library require the identification number of a file or user as a parameter. The common way to retrieve the ID of an entity is to list the files accessible to a specific user. The ID of a user is retrieved either by the method `getUsers`, which lists all users, or `checkPassword` which checks the password of a certain user and thus serves as a login authentication. Listing A.1 shows the intended procedure to list a user's files.

```
string username = Console.ReadLine();
string password = Console.ReadLine();
UserInfo ui = null;
DirectoryInfo files = null;
DirectoryInfo[] sharedFiles = null;

MediaStorage m = new MediaStorage();
try {
  // Check the password and username.
  ui = m.checkPassword(username, password);

  // List the files.
  files = m.listFiles(ui.Id);
  sharedFiles = m.listSharedFiles(ui.Id);

} catch(InvalidUsernameException ex) {
  Console.WriteLine("The username is invalid.");
} catch(InvalidPassphraseException ex {
  Console.WriteLine("The password is invalid.");
}
```

Listing A.1: Checking the password of a user and listing available files.

A.5.3 Using PDFFilter

Instances of `PDFFilter` is retrieved through `MediaStorage` and each instance can only operate on the files of a certain user. The example in Listing A.2 shows how to upload a PDF file to the system and then use `PDFFilter` to get a JPG preview of a the first page the document. Then the file is split into a new file for each page in the original document.

```
byte[] preview = null;
string filename = "myPDF.pdf";
FileInfo[] newFiles;
System.IO.FileStream input = System.IO.File.OpenRead(filename);

MediaStorage m = new MediaStorage();
PDFFilter filter = m.getPdfFilter(BOB_ID);

try {
  // Upload the PDF file to Bob's root directory.
  FileInfo file = m.uploadFile(BOB_ID, filename, input, BOB_ROOT_DIR);

  // Get the preview of the first page.
  preview = filter.getPreview(file.Id, 1);

  // Split the document
  newFiles = f.split(file.Id, BOB_ROOT_DIR);
}
catch (ArgumentException ex) {
  Console.WriteLine("The file is not a PDF-document.");
}
catch (NoSuchFileException ex) {
  Console.WriteLine("The file does not exist.");
}
catch (AccessDeniedException ex) {
  Console.WriteLine("Can not access the file.");
}
```

Listing A.2: Using PDF functionality.

A.5.4 Encrypting a file

Files stored in the system may be encrypted and decrypted using the methods `encryptFile` and `decryptFile` in `MediaStorage`. Listing A.3 shows how to encrypt a file and copy it to a new file on the hard drive. The example then proceeds to decrypt the file in Listing A.4.

```
string KEY = "encryption_key";

MediaStorage m = new MediaStorage();
try {
  // Encrypt the file
  m.encryptFile(BOB_ID, MY_FILE_ID, KEY);

  // Get the encrypted file and copy it to a new file
  System.IO.FileStream fileStream = m.getFile(BOB_ID, MY_FILE_ID);
  System.IO.FileStream outStream = System.IO.File.Create("
      my_encrypted_file");

  byte[] data = new byte[fileStream.Length];

  fileStream.Read(data, 0, data.Length);
  outStream.Write(data, 0, data.Length);

  fileStream.Close();
  outStream.Close();

}
catch (Exception ex) {
  Console.WriteLine(ex.Message);
}
```

Listing A.3: Encryption of a file.

```
try {
  m.decryptFile(BOB_ID, MY_FILE_ID, KEY);
}
catch (System.Security.Cryptography.CryptographicException ex) {
  Console.WriteLine("File is not encrypted.");
}
catch (InvalidPassphraseException ex) {
  Console.WriteLine("Wrong passphrase.");
}
```

Listing A.4: Decryption of a file.

A.6 Class Information

This section describes the important classes that are part of the interface of MediaStore.

A.6.1 FileInfo

Describes the properties of a regular file stored in the system. All files except directories are regular files and is represented by an identification number, a name and a file path, relative the base file path controlled by MediaStore.

Methods

- `string getNameWithNoSuffix()`
  Get the name of the file without the trailing suffix, i.e. the characters after the last punctuation is removed, as well as the punctuation.

– `string getSuffix()`
  Get the suffix of the file name, i.e. the string following the last punctuation in the name.

Properties

– `Id` - Get or set the identification number of the file.

– `Name` - Get or set the name of the file.

– `Path` - Get or set the file path. The path is relative the base file path which is specified in the configuration of MediaStore.

A.6.2 DirectoryInfo

Represents a directory, and may contain information about any files and directories contained in it.

Methods

– `void addChild (DirectoryInfo dir)`
  Add a sub directory to this directory.

– `void addChild (FileInfo file)`
  Add a regular file to this directory.

Properties

– `Files` - Get a `LinkedList<FileInfo>` containing all files contained directly in this directory.

– `SubDirectories` - Get a `LinkedList<DirectoryInfo>` containing the directories stored in this directory. If no directories exists, null is returned.

– `Id` - Get or set the identification number of the directory.

– `Name` - Get or set the name of the directory.

– `Path` - Get or set the file path. The path is relative the base file path which is specified in the configuration of MediaStore.

A.6.3 UserInfo

Contains relevant information about a user.

Properties

– `Id` - Get the identification number of the user.

– `Username` - Get the user name.

A.6.4 AccessInfo

This class describes what access permissions a user has on a certain file.

Properties

- `File` - Get information about the file.
- `User` - Get information about the user.
- `Read` - Check whether the user has permission to read the file.
- `Write` - Check whether the user has permission to write to the file.

A.6.5 MediaStorage

The following methods are part of the interface implemented by the class MediaStorage.

- `UserInfo checkPassword (string username, string passphrase)`
  Login a user to the system by checking that the username and passphrase match what is stored in the User Database.
- `DirectoryInfo listFiles(int userId)`
  List all files and directories that are owned by a specific user.
- `DirectoryInfo[] listSharedFiles(int userId)`
  List all shared files accessible to a user.
- `UserInfo createUser(string username, string passphrase)`
  Creates a new user. Entries are created in the User database and an empty directory is created in the filesystem with the same name as the username.
- `FileInfo uploadFile(int userId, string name, Stream inputStream, int directory)`
  Upload a file to the system. The file will get an entry in the database and will be copied to the filesystem.
- `FileStream getFile(int userId, int fileId)`
  Get a file from the system.
- `DirectoryInfo createDirectory(int userId, int parentDirectory, string name)`
  Creates a new directory in the specified directory.
- `void delete(int userId, int fileId)`
  Delete a file or directory. If a directory is deleted, the contents of the directory is also deleted.
- `void move(int userId, int fileId, int toDirectory, string newFilename)`
  Move a file or directory to another directory. Only the owner of a file or directory is allowed to move it.
- `byte[] getPreview(int userId, int fileId)`
  Convenience method for creating a preview of a file. If the file is in a format known by the library a JPG preview will be created and returned, else null is returned.
- `PDFFilter getPdfFilter(int userId)`
  Get an instance of PDFFilter, which enables functionality on PDF documents.

- `void setPermissions(int userId, int shareWithUser, int directoryId, bool read, bool write, bool recursive)`
  Set another user's access permissions on a directory.

- `UserInfo[] getUsers()`
  List all users present in the system.

- `AccessInfo getAccessInfo(int userId, int forUser, int fileId)`
  Gets the access permissions a certain user has on a certain file.

- `void encryptFile (int userId, int fileId, string passphrase)`
  Encrypt a file with the specified passphrase. The encrypted file will get the extension 'mcrf'. A temporary file is used for encryption, so that the original file is still available if an error occurs before the encryption is completed.

- `void decryptFile (int userId, int fileId, string passphrase)`
  Decrypt a file with the specified passphrase. A temporary file is used for decryption, so that the original file is still available if an error occurs before the decryption is completed.

A.6.6 PDFFilter

The following methods are provided by the class `PDFFilter`.

- `int getNumberOfPages(int fileId)`
  Get the number of pages a PDF document consists of.

- `byte[] getPreview(int fileId, int page)`
  Creates a preview of a page of a PDF document.

- `FileInfo[] split (int fileId, int destDirectory)`
  Splits a document into a new file for each page.

- `FileInfo[] split (int fileId, int destDirectory, int moveSourceToDirectory)`
  Splits a document into a new file for each page, and stores the new files in the specified directory. The source file is moved to a specified directory.

- `int join (int[] fileIds, int destDirectory, string name)`
  Joins several PDF documents into one new document.

A.7 Internal Classes

This section briefly describes the internal classes used by MediaStore but are not part of the interface.

A.7.1 CryptoTools

Has methods for cryptographic purposes. Provides encryption and decryption capabilities using AES with 128-bit keys, by creating streams that can be applied to another stream, such as a file stream. Another functionality is creation of hashes using SHA-256, used by MediaStore for managing passwords.

A.7.2 UserDBConnector

Connects to the database and provides functionality for creating and listing users.

A.7.3 FileDBConnector

Connects to the database and has methods for managing files. Can list existing files accessible by a specific user as well as inserting a new file entry into the database. Other functionality includes common operations on files, such as moving, deletion and renaming of files. Note that this class only makes changes to the information stored in the database, and thus no operations makes changes in the actual filesystem.

A.7.4 FileManager

The class `FileManager` gives MediaStore access to the filesystem. The class can open file streams to stored files as well as save files that are to be stored in the system. Other common operations are supported such as moving, deleting and renaming files and directories.

A.7.5 GSWrapper

Enables the functionality of creating JPG previews of PDF documents. Executes the program GhostScript which creates the preview. `GSWrapper` returns the created image to the caller.

Appendix B

Database Design

The database contains information about the users of the system, including references to all files and schema for access control. Files are stored in the filesystem, the database contain metadata about files. This provides better performance, reduces the size of the database and allow access to files by other means (e.g. sftp). The references in the database must however be updated if files are uploaded using ftp, in order to keep the information consistent with the filesystem.

B.1 Schema

This section describes each table and attributes of the database schema. An underlined attribute indicates that it is a primary key.

B.1.1 Users

This table contains information about the registered users of the system.

Attributes

Attribute	<u>id</u>	username
Type	INT	VARCHAR

Table B.1: Attributes of the table Users.

id A unique integer that identifies each user. Primary key of this table.

username The username of the user.

B.1.2 LoginInfo

Contains all information necessary for authentication of a user.

Attributes

username The chosen username of the user.

passwdhash This is the SHA-256 hash of the user's password concatenated with `passwdsalt`.

Attribute	username	passwdhash	passwdsalt
Type	VARCHAR	BINARY(32)	VARCHAR

Table B.2: Attributes of the table LoginInfo.

passwdsalt A random string that is created upon registration. Is concatenated with the user's password and then hashed using SHA-256 to create the password hash, `passwdhash`.

B.1.3 Files

The table Files contain entries of all files and directories that are stored in the system. The columns `parentId`, `depth` and `lineage` are used to represent the file tree in which the files reside. Maintaining these columns are done by the database using triggers and stored procedures, and does not have to be considered by the calling system.

Attributes

Attribute	id	parentId → Files:id	depth	lineage
Type	INT	INT	INT	VARCHAR
Attribute	name	owner → Users:id	filePath	isDirectory
Type	VARCHAR	INT	VARCHAR	BIT

Table B.3: Attributes of the table Files.

id A unique integer that identifies each file.

parentId The id of the directory that the file resides in. Foreign key from `id` in the table Files. Can be NULL if this file is the root of a tree.

depth The depth of the file in the file tree.

lineage The path of files from the root to this file.

name The name of the file.

owner The id of the user that owns the file. Foreign key from the table Users.

filePath The path to the file where it is located in the filesystem.

isDirectory True if the entry is a directory. False if it is a regular file.

B.1.4 AccessList

AccessList is a table that describe access permissions to files by users, thus describing which files that are shared between users. Each row describes what permissions a user has on a certain directory. All regular files contained in a shared directory are implicitly shared.

Attributes

user The id of the user. Foreign key from table Users.

file The id of the file in question. Foreign key from table Files.

Attribute	user → Users:id	fileId → Files:id	read	write
Type	INT	INT	BIT	BIT

Table B.4: Attributes of the table AccessList.

read A bool that indicates whether the user can read this file or not.

write A bool that indicates whether the user can write to this file or not.

B.2 Stored Procedures

FilesAddNode - Add a file to the table `Files`. The attributes `depth` and `lineage` are calculated upon insertion. The identity of the inserted file is returned by the procedure.

FilesMoveFile - Move a file to a new location. Optionally the file may be renamed as well.

FilesDeleteNode - Delete a file, and in the case of a directory, delete all files contained in the directory.

FilesGetChildren - Get all files that are children of a directory, i.e. have `parentId` set to the identification number of the directory.

FilesGetPath - Retrieve the path from a root directory to the specified file.

FilesGetTree - Get the file tree contained under the specified directory, including the directory itself.