Key Management in Cryptographic Access Control

Simon Thyregod
2nd March 2006

Kongens Lyngby 2006
IMM-MSC-2006-23

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-MSC: ISSN 0909-3192

Abstract

This thesis project is a part of the CryptoOS project, where cryptographic access control methods are used to replace traditional access control mechanisms. This provides a truly decentralised design, where access is not related to users, but to the possession of cryptographic keys. By using a combination of symmetric and asymmetric encryption it is possible to differentiate access to three levels: Full-, read- and "integrity check" authorisation. The "integrity check" authorisation provides the possibility to verify the integrity of data, without knowing its actual contents. This makes it possible to store confidential data on less trustworthy file servers, which only can verify the integrity of the data, and not learn its contents.

Each file has a key, which is used to gain access to the file. These keys are collected in key rings, which provide the possibility to form different access control models. To handle the key rings and demonstrate the proposed design a key managing application and different types of servers has been implemented and evaluated. The implementation proofs that it is possible to use cryptographic access control to replace existing access control mechanisms. The performance is a bit slower than traditional models, but the overall security is increased and the design is completely decentralised and scalable, which usually requires a lot more computations.

Key Words: Cryptography, Key Management, Cryptographic Access Control, Distributed Systems, CryptOS, Cryptographic Key Rings, Distribution of Cryptographic Keys.

Resumé

Dette projekt omhandler nøglehåndtering i kryptografisk adgangskontrol. Projektet en del af CryptOS projektet, hvor kryptografiske adgangskontrolmetoder benyttes til at erstatte den traditionelle adgangskontrol mekanisme. Dette giver et fuldstændigt decentraliseret design, hvor adgange ikke er knyttet til brugere, men til besiddelsen af kryptografiske nøgler. Ved benyttelse af en kombination af symmetrisk og asymmetrisk kryptering, er det muligt at differentiere adgange på tre niveauer: fuld-, læse- og "integritetskontrol"-adgang. Integritetskontrol-adgangen giver mulighed for at verificere korrektheden af data, uden at kende til dets indhold. Dette kan benyttes til at gemme hemmelige data på mindre troværdige filservere, der kun kan kontrollere integriteten af data, og altså ikke lære dets indhold.

Til hver fil findes en nøgle, der benyttes til at få adgang til filen. Disse nøgler er samlet i nøgleringe, der giver mulighed for at danne forskellige adgangskontrol modeller. Til at håndtere nøgleringene og demonstrere den foreslåede model er et nøglehåndteringsprogram, samt diverse servere, implementeret og evalueret. Implementering viser at det et muligt, at benytte kryptografisk adgangskontrol i stedet for de eksisterende adgangskontrol mekanismer. Hastigheden er en smule langsommere end traditionelle modeller, men til gengæld opnås større sikkerhed og et fuldstændigt decentraliseret og skalerbart design, der traditionelt kræver betydelig flere beregninger.

Nøgleord: Kryptografi, nøglehåndtering, kryptografisk adgangskontrol, distribuerede systemer, CryptOS, kryptografiske nøgleringe, distribution af kryptografiske nøgler.

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the master of science degree in engineering. The project is nominated to 30 ECTS points equivalent to five months full time study.

I declare that the work described in this thesis is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Lyngby, March 2nd, 2006

Simon Thyregod

Acknowledgements

I would like to thank my supervisor Christian D. Jensen for his help, positive critissim and discussion of ideas throughout the project. I would further like to thank my girlfriend Luise C. Bendixen for supporting me through the last six months and discussing ideas in the project.

Finally I would like to thanks David Rong Dai for helping me with the final proof reading of the project.

Contents

Chapter 1

Introduction

Today's access control methods usually require a centralised user administration. In many cases this is sufficient, but it has some shortcomings for a distributed operating or file system. Traditional file servers need to be trustworthy before users will allow them to handle secret information, i.e. to store their files. If the user encrypts the files to protect their contents from the server, other users will not be able download and read the information. If the file servers need to maintain access rights for files, it needs to be able to verify the users identity. This can be done with certificates or a user login.

Both solutions are often time consuming because the server needs to contact a central user database or certificate authority and perform computational intensive cryptographic tasks such as public-private key (asymmetric) encryption. Furthermore, the underlying data network can be untrustworthy and it is, therefore, necessary to encrypt the traffic between the client and the server. This occupies server resources and limits the total number of client that can be served.

In this project, a cryptographic access control mechanism will be combined with key management to replace existing access control mechanisms and model access control schemes. The basic idea of cryptographic access control is to encrypt files using a special cryptographic key scheme before transmission to the server. Because of the cryptographic scheme, the server is not able to decrypt files, but can only check their integrity. The files stay encrypted on the server and

can, therefore, be transmitted on the network without further encryption when clients request the files. The cryptographic key scheme allows differentiated keys, so users can be allowed to either read and write to a file, read it or only check its integrity. The cryptographic keys represent a capability and can serve purposes other than protecting files. For instance, they can be used for secure requests to web services or remote function calls. Because they represent capabilities, they can be given to servers allowing them to perform actions on behalf of the user. Key distribution and access control schemes are handled by key rings[1], which contain sets of access rights (capabilities). The proposed solution separates the access control mechanism from the access control scheme, and therefore, it is possible to model different access control schemes on top of the cryptographic access control mechanism.

This thesis will first describe the basics of cryptography and the idea behind cryptographic access control and its key management. A design will then be proposed and implemented in the `C` programming language. The choice of programming language was a requirement to the implementation because the thesis is a part of the CryptOS project[Cry06]. The implementation consists of a Key Manager, which is the core of the project, a command Shell, which uses the Key Manager, different types of servers, and a set of cryptographic functions. The final part of the project will evaluate the implementation, analyse its security aspects and conclude the entire thesis project.

The implementation is a fully functional proof-of-concept for key management in cryptographic access control. The main part of the work is concentrated around the Key Manager and its communication with the servers. The evaluation shows that the described theory can be used to build a distributed operating system or file sharing application on the principals discussed in this thesis.

1.1 Problem Description

The problem description was formulated during the initial phase of the project. It states:

This thesis project is undertaken in the context of the CryptOS project. The overall goal of the CryptOS project is to explore the applicability of cryptographic access control to all aspects of operating system security. Cryptographic access control offers a completely decentralised access control abstraction, which improves scalability and makes it attractive for large scale open distributed systems, e.g., ubiquitous/pervasive computing and computational grids.

[1] The initial key distribution is not handled.

The goal of this thesis project is to design, implement and evaluate a general key management infrastructure applicable for CryptOS. Previous proposals for key management in cryptographic access control have either been ad hoc [Sha][Har01] or application specific [HK03]. This project will investigate a more general key management infrastructure that provides a unified framework for management of keys that are used by different applications, i.e. the protected resources could be local files, remote files, remote functions calls, etc. The implementation should treat these resources in a similar manner from the user's point of view, and allow individual mapping of their resources.

1.2 Structure of Report

This thesis contains six chapters, three appendices and a CD-ROM:

Cryptographic Background
: This chapter describes the computer security concepts necessary for understanding the following chapters. It describes the overall design of the chosen algorithms and their security properties.

Cryptographic Access Control
: This chapter describes the idea behind cryptographic access control and previous work in this field.

Key Management
: This chapter analyses key management for cryptographic access control, and describes how different access control schemes can be modelled using key rings and cryptographic access control.

Design
: This chapter proposes a design for a Key Manager, servers, cryptographic functions and a Shell, which can be used to access the Key Manager.

Implementation
: This chapter describes the implementation of the different parts of the project and how they communicate with each other.

Evaluation
: The evaluation consist of a performance analysis, a functional test, a security analysis, and a description of future work.

Conclusion
: The conclusion summaries this report and evaluates the project.

Appendix B: Key Manager API
: Application programming interface for the Key Manager. Describes all functions available for developing new applications on top of the Key Manager.

Appendix C: Shell User Manual
: A short User Manual describing the different commands in the Shell.

Appendix D: Test
: Detailed test description and documentation for the test.

Attached CD-ROM
: The CD-ROM contains the source code for the implementation, this report as a PDF file, and all LaTeXfiles and images used in the report.

1.3 Notation

This section contains an overview of notations, abbreviations, terms and text types used throughout this thesis.

In this project the following notations has been chosen:

$E_K()$	Encryption with key K
$D_K()$	Decryption with key K
$S_K()$	Signing with key K
$V_K()$	Verification with key K
$hash()$	Hashing (message digest) function
K_{sym}	A symmetric key
K_{pri}	A private key
K_{pub}	A public key
M	A message text
C	A cipher text
DS	A digital signature
CAC-key	A cryptographic access control key

1.3.1 Abbrevations

In the report the following abbrevation will be used:

CAC Cryptographic Access Control.

NFS Network File System.

RPC Remote Procedure Call.

OS Operating System.

1.3.2 Terms and Text Types

Three different terms are used for the key rings in the project:

Private key ring The private key ring is located on a users local hard drive and is only accessible by the user. The private key ring is the root of the users tree of key rings and contains the keys to the next level of keys and files.

Shared key ring A shared key ring is an encrypted key ring which is located on a File Server. From the File Server's point of view, the key ring is treated as other files, but users with the correct key can open the key ring and use it keys.

Current key ring The current key ring is used as a notation for the key ring which the user currently browses, i.e. the current node in the tree of key rings.

Throughout this report *Itallic* writing will be used to introduce new terms, and the `Machine Writer` type will be used for programming terms or source code.

Chapter 2

Cryptographic Background

This chapter describes the fundamental cryptographic methods and algorithms used in this project. However, before the cryptographic algorithms are described, it is necessary to provide a general introduction to computer security.

2.1 What is a Secure System?

Before one can determine whether something is secure or not, it is necessary to define what is meant by being secure. In computer security, a typical approach is to require confidentiality, integrity, availability and, at times, authentication and non-repudiation. These five attributes are described as follows:

Confidentiality is to ensure that secret information is never disclosed to unauthorised entities.

Integrity is to ensure that data will not be corrupted.

Availability is a guarantee of accessibility of data on the network.

Authentication is the ability to verify the identity of an entity.

Non-repudiation means one cannot claim that a certain actions were never performed, e.g. the transmission of a message or the act of signing a message.

It is not always possible, or necessary, to require that all five attributes be fulfilled completely, but they should all be taken into consideration when designing a system that involves security [Pfl03][ZH99]. The five attributes are discussed with regards to this project in section 7.3.

2.2 Cryptography Basics

In this chapter it will be assumed that two entities, Alice and Bob, want to communicate without a third entity, Charlie, being able to understand their communication. They will therefore employ cryptography in their communications. To make the task more difficult, Alice and Bob must inform Charlie about how they will communicate, and how they will make sure that he is not able to understand their communication. The only thing kept secret from Charlie is one or several cryptographic keys. The communication takes place in the open, so Charlie will be able to retrieve everything sent between Alice and Bob.

When Alice wants to encrypt a message(M), she uses a key(K_1) and an encryption algorithm(E) to produce a cipher text(C):

$$C = E_{K_1}(M) \tag{2.1}$$

In symmetric cryptography, the same key is used for encryption and decryption[1]. Bob is able to decrypt the message by providing the same key to the decryption algorithm(D):

$$M = D_{K_1}(C) = D_{K_1}(E_{K_1}(M)) \tag{2.2}$$

The decryption algorithm is basically the same as the encryption algorithm, except everything happens in the reverse order. More details on symmetric encryption will be presented in section 2.3.

In asymmetric cryptography (public-private-key encryption) Alice and Bob use two keys, one for encryption and one for decryption. The encryption and decryption key cannot be derived from each other. In this case the same algorithm

[1] Some symmetric algorithms require a simple transformation from encryption key to decryption key.

is used for encryption and decryption[2]. The only difference is the key. If Alice uses an asymmetric encryption algorithm in equation 2.1, Bob will be able to decrypt the message with the corresponding key(K_2) using the same algorithm.

$$M = E_{K_2}(C) = E_{K_2}(E_{K_1}(M)) \tag{2.3}$$

The order of the encryption does not matter, i.e. Bob can encrypt with key K_2 and Alice decrypt with key K_1. In a real life scenario, Alice usually generates keys K_1 and K_2. She then publishes one of the keys as her public key and keeps the other as her private key[3]. With the public key, everyone is able to encrypt a message and send it to Alice. The only key that will result in correct decryption is Alice's private key, and she is therefore the only one able to decrypt the messages encrypted with her public key. If Alice wants to send a message to Bob, she will need Bob's public key. If she encrypts with her own private key, everyone will be able to decrypt the message, since her public key is commonly known. Her private key can, however, be used for digital signatures, because she is the only one who could have encrypted the message[4]. For more information about digital signatures, refer to section 2.6. More details on asymmetric encryption algorithm used in the project will be presented in section 2.4.

Although Alice and Bob are able to communicate in a secure manner using encryption, they cannot be sure that Charlie will not learn anything from their communication. First of all, Charlie will be able to detect when they communicate and how much they communicate. He might also be able to react on the communication itself. If Alice wants to inform Bob about when to buy a certain stock, and Charlie somehow knows of this intention, he can wait for her to start communicating with Bob and then buy the stock as soon as Alice sends a message to Bob.

Of course, Alice could be smart and send an encrypted "wait" message every minute or hour. However, Charlie would be able to spot, as soon as she sends a real message, that the message did not follow the same time interval as the other messages. Alice could avoid this by randomising the frequency of her "wait" messages. However, since Alice is sending the same message every time, i.e. "wait", Charlie can wait for the message to change to something else, and identify the real message.

Alice could further improve the scheme by sending random encrypted text mes-

[2]This is not always the case. Some asymmetric cryptography schemes uses different algorithms for encryption and decryption.

[3]Depending on the algorithm she uses, she might not be able to choose which key to publish.

[4]Usually a separate key pair is used for digital signatures, as encrypted text will be revealed when signing a message encrypted with your public key.

sages to Bob. When the time is right to buy the stock, Alice will send a message which she and Bob had previously agreed upon. Again, this would not be sufficiently secure, because Charlie could register the message sent to Bob just before he bought the stock, and then wait for the next time the same message was sent.

A better solution for Alice would be to include some additional text in all her messages. By doing this, all the encrypted messages will look different, and Alice will be able to send whatever text she wants, and can even repeat herself. Since the text is added to the end of the messages, she could choose her text randomly. On the other hand, she could also choose some useful information such as the current date and time. This would give Bob information about when the message was send and improve his knowledge about the message. This would also dissuade Charlie from resending a previously sent message and fool Bob to buy a stock which no longer is an profitable investment.

Including a timestamp in each message appears to be a good solution, but it requires Alice and Bob to be synchronised, which sometimes can be a problem if they live far apart or in different time zones.

2.3 Symmetric Cryptography - AES

In 1997, the National Institute of Standards and Technology(NIST) in the U.S. called for candidates to design a new encryption standard to replace the existing *Data Encryption Standard(DES)*, which was no longer considered secure([Sti02], page 95). In 2000, the Belgian-invented algorithm Rijndael was selected to be the new Advanced Encryption Standard(AES). Rijndael was superior to its competitors in overall security, cost, and algorithm and implementation characteristics[Sti02]. In 2001, it was adopted for use by the U.S. government and became Federal Processing Standard 197[oST01].

AES is a symmetric encryption algorithm, which uses key lengths from 128 to 256 bits[5]. Symmetric encryption is encryption on a shared secret, i.e. the two principals have previously agreed on a key.

When AES is applied on a long input text, the text is divided into blocks of the prescribed key size. Depending on the key size, AES algorithm has either 10, 12 or 14 rounds. The first step in the algorithm is to expand the initial key to a session key for each round (same size as the initial key). Then the four

[5]AES can easily be adopted to use longer keys than 256 bits, but the specification only describes 128 to 256 bits([Pfl03], page 72).

subroutines (SubBytes, ShiftRows, MixColumns, addRoundkey) are called on the input data for each round[6].

When an input block has been encrypted, AES uses one of four modes of operation to process the next block. The four modes are *Electronic Codebook (ECB)*, *Cipher Block Chaining (CBC)*, *Cipher Feedback (CFB)*, and *Output Feedback (OFB)*. The four modes describe different ways to combine blocks and keys during encryption and decryption. The most simple, ECB, uses the same key to encrypt all blocks, where the others include other information to make sure that two identical plain texts do not resolve in the same cipher text. This makes cryptanalysis even harder [Sti02]. In this project, the CBC mode is used. In CBC mode, the encrypted block from the last message is used together with the key for the next message. During decryption, the same method is used, i.e. after decrypting the first block, the result will be used with the key on the second block.

2.3.1 Security of AES

AES is considered secure, but this has, however, never been proven as a fact. Since AES became public, many cryptanalysts have tried to break AES in different ways. In general, the attacks can be divided into three categories: Attacks on the algorithm, monitoring of the CPU cache (Side Channel Attacks), and brute force attacks. This section will briefly describe the three kinds of attack and discuss their threat to the security of this project.

2.3.1.1 Attacks on the algorithm

At the time of writing, no successful attacks have been reported on the AES algorithm[KR]. A few attacks on simpler versions of AES with fewer rounds have, however, been found. The best result on 128 bit AES was achieved by breaking seven of the twelve rounds with a *Chosen Plaintext Attack* [ea00]. Although this attack weakens the algorithm, it does not pose a treat to AES in this project. The purpose of the Chosen Plaintext Attack is to find an unknown key by encrypting several different plaintexts. To perform this attack, it is necessary for the attacker to have a chosen text encrypted and to see the result. This could be from a computer that always replied with an encryption of whatever message was sent to it. By using selected plaintexts, the attacker can then uncover the key. This could be a threat to this project if a full chosen plaintext attack was possible. Services running on a computer, such as a log system, will

[6]MixColumns is omitted in the last round.

also be using the cryptographic access control and their key could be revealed, because the attacker would be able to fool them to encrypt whatever text he chose.

2.3.1.2 Side Channel Attacks

As of 2005, the only successful attacks on AES have been side channel attacks[Wik06]. To carry out a side channel attack, one needs access to install foreign software on the computer performing the encryption or decryption. The software is designed to monitor the CPU's memory cache. By doing this, it is possible to obtain the secret key in just 65 milliseconds[DAOT05]. The key can be obtained, because it is possible to see which registers the CPU is using for which operations and thereby learn which registers that contains the key.

Since this attack is based on compromising the computer rather than the algorithm itself, it is not to be considered an actual treat to AES. Again, this might be a threat to the system, because the attacker could monitor the cache when a certain service was encrypting a text. The operation system should, therefore, strictly protect the CPU's memory cache (see more in section 7.3) if services are running on a client computer.

2.3.1.3 Brute Force Attacks

All cryptographic algorithms can be broken with brute force attacks (also known as exhaustive key search). It is only a question of time and CPU power. Since there only exists a finite number of keys, an attacker will always be able to try all possible keys. AES requires a minimum key length of 128 bits, which is equivalent to $3,4 * 10^{38}$ different combinations. Statistically, it is only necessary to try half of the keys before finding the correct key. If the attacker uses a 3 GHz CPU, which is able to try one key per clock cycle[7], it will still take $1,7 * 10^{21}$ years find the right key. Even if all 6.411.000.000 inhabitants on Earth had a similar computer and connected them to each other, the attack would still require 21 times the age of the Universe.

A brute force attack is therefore considered to be unlikely on AES as long as CPU processors follow Moore's law. If, however, scientists are able to design faster supercomputers in the future, the possibility of breaking AES will remain.

[7] In a real scenario it will not be possible to test one key per clock cycle since the algorithm requires some computations.

Furthermore, other weaknesses might be discovered in AES and it is therefore not recommended to use AES (128 bit version) for information which must be guaranteed to remain secret for 30 to 40 years[Pfl03].

2.4 Asymmetric Cryptography - RSA

RSA is the most commonly used asymmetric encryption algorithm. It was invented in 1977 by Ron Rivest, Adi Shamir and Len Adleman from MIT[Pfl03]. RSA is based on the problem of factoring large Integers (see section 2.4.2). Beside its use for public-private (asymmetric) cryptography, it is commonly used in digital signature schemes (see section 2.6).

2.4.1 The RSA algorithm

The RSA algorithm and its security aspects will not be described in detail in this report. For more information about RSA, please see [Sti02]. Briefly, the algorithm consists of the following steps[Sti02]:

1) Choose two large primes (p, q) and compute $n = pq$.
2) Compute $\phi(n) = (p-1)(q-1)$.
3) Find $ab \equiv 1 \bmod \phi(n)$.
4) The public key is now n and b. The private key is: p, q and a.
5) Encryption can be performed by computing: $e_k(x) = x^b \bmod n$.
6) Decryption can be performed by computing: $d_k(y) = y^a \bmod n$.[8]

The algorithm is based on the mathematical group theory for prime numbers, which states that $x = (x^b)^a \bmod n$. This holds according to Fermat's little theorem, as long as a and b are selected according to step 3 above. In real life, $(x^b)^a$ will be an extremely large number and it is therefore necessary to break it into smaller pieces. This is done be applying *mod* n to all calculations, which gives:

$$\begin{aligned}(x^b)^a \bmod n = \quad & (x_1 * x_2 \bmod n * x_3 * \bmod n \ldots * x_b \bmod n)_1 \\ & * (x_1 * x_2 \bmod n * x_3 * \bmod n \ldots * x_b \bmod n)_2 \bmod n \\ & \vdots \\ & * (x_1 * x_2 \bmod n * x_3 * \bmod n \ldots * x_b \bmod n)_a \bmod n\end{aligned}$$

[8]The Chinese Remainder Theorem can be used for fast decryption. It requires that more values are stored in the private key, but allows faster decryption.

As stated in step 4 the public key consist of n and b. Step 1 describes the relation between n, p and q. If it was possible to reverse this step and calculate p and q from n, RSA could be broken. An attacker would then be able to find p and q from n and then compute a from the equation in step 3. The attacker would then have a, p and q which, together, makes the private key.

2.4.2 Attacks on RSA

Factoring large numbers is not as easy as one might think. RSA Security has, for the last decade, administered a public challenge on factoring large number[RS]. At the time of writing, the 663 bit (200 decimal digits) challenge has been factored. To factor the challenge, 80 2,2 MHz Opteron CPU's used equivalent to 55 years of computing time for a single CPU[Sla]. To complete the task, the number field sieve algorithm was used, which, according to [Sti02], has the running time:

$$\Theta(e^{1.92+o(1)(\ln n)^{1/3}(\ln \ln n)^{2/3}}), \tag{2.4}$$

where $o(1)$ denotes a function of n that approaches 0 as $n \to \infty$.

Figure 2.1 shows the relation between computing complexity and the size of n: doubling the key size does not lead to doubling the exhaustive search time.

Applying figure 2.1 to the result from the factorisation of the 663 bit, it would take approximately $2,6 * 10^{110}$ years to factor RSA 1024. [ST03] however suggests that custom-build hardware could factor a 1024 bit number in one year. The approximate price was $10M in 2003. The price would be somewhat lower today and therefore 1024 bit keys are not recommended for high security purposes.

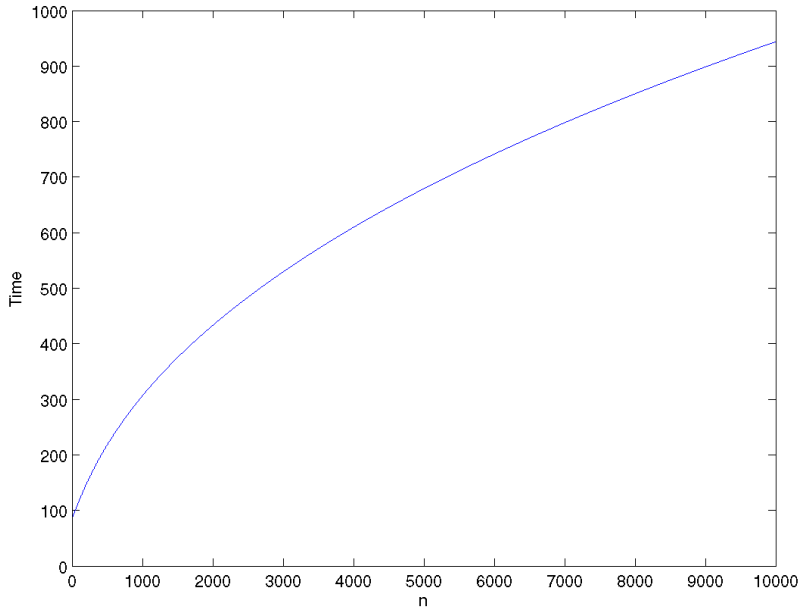

Figure 2.1: Time required to find n using the field sieve algorithm. The time is not an absolute value; it depends on implementation, computer power, etc.

2.5 Hash Functions - SHA-1

Besides symmetric cryptography(AES) and asymmetric cryptography(RSA) hash functions play an importent role in cryptography. The purpose of a hash function is to create a short digest of an arbitrary message. The digest must not have any similarities with the original message, and have a fixed output length (typically 128 to 256 bits). In hash functions, it is easy to compute the output value from the input value, but very difficult "to go back", i.e. compute an input value from the corresponding output value. Because of this feature, hash functions are usually called *(cryptographic) one-way functions*.

In general, hash functions should have these properties[HK03]:

Preimage resistant it should be very computationally intensive to find the input from a given output.

2nd preimage resistant when given one input, it should be very computationally intensive to find another input with the same output value.

Collision resistant it should be very computationally intensive to find two inputs with the same output value.

The collision attack is the usually the easiest way to break hash functions, and it is therefore the attack that cryptanalysts try first. Collisions will always exist, because hash functions produce an output with fixed length from an arbitrary input. The number of possible inputs are therefore infinite, whereas the number of possible outputs are finite.

Hash functions have many purposes in modern cryptography. The two most common uses are in digital signatures (see section 2.6) and for storing passwords. When a password is stored in a database, it can be an advantage to only store the hash value of the password. By doing so, only hash values will be kept in the database and system administrators, etc. will not be able to retrieve the passwords. This improves security because the only person knowing the password is the owner. When the user wants to login, he enters his password, which then is hashed and compared with the already hashed password stored in the database. If the two are equal, the user is authenticated. Although this is believed to be the best method to handle passwords, it does have one disadvantage: the password can not be restored. For the sake of user-friendliness, when users forget their passwords, it is often more convenient to remind them of their passwords instead of creating new ones. This is not possible with the suggested hash functions scheme.

2.5.1 SHA

The Secure Hash Algorithm (SHA) was published in 1993 by NSA. Shortly thereafter, a reviewed specification was published which was also known as SHA-1. It has become very popular and is probably the most used hash algorithm today. When SHA-1 was published, no comments were given on the changes. However, later studies have shown that the original SHA algorithm is more vulnerable to differential cryptanalytic attacks.

SHA-1 has a fixed output size of 160 bits. A collision would therefore require 2^{80} hashes on average. This is considered to be a computational task too heavy for computers today. Xiaoyun Wang, Yiqun Lisa Yin and Hongbo Yu have, however, found weaknesses in SHA-1 and have shown that collisions can be found in 2^{69}. This has been further improved and the theoretical bound is now believed to be 2^{63} [XWY05][Sch].

2^{63} is within reach for a supercomputer's computational power. No collisions have been report in SHA-1 yet, but they are believed to be found soon. Even if a collision in SHA-1 is found, it is not the end of the algorithm. Collisions are, in normal cryptography, only a threat to digital signatures, and does not affect the previously mentioned password scheme.

New versions of the Secure Hash Algorithm have been suggested. These range from 256 bits output to 512 bits output and are called SHA-2 or just SHA-256, SHA-512, etc. These require additional computations and more storage space and are therefore not used in this project. SHA-1 has sufficient security for this project, because the hash algorithm is not used as a digital signature. More about this in section 7.3.

2.6 Digital Signatures

A digital signature is, as previously mentioned, not an encryption. When a message is digitally signed, the signer's private key is used in an asymmetric encryption algorithm, typically RSA. By encrypting with the private key, everyone is able to decrypt, i.e. no confidentiality. However, since the private key is only known by the key owner (signer), only he/she could have signed the message. This provides authentication similar to a written signature, which can only be produced by its owner[9].

[9] Written signatures are not hard to fake, and are therefore more used to state an acceptance of a written statement.

In practice, RSA and other asymmetric algorithms are slow to use and require a great deal of computation. The digital signature (DS) is therefore not computed on the entire message, but on a hash of the message:

$$DS = S_{privatekey}(M) = E_{privatekey}(hash(M)) \tag{2.5}$$

The result is then typically pasted at the bottom of the message in the same way as a written signature. By using this scheme, it is clearly shown to the user, that the message is not encrypted (since it still is in clear text). If encryptions is necessary, the sender must have the receiver's public key. More information on how to get the key will be given in section 4.

To verify a signature, one needs the signer's public key:

$$hash(M) \stackrel{?}{=} V_{publickey}(DS) = E_{publickey}(DS) \tag{2.6}$$

By replacing DS with the result from 2.5, one can see that 2.6 will be true if the public and private keys corresponds to each other.

2.6.1 Collisions in Digital Signatures

If a hash function has collisions, i.e.:

$$hash(M_1) = hash(M_2), \tag{2.7}$$

an attacker could send a message (M_1) with a contract to a victim. If the conditions was beneficial for the victim, he would sign the contract with his digital signature and return it to the attacker. The attacker could now replace clear text in the contract with another message (M_2), which had the same hash value and claim that this was the contract the victim signed. Because the hash value of M_1 and M_2 are identical the signature would appear to be original, using 2.5 and 2.6 on 2.7 we get:

$$hash(M_2) = E_{publickey}(E_{privatekey}(hash(M_1))), \tag{2.8}$$

i.e. the signature of M_1 is valid for M_2.

2.7 Key Management

Having secure communication is not enough. This only provides confidentiality. Sometimes it is necessary to know who one is communicating with, i.e.

to authenticate each other. This can be a difficult task with today's global communication and infrastructure, as it is not possible to meet all recipients of communication and verify their identity. Therefore, other means of verifying authenticity must be used. In this section, three typical methods will be discussed.

2.7.1 Public Key Infrastructure

Public Key Infrastructure (PKI) is one way to handle public keys and identities. It is, among other things, used to distribute keys in the Danish digital signature scheme.

The basis of a PKI is a trusted *Certificate Authority (CA)*, also known as the root CA. In some PKIs, several subroot CAs are trusted to sign certificates when their certificates have been signed by the root CA. This could be a government or private company that is trustworthy for users, e.g. in Denmark, the government has initiated a national digital signature and outsourced the distribution to TDC, therefore TDC is acting as a Certificate Authority.

The purpose of the CA is to verify the principal's identity and issue *certificates*. Its public key must be known by everyone and the private key is used for signing certificates. A certificate is bound to a user and contains information about the user, such as name, address, social security number, etc., and the user's public key. The certificate is signed by the CA.

If Alice wants to communicate with Bob, she will send him a message, signature and certificate, issued by the CA, i.e. Alice sends:

$$Alice \rightarrow Bob : M + S_{Alice}(M) + S_{CA}(K_{Public_{Alice}}) \tag{2.9}$$

Bob will then be able to verify her signature with the certificate and the certificate with the already known public key from the CA. Of course, this only works if Bob trusts the CA and already has its public key.

Statement 2.9 does not provide confidentiality. If Alice wants to send Bob an encrypted message, she will need his public key. In some senarios she is able to obtain Bob's public key from a public key server, which has all user's public keys.

If Alice and Bob wanted to communicate for a longer period of time, they would usually agree upon a symmetric session key. Using a symmetric session key would require less computation during encryption and decryption, and be

somewhat more secure because they are able to replace the session key when appropriate, and minimize the use of the same key.

The PKI proposes a solution to the key distribution problem. In a real life scenario it is unfortunately not very applicable. The main reason for this is its centralised certificate authority. It is very hard to find one central authority that everyone in the world trusts. People from different countries will always have a hard time believing a foreign government or company.

Microsoft have solved this problem by pre-installing several different root certificates in *Internet Explorer(IE)*. By doing so, several different companies have the possibility to act as root CA's. This might seem like a good idea, but have proven not to be. Many of the root or subroot CA's have not fulfilled their requirement and issued certificates without making a proper authentication of the applicant's identity. As a result of this, many untrustworthy certificates have now been issued. Since the root certificates are trusted as default in IE, the verification of the certificates is useless and should not be trusted[10]. It is therefore recommended that untrusted root certificates be removed from IE.

2.7.2 PGP: Web-of-Trust

In 1991, Phil Zimmerman implemented the first version of *Pretty Good Privacy (PGP)*. It uses a so-called *Web of Trust* to determine authenticity between a certificate and a user. Instead of the centralised CA, PGP uses self-signed certificates, where users sign each others certificates. A certificate will therefore contain more than one signature. Furthermore, a log of trusted certificates is kept.

When a signature or authenticity need to be verified, the certificate is either included in the mail or retrieved from a public certificate database. To verify the integrity of the signature, the other signatures on the certificate are checked. If any of the signatures matches a certificate in the log of trusted certificates, the new certificate is trusted and added to the log. If not, one of the certificate databases is contacted to see if it can find *a trusted path* from the received certificate to the users certificate. A trusted path is a path of certificates that have signed each other. The path only needs to be one-way, i.e. it must be possible to establish a path from the certificate to the receiver, but not necessarily the other way around. Each user can specify whether he fully or partially trusts a certificate and how many partially trusted paths he needs to a certificate before it can be trusted.

[10] Read more on http://www.microsoft.com/technet/prodtechnol/ie/reskit/6/part2/c06ie6rk.mspx (23/1-06).

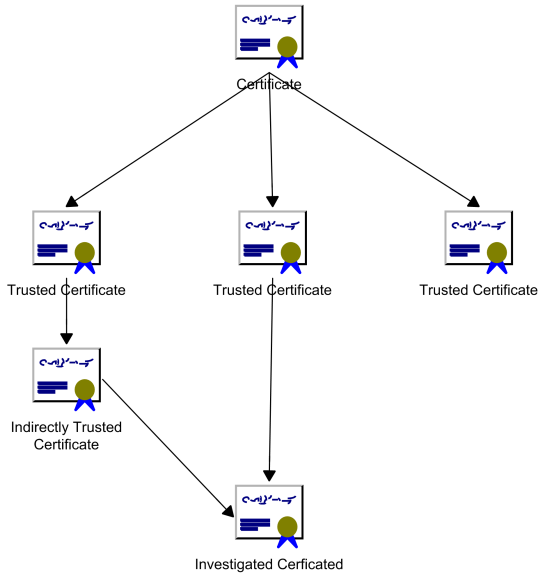

Figure 2.2: The Web of Trust.

Figure 2.2 shows an example of two trusted paths from one certificate to another. The top certificate belongs to the user and the bottom one is the certificate under investigation. Since at least one path exist from the top to bottom the certificate can be trusted.

2.7.3 Confidence Values in Key Management

Ueli Maurer[Mau96] has suggested a further improvement of the PGP and PKI where certificate authorities and entities are trusted with confidence values. Certificates can be retrieved from the PKI based on these confidence values and recommendations given between entities in the system. By combining several certificate paths, the confidence value to specific certificates can be increased. The final judgement is based on probabilistic logic on the combined confidence values. This is basically just a more fine-grained version of the PGP, where decimal numbers are used to express the level of trust, with the addition of recommendations.

Recommendations are especially interesting because the increase the possible paths to a given certificate, i.e. they can be used to provide better judgement of

a certificate. Recommendations could also be applied in PGP or used between different PGPs. This would give the users even better judgement of certificates.

2.8 Summary

This chapter offers an basic introduction to computer security and cryptography. The first part described what security means in current computer systems and what kind of cryptographic methods are used to provide this security. The next part focused on these cryptographic methods and described the algorithms used in this project. The algorithms (AES, RSA and SHA) were explained briefly, and their security toward different attacks was analysed. The last part of the chapter focused on key management and how to handle and distribute the cryptographic keys. This is very important in modern communication, because cryptographic algorithms are not able to handle authentication and non-redudiation.

Altogether, chapter has given the reader an introduction to cryptography necessary to understand the following chapters of this project. The algorithms will, from now on, be used without further explanation. The security analysis of the algorithms will be used in the complete security analysis of the project.

Chapter 3

Cryptographic Access Control

This chapter discusses traditional and cryptographic access control. First, the traditional access control model is described, and some short comings when applied to distributed file systems are discussed. Then, the basics of cryptographic access control are described and a brief summary of its history is given.

3.1 Access Control

Today's operating systems and file servers rely on a reference monitor to control access to devices, file and services[Pfl03]. When a user requests access to a resource, his credentials are checked. Depending on his access rights, the reference monitor grants or rejects access to the requested resource. The reference monitor can use different methods to maintain access rights for the users.

The simplest way to maintain user access rights is with an *Access Matrix* (see table 3.1). In an access matrix, all subjects (users, system users, etc.) are listed together with all objects (files, printers, etc.). Each cell describes the authorisation the users has for the specific object. In table 3.1 `User1` only has authorisation to read `File2`. The `Super User` has authorisation to read and write both files, and to print and configure the printer.

Subject \ Object	File1	File2	Printer1
User1		Read	
User2	Read-Write		Print
Super User	Read-Write-Execute	Read-Write	Print-Setup

Table 3.1: An Access Matrix used to maintain users access rights.

The access control matrix is intuitive and easy to understand and use. It is, however, not suitable for real operating systems, because it uses too much unnecessary memory for all the cells where no authorisation is given. It is therefore more common to use *Access Control Lists (ACL)* or *Capability Lists*.

Access control lists are derived from the access matrix, and are basically the columns of the access matrix. Each object has its own access control list, which describes which subjects are allowed to access the object. `File2` would, for example, have the access control list:

$$File2_{access} = \{User1(R), SuperUser(RW)\} \tag{3.1}$$

Access control lists are the most commonly used method to maintain access rights and are used by reference monitors in Linux and other Unix-like operating systems[Tan99]. Normally, the users are divided into into three groups. For each resource, specific access rights are specified for the owner, other members of his group(s) and all other users. By doing so, it is only necessary to maintain three subjects (user, group, all) for each file.

Capability lists are closely related to the access matrix as well, but instead of describing the authorised subjects to an object, they describe the capabilities for each object, i.e. they are represented by the rows in the access matrix. `User2` would for example have the following list:

$$User2_{capability} = \{File2(RW), Printer1(P)\} \tag{3.2}$$

By describing access rights with capability lists, all objects that `User2` does not have access to are omitted.

3.2 Distributed Access Control

The reference monitor model is only suitable for single operating systems and minor distributed systems on trusted networks. Because access rights are specified per user, it is necessary for the reference monitors to have the ability to verify the users' identity, i.e. some sort of user administration. Since new users

are created and their groups are constantly maintained, the user administration must be centralised, or several user administrations must constantly replicate their data. This model is not very scalable. Furthermore, it requires the user administrations to be trustworthy. If one user administration is compromised, it will be able to compromise the entire system.

Another problem with existing access control mechanisms is that the users must trust the object owners and administrators. Since administrators traditionally have *root* access, they will always be able to access and edit the users' data. This can sometimes be a problem with private data that the users do not want the administrators to see[1].

In distributed systems over insecure networks, the access control model faces another problem: confidentiality. Because the communication is unencrypted, eavesdroppers are able to read all information sent between the parties. This is undesirable in many cases and often, encryption is used to guarantee confidentiality.

Encryption can, however, be a resource intensive task. Before the secret communication can be established, it is necessary to find a session key. Figure 3.1(left side) illustrates how a key exchange of a session key could take place([Pfl03], page 78.). There are many different ways to make key exchanges depending on the purpose of the key exchange. In this example, two-way authentication is guaranteed if the client's and server's certificates are signed by trusted CAs. The client and server first authenticate each other and establish a session key. Then, the file is encrypted and transmitted. The session key is used because asymmetric encryption is much slower than symmetric encryption. If the amount of data that the server is going to send to the client is small, it is not necessary to create the session key. The encryption will, then, only be asymmetric. If the client wants to request another file, the same session key can be used and it will not be necessary to resend the certificates.

[1] This problem can be partially solved with extensive logging.

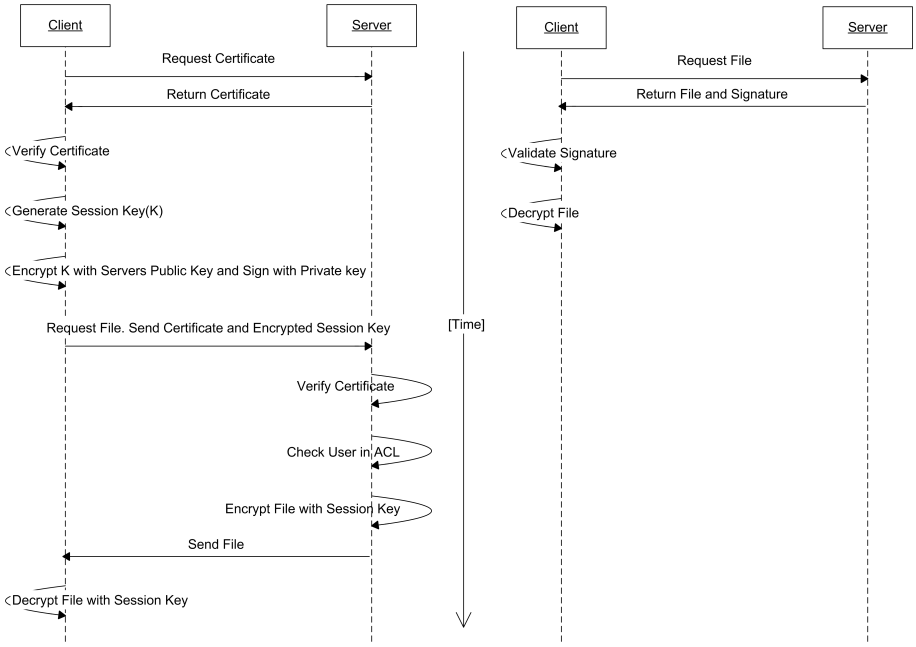

Figure 3.1: Left: A normal file request over an insecure network. Right: A file request using CAC.

3.3 The Cryptographic Access Control Model

The base of Cryptographic Access Control is as the name implies: cryptography. It is a mechanism to control access to objects using cryptography instead of a reference monitor. All files are encrypted before they are transferred to a server. The conceptual proposals of CAC only uses asymmetric encryption[Jen00]. Private keys are used to encrypt files and public keys to read them. Once a file is encrypted it can be placed on a server and read by everyone with the public key (Management of access rights requires a key distribution mechanism, but key distribution is not a part of the model.). Since the files are encrypted, they can be placed on untrusted servers and the information can still be kept confidential.

Later models of CAC suggests that symmetric encryption is used for the file encryption because of better performance. Asymmetric encryption is then used to sign the files to provide integrity. To read a file, the reader must, as a minimum, possess the symmetric key. If the reader also possesses the public key, he will be able to check the integrity of the file, i.e. if it was created by someone with the correct private key. This model also has another advantage.

Because the public key cannot be used to decrypt the file, it can be used by other instances to check the integrity of the file. This could be file servers that the file is uploaded to. It could then reject files that do not have the correct integrity[Har01][HJ03], i.e. ensure that only users with the correct private key can upload a file. In cryptographic access control the users maintain the access control individually. They are able to grant other users access to resources and revoke access again if necessary. The authorisation is controlled only by the keys. If a user wishes to grant another person access to a resource, he will give him the corresponding key. He can choose to give the user full authorisation to the file (symmetric key, private key and public key) or just read authorisation to the file (symmetric key and public key). The model is completely distributed and is therefore highly scalable. There are no central authorities and several models can be nested within each another.

Figure 3.1 (right side) demonstrates the necessary steps for a file request using cryptographic access control. It is assumed that the underlying network is insecure. Cryptographic access control clearly has advantages compared to the traditional reference model. Because the client already has the key to the file, the server does not need the encrypt the file before transmission. Neither does the server need to worry about authentication of the client, because only clients with the correct keys can decrypt the file. Because of the minimal use of asymmetric encryption, the model also has advantages for clients with limited computation resources such as mobile phones and PDAs.

3.3.1 Formalisation of the CAC Model

When a file is created, no keys are needed, since this involves creating the keys. To update a file, one needs:

K_{sym}	is used to encrypt the file.
K_{pri}	is used to sign the file.
K_{pub}	is sent to the server and used for verification.

To read a file, one needs:

K_{sym}	allows the user to decrypt the file.
K_{pub}	is used to check the integrity of the file. If the signature is correct, only a user with the correct private key could have encrypted and signed the file.

3.3.1.1 Create a file: Step-by-Step

To create a file, the a user perform the following steps:
1. Generate the symmetric key.
2. Generate the public-private key pair.
3. Encrypt the file with the symmetric key: $C = E_{K_{sym}}(M)$
4. Sign the encrypted file: $DS = S_{privatekey}(C)$
5. Send $\{C, DS, K_{pub}\}$ to the server.

If the equation 3.3 below is true, the server will save the file. The verification is made to ensure integrity of the file.

$$hash(C) \stackrel{?}{=} V_{publickey}(DS), \tag{3.3}$$

3.3.1.2 Read a file

The client will request $\{C, DS\}$ from the server. The integrity of the file is intact if equation 3.3 above is true. If the integrity is correct, the client will decrypt the file using:

$$M = D_{K_{sym}}(C) \tag{3.4}$$

3.3.1.3 Update a file

Before a file is updated, it will normally be read from the server[2]. The updating process is almost the same as the process of creating a file. The only difference is that the client will not need to create keys and it is not necessary to send the public key to the server. In other words, the client:

1. Encrypts the file with the symmetric key: $C = E_{K_{sym}}(M)$
2. Signs the file with the private key: $DS = S_{private}(C)$
3. Sends $\{C, DS\}$ to the server.

Again the server has to check equation 3.3. This time it uses the stored public key and overwrites the encrypted file (C) and the signature (DS), if the integrity is correct.

[2]CAC is only a mechanism for access control and does therefore not handle locking of files on the server.

As previously mentioned, none of the steps discloses the symmetric key to the server. Therefore, less trustworthy servers can be used to store data. Using a less trustworthy server might lead to availability problems, but confidential data is never disclosed. Besides availability problems, it might be possible to fool the server to overwrite an updated file with an older version using a *Replay Attack*. This can be successfully completed, without knowing any keys, if the server does not implement a log-system or version control.

The model minimizes server side computations, but could be vulnerable to *Denial of Service Attacks*, because everyone can request a file without authentication, i.e. many people can request a file from the server and overload the network. This would make files unavailable for users with correct keys (See the Security Analysis section (7.3) for more information).

3.4 Other applications using CAC

Cryptographic Access Control have already been implemented in several applications. This section presents a brief review of previous work with CAC.

3.4.1 CryptoCache

Christian D. Jensen was the first known to propose cryptographic access control in 2000[Jen00]. The paper *CryptoCache: A Secure Sharable File Cache for Roaming Users* describes how asymmetric encryption can be used to store information on untrusted nodes in networks. This article is the constitution of cryptographic access control and describes the basic principles of cryptographic access control, i.e. how the public-private key pair can be used as a read-write key pair when information is stored on remote servers. The main focus of the paper is how mobile devices with limited storage resources can use file servers for temporary storage. The model is completely distributed and does not require a global authentication framework. The proposal is, in general, focused on applications over mobile networks, but it also suggests CAC use in other applications.

3.4.2 Fast Cryptographic Secure NFS

Shanahan tried to build cryptographic access control into the *Network File System (NFS)* by implementing it on the operating system kernel level[Sha]. The project focuses on the details of designing a new file system with the features of CAC. The project does not consider key management on a file level, but focuses on the possibility of mounting drives on servers using CAC, i.e. an entire block is mounted on a server. It was not possible to implement a stable version of the access control mechanism on the operating system kernel level because of the sensitiveness of the kernel. Cryptography can be a computation intensive task and because of the non preemptiveness of the kernel, the entire operating system froze when a cryptographic task has to be completed. As a result, the project had two hard-disk crashes due to pointer errors. The kernel was originally chosen because of optimal performance, but it was too insufficient API made the task to complex.

3.4.3 Network File System

Harrington implemented CAC in a network file system (NFS)[Har01] on the application layer, instead of the kernel level. Because of the operating systems protection of the memory it was possible to create a stable prototype of a file system using CAC and proof the concept. The main focus is on cryptography and performance. The implementation demonstrates that CAC can be used in real life applications and has some advantages to the traditional reference monitor model. This is especially true for untrusted servers or servers with low computational power.

3.4.4 Remote Procedure Call

Henrik Christensen and Jonas Høgh has developed and discussed an expansion of Remote Procedure Calls (RPC) to use CAC. Their implementation allows users to create RPCs similar to the method defined by Sun, but with encryption using CAC. This guarantees users' confidentiality and authentication, because only authorised clients can communicate with a server. Their project allows programmers to easily create secure communication between servers and clients without worrying about cryptography.

3.4.5 Peer-to-Peer Network

Søren Hjarlvig and Jesper Kampfeldt have suggested and implemented a CAC implementation of a peer-to-peer network in $Java^{TM}$ JXTA[HK03]. Their model uses key rings, where each key either opens a new key or is a key to a file. The files are shared on the entire network and available for everyone, but because of the underlying CAC model, only users with the correct keys are able to read and update them. The key rings are also shared between users similar to the files. Only users with the correct key to a key ring is able to open it and retrieve its keys. The project demonstrates how this can be used as an access control scheme similar to role based access control, where a key ring can represent a specific work function or task that involves several files. The key rings can, however, also be used in other application schemes. In general, the project shows how the cryptographic access control mechanism can be used together with a key management system, such as key rings, to provide a complete security model that offers confidentiality, integrity and some authentication. The model does not offer complete authentication of users, but guarantees that only authorised users are able to update files.

3.5 Summary

This section introduced, and described in detail, cryptographic access control. Its advantages in distributed files systems was discussed and a brief overview of its history so far was provided.

Cryptographic access control is, in short, a mechanism to use cryptographic keys to control access to resources, instead of the traditional reference monitor model. It can be completely decentralised and is, therefore, very applicable to distributed systems. The combination of symmetric and asymmetric encryption allows three access levels to resources: full, read and integrity check. The integrity check can be used by file servers to check the integrity of a file without learning the actual contents. Thus, servers are able to reject file updates without knowing the actual contents based on integrity checks alone.

Chapter 4

Key Management

In chapter 3 the idea of cryptographic access control was introduced, but the chapter did not describe how the keys should be handled. Since every file has its own key this chapter introduces a new problem: handling of the keys. In this chapter key management will be discussed. The main focus will be on handling keys for cryptographic access control using new and existing techniques.

The chapter will introduce and explain *CAC-keys*, *key rings* and *link keys*, and it will be demonstrated how these techniques can be used to provide different access control models, with the same key ring implementation.

Finally the key management term will be expanded to a more general approach that not only include files, but also different kinds of service, and it will be discussed how the solution will able handle all different kinds of access issues.

4.1 Cryptographic Access Control Keys

In a cryptographic access control model, resources are distributed on remote servers. The cryptographic access control mechanism is used to restrict access to resources. To gain access a principal must possesa the correct key, containing

the necessary information. The principals must therefore have a set of keys that provides access to all the resources. These keys should besides the cryptographic keys contain information about the resource, such as resource type, resource owner, name of reaource, location and access rights to the resource. To contain this information *Cryptographic Access Control Keys(CAC-keys)* are introduced. Their exact content can vary from system to system, but should as a minimum contain:

- Name of Resource.
- Type of Resource.
- Owner of the Resource, i.e. contact information.
- Access Rights to Resource.
- Location of Resource.
- Symmetric Key.
- Public and/or Private Key.

A CAC-key is therefore not only a cryptographic key, but a more general notation of the set of information necessary to access a resource. CAC-keys in cryptographic access control have strong similarities to capabilities in traditional access control, because they have a represent a certain access right for a resource for the principal.

4.2 Keys and Files

In this section different ways to handle the relation between files and keys will be discussed.

4.2.1 One key per Directory

The first papers of CAC suggested that a directory of data, containing several files, should have one CAC-key[Sha][Har01]. When a principal requests access to a file, the entire directory will be decrypted on a local drive. The principal is then able to access the files as he pleases. When the principal is finished

working with the files, he can choose to encrypt and upload the files to server[1] or just delete them.

This proposal offers a solution where the principals only have a limited number of keys and the key administration will therefore by rather straight forward. Performance would however not be optimal. Every time a file is requested the entire directory would have to be decrypted and when a file id updated the entire directory should be re-encrypted and transmitted to the server. The reason for this, is that symmetric encryption algorithms usually are implemented with a specific *mode of operation*(see more in section 2.3), that uses the output from the previous encryption in the next encryption. This makes the encryption blocks depend on each other. A change in one plain text block will therefore lead to a change in all consecutive cipher text blocks[2].

4.2.2 A Key per File

As discussed in the previous section a key for several files, in a directory, was not always the optimal way to handle keys and files, because too much unnecessary encryption and decryption would have to take place. Furthermore a key would be closely related to the set of files in the directory. It would for example not be possible to have one key to all the files and a another less restricted key to some of the files.

Another solution would be to give each user a key to each file they need access to. The users could then place all their individual keys on their local drive, i.e. in the "home" or "My Documents" folder. The keys could be in one file or each key could have its own file. If the latter solution is chosen, the files could be placed in sub-folders to give some structure of the keys. Each key would then represent a shortcut or link to a file on the distributed servers. A file could then be opened using a small software tool, that contacted a file server and decrypted the file using the selected key.

This solution would be very easy to implement, but has some problems. First of all, each user will need a key to each file he needs access to. If a key needs to be changed all users have to be notified and receive a new key. The same situation occurs if a file is relocated to another server or deleted. Secondly, the solution is not very structured and causes many key distribution problems. On

[1] Requires "write" authorisation to the files.

[2] In ECB mode each block is encrypted individually with the encryption key. Two identical plain text blocks will therefore result in two identical cipher text blocks. The ECB mode could therefore be used to only encrypt the modified blocks, it is however recommended not to use ECB, because of the weakened security[Sti02].

todays file servers access is usually granted to folders rather than files. This is because users typically need to be able to create files that other users can use. If a key is needed for each file, a user needs to distribute a new key every time a new file is created. This is quite a difficult task and not very user friendly for the users.

4.3 Key rings

To avoid these problems *key rings* have been suggested to handle the key management[HK03]. Basically a key ring is a set of CAC-keys, but it has two new advantages. First of all, key ring can be placed on a file server. Secondly, a key in a key ring can be used to open another key ring. This allows users to create a hierarchy of keys rings which can be used to implement an access control model.

Each user has a private key ring (in the local folder), with keys to files and key rings. Both files and key rings can either be local or placed on a server. The key rings are of course encrypted the same way as the files and again the cryptographic scheme is used to administrate the key rings. This provides the possibility to specify who is allowed change the key rings, i.e. add/remove keys, and who only is allowed to use the keys in the key ring.

Because the key rings are encrypted the file servers will not be able to distinguish key rings from ordinary files. They will therefore perform the same integrity check as on regular files, and thereby ensure that only authorised users are allowed to change the key rings.

4.3.1 Key Rings as Access Control

Because a key ring contains keys, that provide access to several files, the key rings can be used to implement several access control models. In this section two examples will show how key rings can be used to model access control. The two chosen access control models are user group based access control and role based access control. The two models are only given as examples and other access control schemes could also be modeled.

4.3.1.1 Access Groups like Linux

In the Unix and Linux environments, users are as earlier mentioned divided into groups. File owners are then able to specify specific rights (read, write, execute) to each files. The same model can somewhat be modeled using key ring. In figure 4.1 Bob has created a set of files. Everyone (Others) are allowed to read the files. Users from the same group as Bob are allow to read and write to files. Because Bob has full authorisation to the two key rings he is able to access the files and maintain the access rights for the files by adding or removing files from the two key rings.

The model has two minor short comings compared to Unix model. First of all can execute permissions not be handled, but the users cannot execute the files on the servers, because they need to be decrypted first. Secondly it is not possible to grant only write (no read) authorisation. It would however be quite uncommon to allow user to edit files without giving them the right to displaying them.

4.3.1.2 Role Based Access Control

With key rings is it possible to model access control similar to role based access control[HK03][Pfl03]. A key ring can then be used to represent a certain job function or a certain set of access rights.

In figure 4.2 an example of how key rings can be used to model role based access control is given. Alice and Bob both work with purchasing, but are located in different locations. They are therefore sharing a purchasing key ring, but both have access to different location based key rings. The location based key rings grant access to local printers and local files which are only available to people on the same location.

Depending on the setup, they can be allowed to maintain the purchasing key ring, so they can create new files in it.

4.3.1.3 Modeling Other Access Controls

The advantages of using key rings to create an access control model, is that once the key rings have been implemented it is possible to create different access control models only by changing the access rights in the key rings, i.e. it is not necessary to make any implementation changes. In fact, it is possible to have

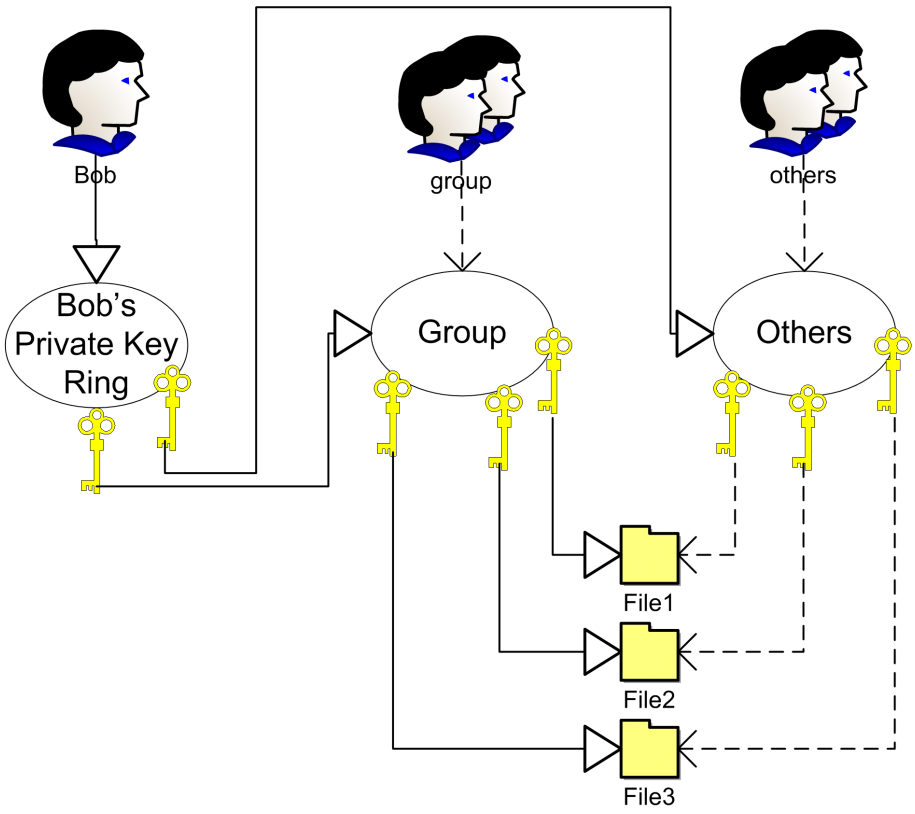

Figure 4.1: Access Groups with Key Rings. The dotted lines represents read-only access.

several access control models in the same system if applicable. This allows some users to use one access control model whereas others might need another.

Although many access control models can be handled with key rings, the Bell-LaPadula[BL76] cannot be handled without changes to the implementation. The reason for this, is that the Bell-LaPadula model is a so called "no read up, no write down" model. Since the symmetric key is used for encryption of the data, it is not possible to allow someone to write to a file, without being able to read it.

A Bell-LaPadula variant could however be modeled. The necessary modification would be to restrict subjects to only write on their highest security level, i.e. they would not be allowed to write up. By doing so, no information will flow from higher security levels to lower security levels.

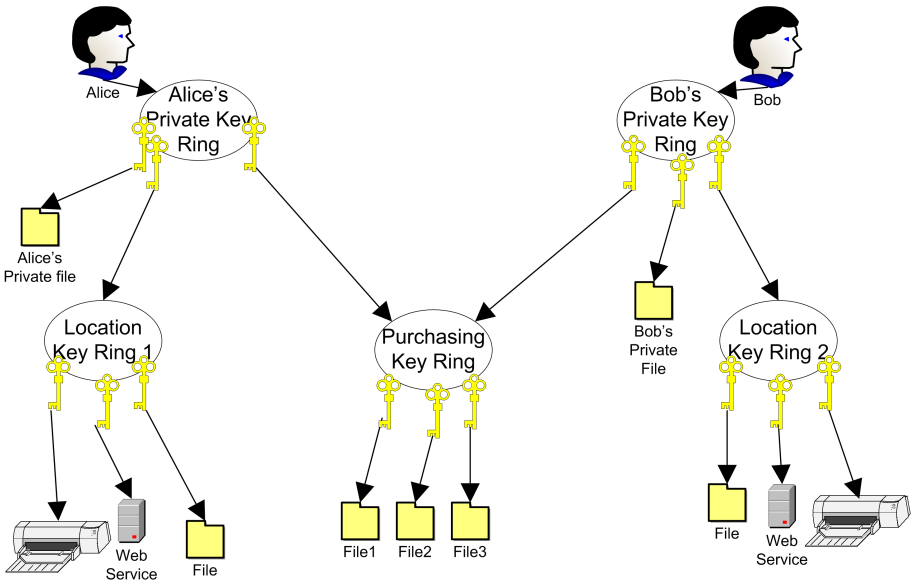

Figure 4.2: Role Based Access Control with Key Rings.

4.4 Link Keys

Whereas cryptographic access control is a mechanism to control access to resources, key rings are closely related to the chosen access control model. The reason for this, is that access to a key ring, always will allow a user to use all keys in the key ring. This is not always satisfactory, because it might be more logical for the user to access the keys from another key ring, i.e. separate the way the files are accessed/viewed from the access control model implemented by the key rings.

A user could of course copy or import keys to other key rings, but by doing so the access rights to the key changes. Another problem with copying keys, is that they loose their relation to certain key ring. This can cause problems if the key needs to be changed for security reasons. Because the copied key has no relationship with the original key it will not be changed together with the original key. A keys copied to other key rings will therefore a some point be invalid which is annoying for the users.

To make the way the users handle the keys more flexible and avoid invalid keys *Link Keys* are introduced. Basically a link key is a reference to another key[3]. It

[3]Files have unique identification numbers on each server, so a file identification number

is similar to creating a “symbolic link” in the Unix filesystem or a “short cut” in the Windows environment, but with the differens that it does not contain the path of the key pointed to, only a reference number to the key/file. The link key only contains information about the original key and not any cryptographic keys. If the cryptographic keys in the original key is changed the Link Key will still be valid, because it only refers to the original key.

Link Keys can therefore freely be placed anywhere users wants to, without security considerations[4]. If a original key is changed or moved the Link Key will still refer to the original key number and can therefore be considered to be automatically updated.

In figure 4.3 Alice has created link keys in her private key ring to her two most used files. This enables her to access the files easier and faster than normal.

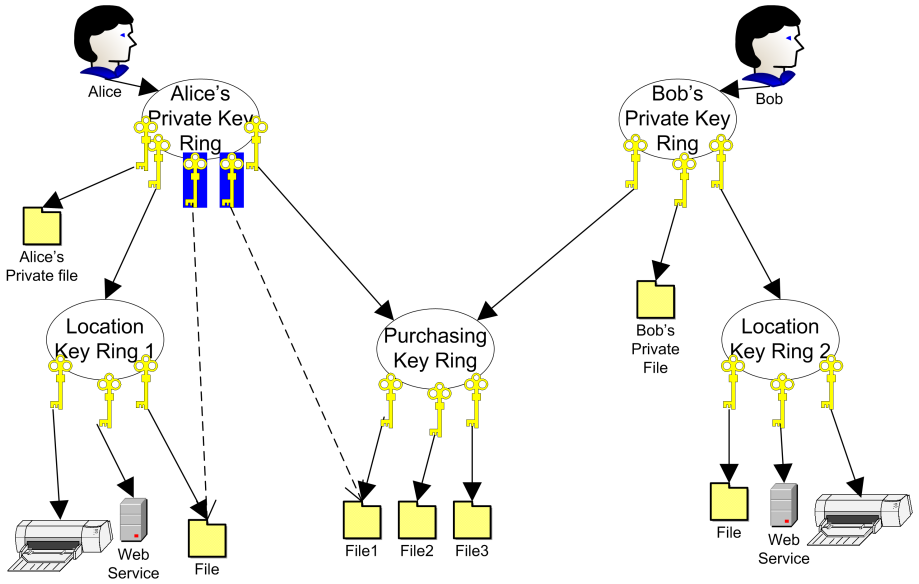

Figure 4.3: Key Rings with Link Keys. The dotted line indicates link keys.

A Link Key is not able to contain the path to the key it is a reference to, because the path will be different depending on which key ring it is placed in. Instead the Link Key should seach for the correct Key (by Id) in all keys and key ring, which the user has access to. Such a search could be a *Depth-first search* or a *Breadth-first search* ([THCR99] page 469-485).

and server id, will be a unique identifier for a file, and therefore also for its key.

[4]The users should however still try to maintain some discipline in the structure of the files.

4.5 Key Rings in a Wider Perspective

In an operating system access is controlled on more than just files. Services such as printers, remote services, web services etc. should of course also be handled in a distributed operating system. Key rings should therefore also be able to handle key to other resources than files.

This is why the definition of CAC-keys is kept rather large and key rings only are described to contain CAC-keys. By doing this the key scheme is more flexible and can be used in a much wider set of applications than ones only focusing on files.

Because keys represent a capability a user can grant it to other users or servers. This can be done by adding a key to a shared key ring allowing users of the key ring to use the key. However, Hagimont et al. [DH96] suggests that capabilities also could be granted to servers in a different way. A print server could for example recieve a key instead of a file, and use the key to retrieve the file. With this scheme a user will not have to download and decrypt a file, before it is encrypted and sent to a server. Instead, it is only necessary to encrypt the key and send it to the server.

These more general ideas about keys will be will be used to design and implementation a key handling that is more general than earlier proposals.

4.6 Summary

In this chapter CAC- and Link Keys was introduced. Besides focusing on the keys, the chapter discussed different ways to handle the keys and files, and introduced key rings.

Whereas cryptographic access control is a mechanism to control access to resources, key rings can be applied on the level above to implement a certain security model such as role based access control or user groups. This is a strong feature of key rings, because it allows administrators to create different access control models, without changing the implementation or doing any programming. It is even possible to use several different access control models in the same environment and actually also to the same files.

Chapter 5

Design

The main purpose of this project is, as earlier described, to design and implement a key manager for the CryptOS project. The earlier chapters discussed cryptography and the basic idea behind key management in cryptographic access control. In this chapter a system design will be proposed, analysed and discussed.

Implementing a key manager will not be enough in itself. First of all, there should be some sort of interface for the users. Because the implementation has to be in the C programming language, it is not within the scope of this project to design and code a graphical user interface. Instead a command line shell to the key manager will be implemented.

In order to demonstrate and evaluate the Key Manager, it is also necessary to design and implement the different kinds of servers that the key manager should be able to communicate with. In this project, it has been decided to implement a file server, two different print servers and a *service server*[1].

Figure 5.1 shows an overview of the design. From the command shell, it is possible to call functions in the Key Manager, that then contacts the servers or local drives to perform the requested command. The *Print Capability Server*

[1]The service server is an example of how the Key Manager can make encrypted communication to servers through channels such as web services and remote functions calls

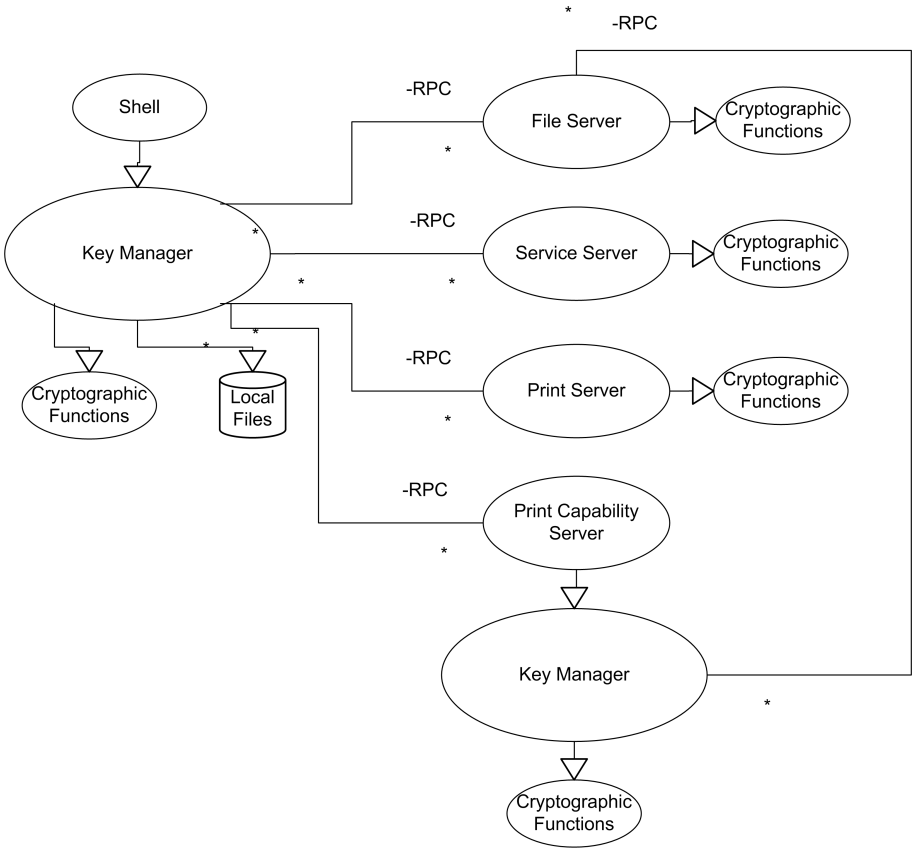

Figure 5.1: Overview of the seven components: Key Manager, Shell, File Server, Service Server, Print Server, Print Capabilty Server and Cryptographic Library.

uses the Key Manager to download a file from a received CAC key. The Key Manager handles all contact with the File Server (see more in section 5.7).

The Key Manager should handle all encryption, decryption, verification etc. so the commands should be as easy as possible for the user. In a real life scenario, many instances of the different components will exist simultaneously. The Key Manager will normally be an integrated part of the local operating system or an application running on the operating system[2]. Depending on the distribution of the operating system, background services such as logging etc. might also use the key manager to handle files.

[2]A key manager running as a normal application will be vulnerable to side channel attacks[DAOT05].

In the following sections the different parts of the project will be analysed and designed.

5.1 Key Manager

The Key Manager should allow users to browse key rings in a way similar to browsing traditional directories. Therefore, it should be possible to view the keys in the current key ring, open a key ring or file, and go back to the previous key ring, i.e. a level up in the tree structure. Furthermore, it should be possible to create and update key rings. These operations result in the following requirements for operations on the key rings:

Open a Key Ring opens or loads a key ring from a server or local drive. Encrypted key rings should be decrypted and their integrity should be verified.

View Key Ring views the contents of the loaded key ring, i.e. all names of the keys in the key ring. This function should give the user an overview of the loaded key.

Previous Key Ring should leave the current key ring and go back to the previous key ring, i.e. the parent key ring.

Update Key Ring is necessary because the user always works on a downloaded, decrypted version of a key ring. If he makes changes, the key ring on the server should be updated. This operation can be called automatically whenever a download key ring is modified.

Create Key Ring should allow a user to create a new key ring in the current key ring (Equivalent to `mkdir`). When a new key ring is created on the downloaded key ring, it should be updated.

Besides handling the key rings, the Key Manager should also handle keys and their corresponding files. It should be possible for the user to create, update and download files to which he has the correct keys. The Key Manager should handle all encryption, decryption and verification within the key manager, to ease the usability for the user. The user should, of course, be informed if problems occur, but in general, he/she should not have to worry about performing the correct encryption steps. This means that all encryptions should be handled in the lower levels of the Key Manager and not be accessible during normal use.

Therefore, the key operations should be as follows:

Create Key creates a key in the current key ring. This action should normally be used when a new file is created and uploaded. When a new key is created in the current key ring, the current key ring should be updated as well.

Get Key should download the requested key so the user can display its data.

Remove key removes a key from the current key ring. When this action is performed, the system should consider whether the file or key ring that the key represents should be deleted as well. In addition, the current key ring should again be synchronised with the key ring on the server.

Renew Key generates a new key. To make the new key affective, it is necessary to download and decrypt the corresponding file. The file can then be re-encrypted with the new key and uploaded. The "Renew Key" function could be recursive, so if it was called on a key ring, all keys and key rings in the key ring would be renewed. This would be quite a resource consuming task if the key ring has many sub keys. Although it is recommended to have keys in a tree structure, it will be possible to make circular designs with the key rings, which could cause recursive function like this to enter infinite loops.

Import/Export Key is used to import and export keys. This replaces traditional `copy`(`clone`) and `move` functions. Import/Export are used instead of copy and move, because key rings are not as easy to manipulate as files. To move a key from one key ring to another is quite difficult, because all key rings involved needs to be downloaded, verified, decrypted, updated, encrypted, signed and uploaded. Furthermore, it is difficult to specify source and destination key rings, because key rings do not have the same structure as files (they can, for example, be circular). Thus, it is decided to only allow export and import of keys. In general, it is not recommended to have more than one of the same keys in the system, because this can cause problems with revocation of keys. It should, furthermore, be possible to export keys that only allow reading of a file from a "write" key.

Export as Link Key allows users to export a *Link Ley* to a specified key (see more in section 4.3). The *Link Key* can be imported similarly to a normal key and be placed in a given key ring.

The main purpose of the Key Manager is to provide access to files and services. Hence, it should allow easy access to these resources, and the following operations should be available for the user:

Create File creates and uploads an encrypted file onto a server from an already existing file on a local drive specified by the user. The operation should also create a key to the file in the current key ring.

Download File downloads and decrypts a file from a given key.

Update File replaces an existing file on a server with a newer version of the file from a local drive.

Request sends an encrypted request to a server, and receives an encrypted reply using the symmetric key in the key ring.

Print a File prints a file on a print server, i.e. transfers the file to the server and prints it. It should be possible to communicate with the print server using requests to cancel print jobs and list printer queue.

Print Capability sends an encrypted key to a Print Capability Server, which then is able to fetch the file and print it.

The Key Manager should have a detailed API that allows creation of other applications on top of the key manager. The applications could be rewritten system functions (if a distributed operating system is desired) or perhaps graphic user interfaces programmed in higher level programming languages.

5.2 Shell Extension

The Shell extension should be an extension of the existing shell (terminal) in Linux. The shell should allow users to use standard Linux commands, as well as commands to the Key Manager.

The Shell should be able to parse simple command lines and call the correct functions in the Key Manager. To distinguish normal shell commands from commands to the Key Manager, a special escape character will be used when traditional commands are used.

The new commands are:

`open`	Opens a key ring.
`back`	Opens the parent key ring.
`view`	Views a key ring or key.
`create`	Creates a file and corresponding key from a local file.
`createkr`	Creates a new key ring.
`update`	Updates a file with newer version from a local file.
`download`	Downloads a file.
`changekey`	Changes a key, i.e. replaces the old key.
`import`	Imports a key from a local file.
`export`	Exports a key to a local file.
`exportlink`	Export a Link Key to a local file.
`request`	Makes a request on a Service- or Print Server.
`print`	prints a file on a Print Server.
`printcap`	sends a key to a Print Capability Server.
`remove`	Deletes a key from the key ring.
`quit`	Quits the shell extension.
`help`	Displays all commands and a help text.

Note that not all the functions from the Key Manager is available for the user. Functions such as `Get Key` and `Update Key Ring` are called within the Key Manager when necessary, so users will not have to worry about these. Furthermore, encryption is completely hidden from the user. He will only be informed if an encryption or verification is not successful, i.e. if he downloads a file with a depricated key and the verification fails.

5.3 Cryptographic Library

The cryptographic library should provide access to the algorithms described in the earlier section. The purpose of the cryptographic library is, therefore, to provide a set of functions that simplifies the handling of encryption algorithms. The functions should configure the cryptographic algorithms and ensure that the key sizes, modes of operation etc. is the same for all components. If it is necessary to replace or change the settings for one of the cryptographic algorithms, it can be done in the cryptographic library. This will then be affective for all components. The interface should, therefore, be as simple as possible, so that everything is handled in the library.

The Key Manager handles both files and requests (messages). The file servers do not know the symmetric key and it is, therefore, necessary to sign the files in order to ensure that the file servers are able to verify the integrity of the files. When the Key Manager needs to communicate with a Service or Print server,

Function	Algorithm	Description
Encrypt File	AES	Encrypts a file using a key
Decrypt File	AES	Decrypts a file using a key
Sign File	RSA and MD5/SHA	Signs a file
Verify Signature	RSA and MD5/SHA	Verifies the signature of a file
Create Key Pair	RSA	Creates a key pair
Create Key	(AES)	Creates a symmetric key
Encrypt Message	AES	Encrypts a text message
Decrypt Message	AES	Decrypts a text message

Table 5.1: Cryptographic Library Functions

the server, in order to act on the request, will already know the symmetric key, i.e. the CAC key is shared between the Key Manager and Server. It is, therefore, not necessary for the Key Manager to sign requests. A corrupted request will not be readable after decryption.

Table 5.1 contains the functions that should be accessible for the other components. These functions should handle all contact with the cryptographic algorithms.

5.4 File Server

The File Server should be able to receive incoming connections from the Key Manager, and handle four different kinds of requests: Create, Update, Download and Delete.

When files are created the server first verifies the signature. If the signature is correct, the server saves the file, public key and signature. The files can be saved in many different ways, but it is suggested that the files are given a unique identification number. That way, the file, public key and signature can be saved in different files that all include the same identification number.

When a file is updated, it is important that the server verifies the signature with the existing public key, because it would otherwise be possible to overwrite existing files. A naive implementation of the server will be open to replay attacks, because old files can be uploaded again and used to overwrite newer files. It is, therefore, suggested that the server implements log to avoid these attacks (more about this in section 7.3). If the signature is correct, the new file and signature can replace the old ones.

The server should also allow users to download files. Since access control is handled by the key rings and cryptography, the server does not have to verify the authenticity of the user. Therefore, it can send the files right away. It should be possible for the user to request the file, signature or public key. During a normal file request, it will only be necessary to send the file and the signature. The user should already have the public key. The public key does not, however, reveal any confidential information and has already been transmitted in clear text during the creation of the file. It is, therefore not a problem to send the public key if requested. The public key can, among other things, be used to verify that one is in possession of the correct key, without having to download the entire file and signature.

Finally, it should be possible to delete files. To delete a file, a user should be able to demonstrate that he has the correct symmetric and public key, i.e. "write" authorisation. This should be done without disclosing any of the keys to the server. This can be done by encrypting and signing a special file or text string, for example an empty file.

5.4.1 File Hierarchy

Because the file server has to handle many files with unique identification numbers, it would be a good idea store the files in different directories indexed with the identification numbers.

If the server has to handle 1.000.000 files with numbers from 0-999.999 it could create 1000 directories using numbers from 0-999. These would then represent the first three numbers of the identification number. To find a file, one would only need to search twice through 1000 files/directories, instead of through one million files. This could, of course, be further improved by creating fewer initial directories and creating subdirectories in the initial directories. The optimal solution between directories and files depends on the search time compared to the time it takes to open a directory.

5.5 Service Service

The Service Server should be able to accept and reply to incoming requests encrypted with one or several different keys. The content of the reply is in this project of minor importance, but should illustrate that the incoming request was understood and reacted upon.

In a real life scenario, this server could be expanded to handle a certain task that can be invoked from a remote client. The task could be to perform a computational task or a specific service. This could, for example, be a server administrating resources in an organisation. Users with the correct key would then be able to book rooms, projectors, etc. and view the current allocations for a specific resource. By using different keys it is possible to specify which resources a given user has access to.

5.6 Print Server

The print server should be able to receive both files and requests with instructions. Possible instructions would be to print a file, cancel a print job or display the print queue.

The file upload should be similar to the uploads to the File Server, except the Print Server should have a fixed cryptographic key, which should be used for encryption of the files. The Print Server needs to possess the key (opposite the File Server) because it needs to be able to decrypt the file before it is printed. The Print Server also needs to be able to receive and react on requests. This part can be handled similar to the Service Server.

5.7 Print Capability Server

The Print Server described above receives an encrypted file, decrypts it and prints it. If a user wishes to print a file from a key ring, he will need to download (and decrypt) the file, find the key to the printer, encrypt the file and send it to the printer. This is of course not an optimal solution for the user or the network, because of the unnecessary network traffic and cryptographic function calls. Instead it would be much smarter to use Hagimont's *Hidden Software Capabilities*[DH96]. A CAC key is essentially the same as a *capability* and a user can therefore send the capability to the server instead of the file. The user already has the CAC key, so the only thing necessary is to encrypt it and send it to the server. The *Print Capability Server* can then receive the CAC key, decrypt it and retrieve the file using the information stored in the key.

With a Capability Server like the one desribed above the user replaces two file transportations and cryptations with one key transportation and cryptation. Because the key usually will be smaller than the file, the necessary computation and network traffic is reduced. The server needs to perform a bit more, but it

can wait with the file download until the printer is ready and the network is less occupied.

To handle the CAC key received from the client, the server needs a Key Manager to download and decrypt the file. The Print Capability Server will therefore need to build on top of a Key Manager (as shown in figure 5.1). The Key Manager will then handle all communication with the File Servers and the decryption of the file.

The Print Capability Server illustrates two important features of the project. First of all, it illustrates how the system can be used to handle capabilities, in some situations, smarter than files. This can used for other purposes than just file printing, and is applicable for more general situations where a user wishes to allow a server to perform a task using one of his capabillities. Secondly the server illustrates how the Key Manager can be used by other applications to handle the CAC keys and download and decrypt files.

5.8 Summary

In this chapter, the general design of the project was outlined. The different components as well as their connection to each other were described. The project consists of a Key Manager, a shell extension, a File Server, a Service Server, a Print Server, a Print Capability Server and a cryptographic library.

The chapter described the components function in the project and basic requirements for the components. The requirements were built on the previous description of cryptographic access control and key management. The actual implementation was not discussed, but some design suggestions were proposed.

Chapter 6

Implementation

In this chapter, the implementation of the project will be described. The implementation is built on the design from the previous chapter. Figure 5.1 will be used as reference for the overall design. The seven parts[1] of the project will be described with references to the source code.

Because the project is a part of the CryptOS[Cry06] project, it was necessary to implement the project in the `C` programming language. The idea with CryptOS is to design and implement and entire operating system build on cryptographic access control. The current plan is to build the operating system on top of an existing kernel like the L4 micro-kernel[Gro06]. These kinds of micro-kernels only allow development in `C` and assembler. `C` was, therefore, the most obvious choice for the project.

This chapter first describes the data structures used in the project. Then, the implementation of the different parts from the design section will be described from the bottom up, i.e. the cryptographic functions, the Key Manager, the servers and then the Shell build on top.

[1] The seven parts of the project are: Key Manager, Shell Extension, File Server, Service Server, Print Server, Printer Capability Server and Cryptographic Functions.

6.1 Data structures

In the previous chapters, keys and key rings were introduced. To handle this in `C` it is necessary to define a data structure that is appropriate for the data they contain. Furthermore, RPC connections are used for data traffic and requires that the data transmitted is contained in data structures. In this section, these different data structures will be described.

6.1.1 CAC Keys and Key Rings

The Key Manager's most important data structure is the `CAC Key`. The CAC Keys are handled in the key manager and grouped in key rings to allow for key ring based access control. The table below(table 6.1) show the attributes within a key and their data types.

Attribute	Type	Description
sid	int	A unique key identification number.
type	int	Type of key (Read/Write/Local/Keyring/Print/ Link/Service etc).
owner	char[]	The contact person for the key.
filename	char[]	The file/display name.
ip	char[]	The IP address
path	char[]	The path if the key is local.
symkey	char[]	The key for symmetric encryption.
pubkey	R_RSA_PUBLIC_KEY	The public key.
prikey	R_RSA_PRIVATE_KEY	The private key.
next	key*	Pointer to next key.
prev	key*	Pointer to previous key.

Table 6.1: The contents of a key and data types.

This project only uses the key described in table 6.1. This key data structure is used to handle keys to files, key rings, services, and printers. The files and key rings can be on a server or local. As well, the keys can be *read/view* or *write/change*. All this information is handled by the `type` attribute. The different types are enumerated with the following names:

{`CHANGE_KEYRING, VIEW_KEYRING, CHANGE_KEY, VIEW_KEY, WEB_KEY, PRINT_KEY, LINK_KEY`}

There is no attribute indicating whether a file is on a server or on a local drive. For this purpose, the `ip` attribute is used. If it is a local file, the attribute is set to `0`. The private and public keys are defined by the implementation of RSA used in this project (see more in section 6.2). These two types of keys are `structs` and contain data such as key size, exponent, and modulus. The private key has additional information about the prime exponent etc., so that the implementation can use the Chinese remainder theorem for faster signing. *View* keys do not use the private key, and link keys have no keys at all, since they point to another key (more about this in next section).

The final two attributes are pointers to other keys, specifically to the previous and the next key. These two attributes allows the keys to be linked together in a linked list. It would have been possible to neglect these two attributes and create a linked list structure that contains a key and two pointers. Doing so could save a little memory, but the current solution makes it very simple to handle keys and the programmed code easier to understand.

The keys are defined under the cryptographic functions in `keys.h`. The methods to maintain the key rings are under the Key Manager in the `KeyRing.c` file.

6.1.2 RPC Structures

RPC connections require structures to be defined for communication between servers and clients. RPC connections only accept pointers to data, and therefore a pointer to a `char` array will be interpreted as a pointer to the first `char`.

In this project, it was necessary to define the structure for RPC communication as follows:

`stream` contains a block of data to add to a file (the `buff` attribute). In addition, it has information about the session id (connection number between server and client) and the size of the block of data.

`data` is similar to the `stream` element, but is only used for server replies. Therefore, it does not contain a session id. The `buff` attribute has a fixed length.

`dataElem` is used to send to a file server when a file is uploaded. The data element (`dataElem`) has session id, file number, hash (signature) of the file, the public key (key size, modules, exponent) and information about whether to create a new file or update an existing (the `type` attribute).

`request` is used for the two way communication between the key manager and a Service Server. The structure contains the actual request (`text`) and information about which symmetric key to use(`KeyID`). The `KeyID` allows the Service Server to handle several tasks with different keys. The client should specify which key to use.

An additional structure is defined on the Printer Server to handle the list of print jobs. It has a `jobId` and a pointer to the next `jobId`. The structure can, therefore, be used as a simple linked list.

6.2 Cryptographic Functions

Before the Key Manger can be implemented, it is necessary to have a cryptographic fundament that the Key Manager can used. It is also vital to have functions for symmetric encryption, asymmetric encryption and a hash function for digital signatures. Because the implementation uses RPC connections, it is further necessary to have a function that can convert encrypted text to base64[2] (More about RPC in section 6.4).

The requirements from the design chapter yields the following necessary functions:

1 Encrypt a file.
2 Decrypt a file.
3 Sign a file.
4 Verify a file.
5 Encode a file to base64.
6 Decode a from base64.
7 Encrypt a text string.
8 Decrypt a text string.
9 Encode a text string.
10 Decode a text string.
11 Create a symmetric key (AES).
12 Create a asymmetric key pair (RSA).

Because encryption and encoding always need to be performed at the same time, this can be combined in one function. The same goes for decryption and decoding.

[2]Base64 is a conversion of binary data to ASCII characters.

Implementing cryptographic algorithms is a very difficult task. First of all, it is very easy to make minute mistakes that completely undermine the security of the algorithm. Secondly, cryptography is often a computationally heavy task and it is difficult to write perfect and efficient code. Therefore, it was decided to use implementations of the algorithms that already exist and were tested. For AES, this project will use a FIPS-197 compliant AES implementation by Christophe Devine called *AES Crypt 2 ver. 1.0*. For RSA, the RSA Laboratories reference implementation called *RSAREF* will be used. Finally, for base64, it was decided to use Bob Trower's implementation from the *Crypt Data Packaging* project. The AES and base64 implementations are both used as full external programs to convert files. They are included in the source code so that their sub-functions can be called when a text string is encrypted, for example. The RSA functions are only included in the source code, but can be called directly on files.

To handle all these encryption functions, it was decided to implement a set of cryptographic functions which are able to perform the necessary tasks. Figure 6.1 illustrates the relationships between the cryptographic functions. The `FileEncrypter` is the central part of the functions and the only one that needs to included to use the functions.

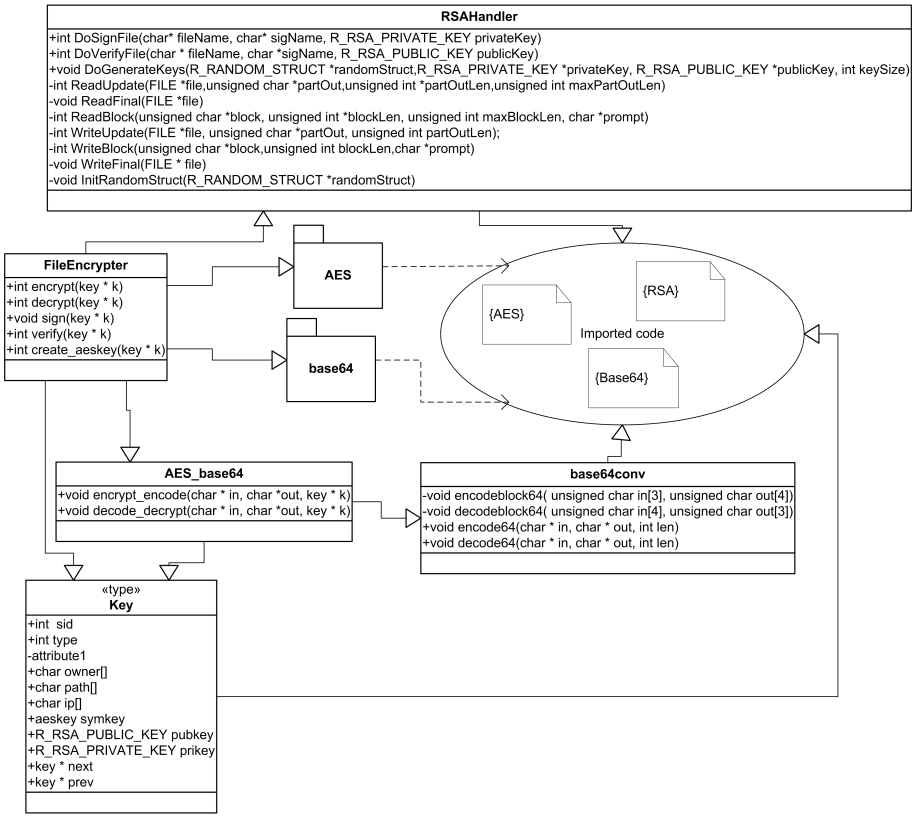

Figure 6.1: Overview of the cryptographic functions.

The implemented cryptographic library also includes the definition of the CAC key. The rest of the project uses this definition, so if, for example, one wants to change the key size, it can be done here. The CAC key implementation has an enumeration of the different types of keys and the following implementation of the CAC key:

```
0   typedef unsigned char aeskey[17];
1   struct key_element{
2     int  sid; //Key ID
3     int type; //Types of keys: ring, write or read
4     char owner[MAX_NAME_SIZE];
5     char filename[MAX_NAME_SIZE];
6     char path[MAX_PATH_SIZE];
7     char ip[15];
```

```
8     aeskey symkey; // Symmetric key to the file
9     R_RSA_PUBLIC_KEY pubkey;
10     R_RSA_PRIVATE_KEY prikey;
11    key * next;
12    key * prev;
13  };
```

In line 0 the AES key is defined, so it can be used by the rest of the project. It is currently defined as a 16+1 byte `char` array which is equivalent to 128 bits + 1 byte. The last byte can be used to insert a string terminating character. The definition of the public and private keys are taken from the RSA implementation to avoid problems converting keys between the project implementation and the RSA implementation. The rest of the key definition can be understood directly from the comments in the source code.

The cryptographic functions are implemented as follows:

`int encrypt(key * k)`
: The `encrypt` function is used to encrypt and encode a file. It uses the external programs `aes` and `base64` to perform these tasks. It takes a key as argument, which contains a file name and a symmetric key. It then creates a command line string with the path to the `aes` file name and symmetric key. By calling the standard `C` command `system()`, the command string is executed in the console. The output of the encryption is a file with the same name as the original, concatenated with "`.crypt`". The encoding process to base64 is called on the new file using the external program `base64` in the same manner as the encryption procedure. Converting to base64 does not require a key, because it is only a conversion to avoid special characters.

`int decrypt(key * k)`
: The purpose of this function is to do the opposite of the `encrypt` function, i.e. to decode and decrypt. This done in the opposite order calling the same two external programs with an argument specifying to decode and decrypt respectively.

`void sign(key * k)` and `int verify(key * k)`
: These two functions call `DoSignFile()` and `DoVerifyFile()`, implemented in the *RSA-handler*, with the correct arguments.

  The RSA-handler has been implemented to handle signing, verification and generation of keys. It reads and writes from the correct files and uses the functions `R_SignInit`, `R_SignUpdate`, `R_SignFinal`, `R_VerifyInit`, `R_VerifyUpdate` and `R_VerifyFinal` from the RSAREF implementation.

When a signature of a file is created, it is placed in a separate temporary file, which can then be used by the Key Manager.

`int createSymKey(key * k)`
This functions generates a new key and replaces an existing symmetric key in a CAC key. A symmetric key is 128 bits in the current implementation, which is equivalent to 16 bytes or 4 integers (of four bytes). To generate a new symmetric key, it is, therefore, necessary to generate four new integers and combine these to create truly 128 bits random bits. This is done by using the math function `rand` to create an integer four times and copy each part to a part of the symmetric key.

`void encrypt_encode(char * in, char *out, key * k)`
This function encodes and encrypts a text string of arbitrary length. The function has been implemented using parts of the source code from the two external programs `base64` and `aes`.

For `base64`, a sub routine that takes three characters as input and returns four encoded characters has been copied to the source code. A function that takes an input of arbitrary length and returns an encoded one has been written on top of the imported code. The output will be 4/3 times the length of the input, and this memory space must be allocated by the function calling the new routine.

The task is a bit more difficult for the `aes` part. In order to make the encryption more secure, it was decided to use the Cipher Block Chaining mode, where the result from one block encryption is '*exclusive or*'ed with the next block before the next block is encrypted. The function, therefore, calls `aes_encrypt` with one block at the time, and uses the result as the `IV`[3] parameters for the next.

`void decode_decrypt(char * in, char *out, key * k)`
This function has been implemented to called the decrypt and decode functions on a text string. The function has been implemented similarly to the `encrypt_encode` function with equivalent sub routines.

Implementing CBC for decryption is not a straightforward task because it is necessary to continuously keep the encrypted block from the previous decryption to decrypt the next block. To constantly have the previous cipher text block, the text is copied to a local variable before decryption[4].

`void DoGenerateKeys(R_RANDOM_STRUCT *randomStruct, R_RSA_PRIVATE_KEY *, R_RSA_PUBLIC_KEY *, int keySize)`
This function is used to generate a RSA key pair. The function call is a

[3]The `IV` parameters are '*exclusive or*'ed with the plain text blocks just before encryption. In ECB they stay the same throughout the encryption.
[4]The decryption algorithm overwrites the incoming cipher text with the plain text.

direct call to the *RSAREF* implementation and will not only find the two exponents `p` and `q`, but also the two primes, the two prime exponents and the coefficient, so the Chinese Remainder Theorem can be used for faster encryption/signing.

6.3 Key Manager

The Key Manager is the most essential part of this project. Its job is to handle the CAC keys, key rings, and communication between the different types of servers. The Key Manager consists of seven different parts that each handle their own responsibilities. Besides the seven parts, the `define.h` defines the different constants in the project. Figure 6.2 illustrates the relationships between the seven parts.

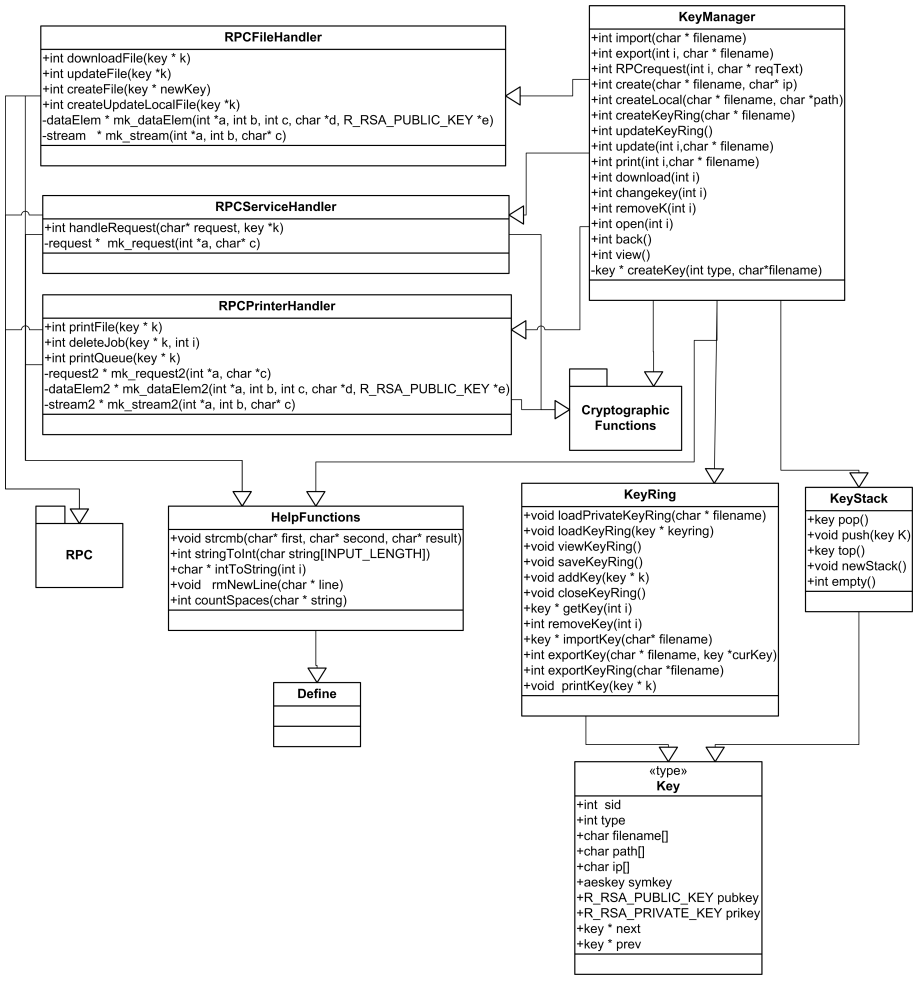

Figure 6.2: Overview of the Key Manager.

The next sections will describe the different parts in detail from the bottom up.

6.3.1 Define.h

In `define.h` the different constants used by the Key Manager is defined. These constants make it easy modify the different sizes used in the project. If, for example, a new attribute is added to the CAC key, the `LINES_PR_KEY` constant should be increased by one. The constant is used when keys are saved and read from files. Changing the constant affects all necessary areas in the project, therefore using constants makes it easier to modify and understand the source code. Table 6.2 describes the different constants used by the project.

Constant Name	Value	Description
INPUT_LENGTH	66	Maximum length of input in shell (Should be divisible by 3).
MAX_PATH_SIZE	60	Maximum path length.
MAX_NAME_SIZE	30	Maximum length of a file name.
line_length	710	Maximum line in a file including the newline character. The `R_RSA_PRIVATE_KEY` is 709 characters.
LINES_PR_KEY	9	Number of lines for each key in a key ring, when saved.
BLOCK_SIZE	4096	Block size for blocks in file transfer(RPC).
PRINT	0	Extra print to screen (1 = VERBOSE/DEBUGGING MODE).

Table 6.2: Constants in the Key Manager.

Most of the constants explain themselves. The `PRINT` constant is used for debugging. If its value is not zero, more information will be printed to the screen during execution of the program. This is very helpful during programming, debugging and testing of the project. When a specific part of the project is tested, the `PRINT` constant can be set to a specific value that prints out extra information during execution of that part.

6.3.2 Helpfuntions

The help functions are used by different parts of the Key Manager to facilitate the manipulation of `strings` and `integers`. Their implementations are somewhat trivial and their purposes is only to avoid repeating source code and

to make the source code more understandable. By not repeating trivial source code, it is easier to make changes in one place that affect the entire program, instead of updating many bits of source code located in different areas. Similar functions might exist in the `C` library, but these help functions are implemented specifically to serve the Key Manager in the most beneficial way.

The five implemented help functions are:

`void strcmb(char* first, char* second, char* result)`
: `strcmb` is short for "string combine", i.e. the functions is used to combine two strings into a third string. The function only accepts pointers to the strings and they must end with the string termination character `'\0'`.

`int stringToInt(char string[INPUT_LENGTH])`
: Converts a string to an integer.

`char * intToString(int i)`
: Converts an integer to a string.

`void rmNewLine(char * line)`
: Removes a newline character at the end of a string. This functions is, among other things, used when a key is read from a file. Because each attribute is written on it own line, it is necessary to remove the newline character, when the line is copied to key.

`int countSpaces(char * string)`
: This function is used to count the number of spaces in a string.

6.3.3 KeyStack

Because it should be possible to browse the different key rings, it is also necessary to be able to go back to the previous key ring, similar to the "`cd ..`" command. Key rings are not guaranteed to have a tree structure like a file system because many keys can open the same key ring. It is not possible to determine which key ring was used by a user to access another key ring. Therefore, it was necessary to implement a stack, where old keys to key rings are stored. When a new key ring is opened, the key to the last key ring is added to the stack. If a user wants to go back to his previous key ring, the top of the stack is popped.

To implement a stack, a stack structure that has a key and a pointer to the next element in the stack must be defined. Besides this, it is necessary to implement functions to push elements onto the stack and pop them off again. This implementation is done in `KeyStack.c` and `KeyStack.h`.

The implementation is an extension of a standard stack implementation([McD01], page 84). The implementation has been extended to handle keys instead of integers.

The implementation has six functions, which are all rather straight forward in their implementation. The six functions are:

```
key   pop()
void  push(key)
key * top()
void  newStack()
int   empty()
key * bottom()
```

The `newStack()` function should be called when the Key Manager is initialised. When the stack is initialised, all the other functions can be called whenever necessary. The stack creates a new key element when a key is added (pushed) and frees (unlinks) it from memory when it is returned (popped). By doing so, the stack does not allocate unnecessary memory. When a key is added to the stack and a new key ring is loaded, the old key ring is freed (unlinked) so it only exist on the stack. With this setup, the Key Manager does not allocate more memory than necessary, because old key rings are continuously discarded.

6.3.4 KeyRing

The *Key Ring* part of the Key Manager handles all operations that pertains to key rings. When the Key Manager is initialised, the private key is loaded and handled by this part of the Key Manager. The implemented functions works on the current key ring and make key ring operations simple for the Key Manager. The source code for this part is in `KeyRing.c` and `KeyRing.h`.

Below is a description of the different functions. The implementation details are only described when necessary. Refer to the source code for complete listings of the code.

`void loadKeyRing(key * keyring)`
: This functions takes a key to a key ring as input and load the key ring into the program. The key ring must be download and decrypted by the Key Manager before it can be handled. A key ring is, as previously mentioned, handled as a linked list. The first element in the linked list is the key to

the key ring itself. This key is used if the key ring needs to be updated or pushed onto the stack.

When the first key is set, the private key ring file is loaded. Each line in the file represents an attribute for a key which is handled by a *switch* statement. The `stdio` function `fgets` is used to read one line at the time from the file. The public and private key in each key can not be represented as one line, because it contains special characters. It is, therefore, necessary to read these with `fread`.

`void loadPrivateKeyRing(char * filename)`
: The private key ring is specified with a file name by the user or program at start-up. This function creates a key from the file name and loads the key ring using the `loadKeyRing` function.

`void viewKeyRing()`
: This function is used to print a key ring in the command prompt. It iterates through the key ring, displaying the file name for each key and the type, starting from the second key (index 1).

`void addKey(key * k)`
: This function adds a new key to the key ring just after the key to the ring, i.e. position 1 (zero indexed). The function sets the first key to point to the new key and the new key to point to the previous second key. If the key ring is empty, the `next` pointer is set to `NULL`.

`void closeKeyRing()`
: This function calls `free` on all keys in a key ring. This unlinks all the keys in key ring and deallocates the memory. Before each key is unlinked, the next is found. Otherwise it would not be possible to find the next key because of the linked list implementation.

`key * getKey(int i)`
: Returns a pointer to the key with index i. This function iterates through the key ring for i times.

`int removeKey(int i)`
: Removes the key with index i. This functions finds the keys before and after the key with index i and links them together, i.e. set the two pointers to point to each other. The key with index i is then freed from memory.

`key * importKey(char* filename)`
: Reads a key from a file and adds it to the key ring. The implementation is very similar to the `loadKeyRing` function.

`int exportKey(char * filename, key *curKey)`
: Writes a key to a file. This function is opposite to the import function and uses `fputs` and `fwrite` instead of `fgets` and `fread`.

`int exportKeyRing(char *filename)`
Exports the entire key ring to a file. This function is very similar to `exportKey` as it just exports all keys in the ring.

`void printKey(key * k)`
This function prints a key to the command prompt. This is mainly used for debugging. If the `PRINT` constant is greater than 1, all information in the public and private key is converted to *base64* and printed[5].

6.3.5 RPCFileHandler

This part of the Key Manager handles all file related functions, i.e. upload, create, and download of files on servers and local files. The files must be encrypted and signed before uploading and decrypted and verified after downloading by the main part of the Key Manager. The *File Handler* determines whether a file is remote or local, and uses the appropriate method to download or upload the file. Local files are always created/overwritten (not updated) because the local file system is not able to perform integrity checks.

The File Handler has four main functions which will be described in detail below:

`int downloadFile(key * k)`
The `downloadFile` function first determines whether the file in question is a remote file on a server or a local file. If it is a remote file, it connects to the server and calls the `download_hash` function which downloads the hash for the file. After the hash is downloaded, it is decoded from *base64*[6] and saved in a file called `sig.temp`.

When the hash has been downloaded, the file is downloaded in blocks. Each block has the size of constant `MAX_BUFF`, which is set by the `RPCFile.x` file. The File Handler calls the `download_file` function on the server with a `stream` structure. The structure contains information about the file id and the block number for the file. The server returns a `data` structure with the data from the requested block. This is continued until the size of the returned data is less than `MAX_BUFF`, i.e. the last block is returned. The data is continuously written to a file, and that file is closed when the final block is received.

[5]It is necessary to convert the public and private keys to *base64* because they can contain special characters that are not suitable for displaying on the screen.

[6]During RPC transmission, it is necessary to encode hash values and keys with base64, because they might contain characters that are used as control characters by the RPC connection.

If the file is a local file, it is copied together with the hash value from its local directory to the current directory from which the Key Manager is initiated. This is done by combining *command line* commands in a string which is used to call the external function `system(char *)`[7].

`int updateFile(key *k)`
The function is used to update an existing file on a server. The implementation is quite similar to the `downloadFile` function. It first determines whether the file is local or remote. If it is a local file, the `createUpdateLocalFile` function is called (see below).

If it is a remote file, the function connects to the server and sends a block of data at the time and then finally the hash of the file and the public key. Before the server verifies the received file, it makes sure that the new public key is the same as before. This ensures that users are not able to create a new key pair to a file and overwrite an existing file on the server. If a key needs to replaced, a new file with a new key pair is uploaded and the old one removed.

`int createFile(key * newKey)`
This function is very similar to the update function, except it checks to see if the file already exists on the server and specifies that the file should be created.

`int createUpdateLocalFile(key *k)`
This function creates and updates local files. Because the files are already encrypted from the main part of the Key Manager, this function only copies them from the specified path to the current working directory. This is done by calling the `system` function similar to the `downloadFile` function.

Beside these main functions the File Handler has two sub-functions `mk_stream` and `mk_dataElem` which are used to create `stream` and `dataElem` structures for the RPC connection.

6.3.6 RPCServiceHandler

The *Service Handler* part of the Key Manager handles requests to servers. The requests are text strings which are encrypted (AES) and encoded (base64) before transmission. The command is decrypted, interpreted, and replied to by the server. The reply is decrypted and returned.

[7]The `system` function executes a command in the commando prompt.

If a request is encrypted with a false key, it cannot be properly decrypted by the server and therefore, it is not necessary to perform integrity tests, i.e. only symmetric encryption is necessary.

To encrypt and encode the text strings, the `encrypt_encode` function from the cryptographic functions is called. The Service Handler then establishes a connection to the server specified by the key, and transmits the encrypted request.

An encrypted response is received in return, which is decrypted and decoded with the `decode_decrypt` function from the cryptographic functions and returned to main part of the Key Manager.

The Service Handler has one sub-function, `mk_request`, which creates a `request` structure.

6.3.7 RPCPrinterHandler

The *Printer Handler* handles communication with the two different print servers, i.e. the File Printer and the Capability Printer. The `PrintFile` function is very similar to the `createFile` function from section 6.3.5. It uploads an encrypted file to a File Printer together with its hash value and public key. The server has a fixed set of keys, which can be used to decrypt the file and print it.

The Printer Handler has two functions to handle files on a print Server: `deleteJob` will cancel a job from the print queue, and `printQueue` will retrieves the print queue. The functions are fixed implementations of the request function from the Service Handler, with the only exception that `deleteJob` allows the Key Manager to specify which job to delete from the print queue.

The last main function is `PrintCap`. This function uploads an encrypted key to a Capability Printer. The implementation is analogous to the `PrintFile` and `createFile` implementations, except this function uploads a key instead of a file.

The Printer Handler has three sub-functions `mk_request2`, `mk_dataElem2` and `mk_stream2` which creates `request`, `dataElem` and `stream` structures.

6.3.8 KeyManager

The two files `KeyManager.c` and `KeyManager.h` are the main part of the Key Manager. They combine all the other parts of the Key Manager and provide a set of functions which can be used by other applications (See more about how to use the functions and a list of all the functions in the API section (section A.1)). In this section, some of the functions will be described to give the reader an overview of how they are implemented. Many of them are self explanatory and detailed descriptions of their implementations are not necessary.

The Key Manager has a total of 17 functions which are to be called from other applications. They all return integers. Zero is returned if the function call was successful and another integer value is returned if an error occured[8].

Because the other parts of Key Manager provide most of the necessary sub-functions, the functions in the Key Manager are relatively easy to implement. They are also fairly straightforward to understand from the source. Below is a source code example of a simplified[9] function that creates a file on a server.

Prior to the file being created on the server, a key to the file is created and the file is encrypted, signed, and sent to the server. Finally, the key is added to the current key ring, which then is updated. The implementation does not need to be concerned about whether the key ring is a local or remote key. This is all handled by the underlying code.

```
1   int create(char * filename, char * ip){
2      key * k;
3      k = createKey(2, filename);
4      strcpy(k->ip,ip);
5      encrypt(k);
6      sign(k);
7      createFile(k);
8      addKey(k);
9      updateKeyRing();
10    return 0;
11  }
```

Line 2-4 creates the new key and sets the IP address to the specified server.

[8] Some functions return different values depending on the error so the application using the Key Manager can take appropriate actions.

[9] The function has been simplified for the ease of readability. The actions performed are the same as in the source code.

Line 5-9 performs the necessary encryption, creates the file, and updates the key ring.

As one can see, the implementation is easy because all the necessary functions are available. The implementation is almost just a formalisation of the design described in the section with the Key Manager design (section 5.1).

If a user wants to remove a key from a key ring, it is also a very simple task. Only two function calls are necessary to the underlying functions, one to remove the key from the current key ring, and one to update the original key ring with the modified one.

```
1    int removeK(int i) {
2      if (removeKey(i)) {
4        return -1;
5      }
6      updateKeyRing();
7      return 0;
8    }
```

In line 2, the Key Manager tries to remove the key. If the key is not found, the function returns -1, otherwise the key ring is updated in line 6. The key ring needs to be updated because it is read from a file into memory. In order to make an effective change, it is not enough to make the change in memory. The change must also be reflected in the original file. This is handled by the `updateKeyRing` function.

Besides the public functions, the main part of the Key Manager has two internal functions which are very important:

`key * createKey(int type, char * filename)`
: This function allocates memory to a new key and creates it with the given parameters. It is always necessary to create a symmetric key. The only case where the creation of a public-private key pair can be omitted is for the *Service Keys*. The generation of the keys, both symmetric and asymmetric, are handled by calls to the cryptographic functions.

`key * hasKey(key * curKey, int sid)`
: This function is used to find the key that a Link Key points to. Because keys can be in more than one key ring and the actual path to the key can be different from user to user, it is, therefore, not possible for the Link Key to specify where the original key is. The Link Key can only contain the

original key id, and the Key Manager must search through all key rings to see if it can find the key. This is done by calling `hasKey` on the private key ring. The function searches through all keys in the key ring. First, the keys pointing to files, services, and printers are searched. Secondly, the function calls itself recursively through the first key ring, second key ring, and so forth. If the original key is found, a pointer the key ring is loaded and a pointer to the key is returned. Otherwise `NULL` is returned.

The reason why all keys to files, services, and printers are searched in the key ring before the key rings are searched, is that key rings needs to be downloaded before they can be searched and it is faster to search the already downloaded key ring first.

The current implementation has one weakness. It is possible to create key rings that point to each other, i.e. a loop. If the key rings form a loop, the function will never terminate. At present, it is recommended that key rings do not form loops, but the function could be extended to handle this as well. For example, a list could be implemented to keep track of already visited key rings.

The implementation of the Key Manager has been kept as readable as possible, and it should be very easy to implement further functions or update existing ones. Furthermore, the modular design allows parts of the Key Manager to be replaced, if necessary. The replacement will usually only have to be done in one place and will affect all parts of the Key Manager.

6.4 RPC Servers

To demonstrate how the Key Manager is able to handle different kinds of keys and communication with different kinds of servers, four types of servers had been implemented. The four types are: a *File Server*, a *Service Server*, a *Print Server* and a *Print Capability Server*.

In this section, the implementation of the different servers will be described. Figure 6.3 illustrate the four servers and how they interact with the other parts of the project.

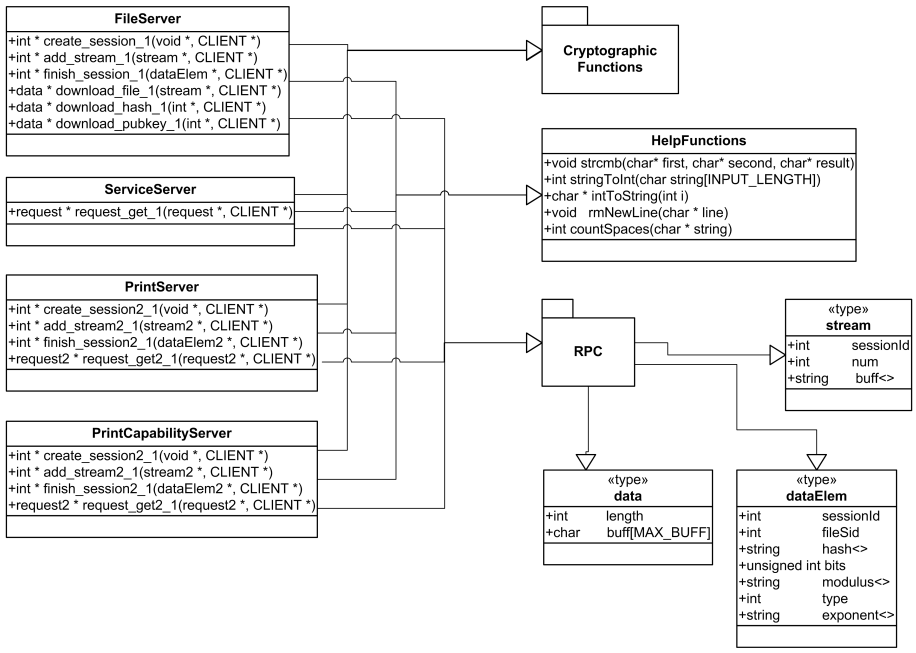

Figure 6.3: Overview of the RPC Servers.

All the servers are implemented using *Sun Remote Procedure Calls(RPC)*[Mic94]. They each have a `.x`-file which specifies their functions and data structures. The `.x`-file can be found in directories starting with *RPC*. These files have been RPC-generated using `rpcgen`. This creates four files all sharing the first part of the file name with the `.x`-file.

The File Server has, for example, a `.x`-file named `RPCFile.x`. Generation of this file gives the four files:

```
RPCFile_clnt.c
RPCFile.h
RPCFile_svc.c
RPCFile_xdr.c
```

In order to make the RPC connection work, the client, i.e. the Key Manager, must include `RPCFile_clnt.c` and `RPCFile_xdr.c`, whereas the server must include `RPCFile.h`, `RPCFile_svc.c`, and `RPCFile_xdr.c`. It is now possible to implement the functions specified in `RPCFile.x` on the server and call them from the client. The client must connect to the server using the `clnt_create` function. Connections are always established by the client calling a function on

the server, which results in a reply sent to the client.

6.4.1 File Server

The File Server is the most essential server because it handles the files. The server handles two kinds for files, namely data files and files with key rings. All files are encrypted and the server is, therefore, not able to distinguish the two types of files. The servers only job is to check the integrity of the files and make them available for download.

The File Server has six functions, which the client can initiate. Below are descriptions of the six functions. The first three are for file uploading and last three are for file downloading.

`int *create_session_1_svc(void *a,...)`
: This function is used to create a session for uploading a file. The function returns an integer which is the session number. The session number is also used to store the files instead of the original file name. The reason for this is that even the file name can contain confidential information, and therefore publishing it might not be a good idea. The server uses the `srand()` to generate a random number and verifies that the number has not already been taken using the `stat` function. Both functions are parts of standard `C`. It then opens the file *(session id).file* for writing.

`int *add_stream_1_svc(stream *s,...)`
: This function adds a block of text to the open file. The block size is defined in the `define.h` file. Before adding the block, it ensures that the correct session id is used. The communication between the client and server is not encrypted, but the file has been encrypted on the client. It would be possible for an attacker to add a block of data to the file during upload, because the communication not is encrypted. This would, however, result in a rejection of the file, because the signature would not match the changed file.

`int *finish_session_1_svc(dataElem *d,...)`
: This function closes the file and verifies the signature (`hash`). The signature and the public key is sent to the server in a `dataElem` structure. Because RPC connections have problems with some characters, the public key and signature has been converted to base64 before transmission. The `decode64` function from the cryptographic library is used to decode the structure from base64. The integrity of the file is verified and the public

key and signature are saved together with the file as *(session id).pubkey* and *(session id).hash*

`data* download_file_1_svc(stream* strm,...)`
This function is used to download a file. It receives a `stream` structure which specifies file id, block size and block number. It then returns the request block of requested block size. Because the server cannot be sure that the next request will be from the same file, it has to open and close the file for each request. It returns the block in a `data` structure which also contains the total number of bytes read. If the total number of bytes read is less than the requested block size, the client knows that it received the last block.

`data* download_hash_1_svc(int* fileId,...)`
This function is very similar to the `download_file` function. Instead of opening the *(session id).file*, it opens *(session id).hash*. Because the size of the hash value is less than the block size, the hash value can be returned in one block[10]. Before the hash value is returned, it is converted to base64[11].

`data* download_pubkey_1_svc(int *fileId,...)`
This function is a copy of the `download_hash` function. The only difference is that it returns the public key instead of the hash value.

This function will not be used during a normal file download, because the client already knows the public key. If the client is not able to verify the signature of the downloaded file, it is possble to download the public key and see if it matches the one already possessed. If not, the key has been changed and the owner of the key should be contacted.

6.4.2 Service Server

The Service Server is an example of how keys can be used to handle other things than files. The Service Server receives an encrypted request, which is decrypted and printed to the screen. It then replies with a naive reply which also is encrypted. The purpose of the implementation is only to demonstrate how a such a server can be created. The source code can easily be expanded to parse the incoming messages and react upon them accordingly (see, for example, the Print Server below).

[10] If the hash algorithm is changed to a new algorithm that uses longer outputs than the block size, this would have to be changed.

[11] This was not necessary for the `download_file` function, because the stored file is already converted to base64.

The Service Server only has one function that can be understood directly from reading the source code. The key to the server is hard coded, but the request sent to the server contains a key id that can be used if several keys are wanted (This is supported by the Key Manager). The server uses the two functions `encrypt_encode` and `decode_decrypt` provided by the cryptographic functions to decrypt and encrypt the messages.

6.4.3 Print Server

The Print Server is used to print a file. Before the file can be Printed, it is uploaded to the server. The Print Server then decrypts the file and adds it to the print queue. The client can call different functions to retrieve the queue, cancel a job and print a specific job from the queue.

The Print Server is a combination of the two previous servers. It has the three functions used for file uploading from the File Server, and an extension of the Service Server to handle the requests. When a file is uploaded to the Print Server, it must be decrypted before it can be printed. It is, therefore, necessary for the server to have a symmetric key and a public key. These keys are hard coded in the server and in the file, but this can of course be modified if severals keys are necessary.

The three functions for file uploading are nearly identical to their respective implementation in the File Server, so further details are unnecessary. When a request is sent to the Print Server, it is sent to an extended version of the request handler from the Service Server. Again, the request is decrypted, but this time the request is matched, using `strncmp()`, to the three different accepted requests. The server has a print queue, which is implemented as an array with the file id for each file. The different requests resolve to specific actions applied to the print queue, and an appropriate reply is returned. If the request could not be interpreted, an error message is returned. Before transmission to the client, the reply is encrypted and encoded to base64 using the `encrypt_encode()` function.

6.4.4 Print Capability Server

The Print Capability Server is an example of a server that is able to receive a capability[DH96] and use the Key Manager to handle the capability. In this case, it can receive a CAC key and download the file using the Key Manager. The file can then be printed on a local printer.

The Print Capability Server has the same functions as the Print Server. However, they are used differently in this case, and the Print Capability Server also uses the Key Manager to handle the received key. The three functions to receive a file are used to receive the CAC key instead.

The first time a connection is established, the Key Manager initiates the stack and loads the private key ring:

```
1     if(startUp) {
2       // Initialise the key manager
3       newStack();
4       // Load key ring
5       loadPrivateKeyRing("default.keyring");
6     }
```

It can then receive the key and decrypt it using the cryptographic functions. When the CAC key (capability) is received, it can be imported into the Key Manager using:

```
if(import(filename)) printf("Error: Could not import key!\n");
```

When the key is imported, the Key Manager can be used to handle the key. In this case the most obvious choice is to download the file:

```
if(download(1)) printf("Error: Could not download file!\n");
```

When the file is downloaded, it can be printed. In this case, the server waits for a `print` request and displays the file on the screen (This part is similar to the Print Server).

The advantages of the Print Capability Server is that the client does not need to download the file, decrypt it, encrypt it with another key and send it to the server. Instead, it can send the key which is already in its possession. It is then the server's job to download and decrypt the file.

6.5 Shell Extension

The Shell Extension is a simple implementation to illustrate how the Key Manager works. The code for the Shell Extension gives a good idea of how the Key

Manager can be included in an application.

The Shell Extension is an extension to the existing Unix shell. It allows users to enter commands that are parsed and used to call the functions in the Key Manager. It also allows user to use existing commands by using an escape character.

The implementation has two parts:

`Commands.c` and `Commands.h`
: These two files provide the sub-functions for the main program. `Commands.h` has an enumeration of all allowed commands. A text command can then be parsed to match one of the allowed commands. This simplifies the code and makes it easier to handle commands.

  `Commands.c` has two functions:

  `int find_command(char input[INPUT_LENGTH])`
  : This function finds the input command. It uses the `strcmp` function to compare the typed text with defined commands. It then finds the correct enumeration for the command and returns it.

  `void do_command(int cmd,char input[INPUT_LENGTH])`
  : Takes an enumerated command and a string of arguments as input. The syntax for the users is:

    `cmd no op text`

    where `cmd` is the command. The second argument, `no`, is the key number, which is used in most commands. The third argument, `op`, is a one letter option which can be used by some commands. The fourth argument is the command text. Below is an example of a command that exports key number 2 from the key ring to a file called `myFile.key` :

    `export 2 myfile.key`

    If the user wants the key to be readonly, this is specified with the `r` option:

    `export 2 r myfile.key`

    To simplify the command parsing, it has been decided that key rings can only have up to 10 keys[12] in a key ring, and that file names not are allowed to have spaces. The command string starts with the first argument. By counting the number of spaces, it is possible to determine the number of arguments. If the argument string has two spaces, it has three arguments, one in position 0, one in position 2,

[12]The maximum of 10 keys in a key ring is only for the simple shell implementation. The Key Manager can handle 2^4 keys in a key ring.

and one in position 4. If it only has one space (the `op` is omitted), it has arguments in positions 0 and 2. If no spaces are found, it only has the `no` argument in position 0. This is a very simple implementation that allows easy command parsing.

The `do_command` function first counts the number of spaces (`args`) and converts the `no` input to a integer(`i`). It then uses a `switch` statement on the enumerated command to find the requested command. Below is an example of how the `export` command is handled:

```
1   case EXPORT:
2       if (args==3) { // Read or link export
3         if (!strncmp(&input[2],"r",1)) { // Read key
4           if(exportRead(i, &input[4]))
5             printf("Error exporting read key!\n");
6         }
7         if (!strncmp(&input[2],"l",1)) { // Link key
8           if(exportLink(i, &input[4]))
9             printf("Error: exporting link key!\n");
10        }
11      }
12      else // Normal export
13        if(export(i, &input[2]))
14          printf("Error: Could not export key!\n");
15      break;
```

Line 2 tests whether the command has two or three arguments. If it has three arguments, the value of `op` argument is tested. After this, the appropriate function in the Key Manager is called on the form:

```
if(export(i, ..&input[2])) printf("Could not export!\n");
```

This is how all calls to the Key Manager are made, i.e. in an `if`-statement that prints out an error message if the Key Manager encounters errors while performing the task.

All functions calls are built on the same principals as the export example. By reading the code, it is easy to see the similarities. It is, therefore, also quite simple to extend the implementation to handle new functions. Note that keys are always kept in the Key Manager and can only be exported to files. As a result, applications built on top of the Key Manager do not have to worry about keys, key rings and their implementation.

`Shell.c`

The `Shell.c` is the main part of the shell. It expects the path to a private key ring as argument when executed. The first part of program prints out a

welcome message, initialises the key stack by calling `newStack()` and loads the private key ring into the Key Manager calling `loadPrivateKeyRing()`.

Next, a `while` loop runs. The first thing the Shell Extention does is to print ‘‘`cmd> `’’. It then uses a while loop to wait for the user to type a command. A soon as the user types a space or newline character, the `find_command` function from the `Commands` part is called.

If the user typed a space after the command, this indicates that the user wants to provide an argument to the command. A new `while` loop is then initialised for the user to type some text and a newline chararter. The second piece of text is considered the argument of the command and is used together with the first found command to call the `do_command` function.

6.6 Compiling and Running the Project

If new development is made to the project, it is necessary to recompile it. This is done using the `Makefile` in the root directory, and by typing `make`¡enter¿ on a terminal console.

The `Makefile` compiles the five parts[13] of the project to executable programs which are placed in their corresponding directories. If new files are added to the project, they must be added to the `Makefile`.

If changes are made to the RPC connections, it is necessary to regenerate the `.x`-files and compile the project again. This is done in their respective directories. If one wants to re-generate the `RPCFile.x`, this is done by typing the following commands from the root directory:

```
cd RPCFile/
rpcgen RPCFile.x
cd ..
make
```

The project uses external programs to perform AES encryption and Base64 conversion of files. If the programs are moved to another platform, it will be necessary to recompile these. This is done by typing:

```
cd Crypt/base64/
```

[13]Shell, File Server, Service Server, Print Server, and Print Capability Server.

```
gcc -o base64 base64.c
cd ../aes/
make
cd ../../
```

Please note that it is not necessary to recompile the project, because these are external programs.

The Shell can be started from a desirable directory by specifying the path to the compiled program (See more in the Shell User Manual in section A.2). The servers must be started from their respective directories, by typing "`./Server`". Several different servers can be run the same computer in different windows.

6.7 Summary

In this chapter, implementation of the data structures and the different parts of the project was described. The most important part is the Key Manager, which is the main the part of the project. To illustrate the purpose of the Key Manager, four different kinds of servers were implemented, together with a Shell extension which can be used to test the Key Manager.

In order to establish RPC connections to the servers, it is necessary to convert the encrypted contents to *base64*, because it can contain special control characters. To provide the cryptographic functions and *base64* conversion, existing implementations were used. The reason for this was to maintain focus on the essential parts of the project, to save time, and to use thoroughly tested implementations.

Chapter 7

Evaluation

In this chapter, the design and implementation of this project are evaluated. This is done through a functional test, a performance evaluation, and a security analysis. The last part of the chapter discusses possible further work, which has been out of scope for this thesis project.

7.1 Test

A complete functional test of the implementation has been carried out. Please see appendix A.3 for screen dumps from the program. The test verifies that the implementation works as intended and that the proposed theory can be used to design and implement a fully functional key management system for cryptographic access control. When the initial key distribution has been completed, i.e. when users have their private key ring with keys to their granted key rings, a file can, in one command, be encrypted, signed and uploaded to a server. Authorised users will automatically gain access to the file, because its key is added to a shared key ring. Although the file is placed on a remote server, the user does not have to worry about the confidentiality of the file, because of the implemented cryptography scheme.

The functional test only proves the concept of the design. This is because the

implemented Shell and, to some extent, the implemented servers are only designed to handle correct requests (to perform a complete structural test of the implementation is out of scope for this thesis project). Although the implementation is not extensively tested, confidentiality can be guaranteed safe because of the cryptographic algorithms used, which have been heavily tested and are approved to handle secret information. An attacker might be able to crack a server, resulting in availability problems, but the contents of the file will remain secret.

The test appendix documents a successful completion of the implemented commands. The test has, therefore, been approved, and the performance of the system will be evaluated in the next section.

7.2 Performance Evaluation

To perform a benchmark test, the implementation has been expanded to include a time measuring mechanism. In the main `while` loop of the Shell implementation, the time is registered just before the `do_command` function is called and just after it returns. The time used to complete the command is then printed on the screen. The modification to the program is shown below:

```
c0 = clock();
do_command(cmd,input);
c1 = clock();
cputime = (c1 - c0) / (CLOCKS_PER_SEC);
printf ("\tElapsed CPU time test:   %f  \n",cputime);
```

A benchmark test was performed on a personal computer running Fedora, with 1.2 GHz CPU and 512 MB RAM. The server is also running Fedora and has a 733 MHz CPU and 256 MB RAM. The two computers are connected via a Local Area Network.

The different commands have been tested on a shared key ring and two different file sizes. The result is shown in table 7.1.

The general performance of the implementation is quite fast, except when files are created. File creation is very time consuming because a new CAC key needs to be generated, i.e. it is necessary to generate a public-private key pair. The other operations are not noticeably slower for a user than normal file operations.

Command	15 KB file	50 KB file	Key Ring
Create	10.14 s	10.17 s	10.10 s
Download/Open	0.02 s	0.07 s	0.01 s
Update	0.06 s	0.06	0.07 s
Export	0.01 s	0.01 s	
Import	0.06 s	0.06 s	

Table 7.1: Time measurement on executing different commands.

Operation	1st sub-tree	3rd sub-tree	6th sub-tree
Download Link	0.2 s	0.4 s	0.5 s

Table 7.2: Time measurement for Link Keys.

To test the performance of the Link Keys, a tree of key rings was created. The Link Key was placed on a sub-tree to the root and the key that it pointed to was placed in three different locations (on the first, third and sixth sub-tree). The time to find the correct key and to download the corresponding file was measured. The result is shown in table 7.2.

Link Keys functions require slightly more time to run than the other commands, but the total time is still less than one second. The key rings used for the time measurements only contained one other key ring and a file. Larger key rings with many sub key rings will be more time consuming. Besides the time consumption problem, Link Keys have other disadvantages. They require a lot of data traffic and send many file requests to the File Server. If Link Keys are used by many users simultaneously, they can overrun the server. It should, therefore, be recommended to use Link Keys only in a limited extent.

The current implementation stores the files as base64 encoded, which require 1/3 more hard drive space than traditional files. The files are not decode from base64 before they are stored, because this would require them to be re-encoded before transmission. This setup was chosen to save computational operation on the server side. Some scenarios might involve limited disk space as opposed to CPU power. In these cases, it would be advantageous to decode the files before they are saved. Besides storing the actual file, the server also stores the hash value and the public key, which are 388 bytes in total. Key rings also require more disk space than traditional directories, because they not only need to keep information about location of the files, but also its keys. Each un-encoded key in a key ring requires about 1 KB of space. A file of size S(in KB) with one key

will therefore require:

$$\frac{(S+1)*4}{3} + \frac{388}{1024}KB \tag{7.1}$$

A 10 KB file will therefore use 15 KB and a 100 KB file will use 135 KB. The total additional space consumption is therefore 33%-50%.

7.2.1 Scalability

The system design is highly scalable, because no central authorities are required. The system does not have any limitations on servers because they are known by their IP addresses. The servers do not need to know each other, because communication only happens from the client to the server and back (except the print capability server, which uses a key with server address). All information necessary to access a file is kept in a CAC key which can be distributed or placed in a key ring. The total amount of files and key rings are only limited to the available hard disk space on the servers.

Because the servers does not need to know of each others existence, and they are not aware of the contents of the files, many different access schemes can be implemented on the same set of servers. The access schemes can be related in some way or have a completely different set of users. Servers can be added and removed as needed, as long as the files are moved and the keys are updated.

By using the API to the Key Manager provided in section A.1, it is possible to develop other applications that uses the Key Manager. Those applications can use the Key Manager to create and access files and key rings on the different servers instead of traditional network drives.

7.3 Security Analysis

The system design and implementation makes it resistant to most attacks. The system does, however, have a few vulnerabilities which will be discussed in this section.

Untrustworthy Server

The system is designed to handle untrustworthy servers. An untrustwor-

thy File Server will not be able to learn any information about the file. This holds for the implemented File Server, but also other developed file servers because no information about the files is transmitted. An untrustworthy server, however, cannot be guaranteed to have the files available for download. In fact, it can delete the files if it needs disk space. If a server deletes an encrypted key ring, it can have consequences for all files in the key ring, because the keys to the files are stored in the key ring. Therefore, a lost key ring can have severe consequences for a system and it is important that key rings are placed on the most trustworthy servers.

Untrustworthy User

Because the administration of authorisation has been decentralised, it is up to the users to administrate the access to the different keys. This can lead to problems if a user is not trustworthy and distributes keys to a person who should not have access to those keys. For the reading part of the authorisation, it is not much different from a centralised security administration, where a user can download a file and send it to unauthorised persons. However, for the write/update part of the authorisation, it is somewhat different because a user can, maybe by mistake, allow another user to modify files which he should not have authorisation to. This would not be possible in a centralised user administration (assuming that the user not is an administrator).

The decentralised user administration can also lead to another problem. Many companies today use segregation of duties, i.e. if a user is allowed to perform a certain task, there is another task he must not be allow to be perform. This is used to ensure that no employee is able complete a business transaction in its entirety, e.g. place and confirm an order. Segregation of duties can be hard to maintain with cryptographic access control, because it is difficult for the users to determine the capabilities of other users. They are, therefore, not able to determine if they, by giving a key to a user, will allow him to perform an unwanted set of tasks.

A last problem with untrustworthy users is that the system is anonymous, i.e. the servers are not able to distinguish between users, as long as they have the correct keys. This makes it impossible for servers to log who performed which changes. If important information is deleted or modified, it is not possible to find the responsible user. The servers can, of course, implement extensive logging. By keeping all old copies of files, it will always be possible to go back to a previous version, but the malicious person can never be found[1]. The key to the file can also be replaced by a new one, but if the malicious user is among the authorised persons, he will receive the new key.

[1] A server can log the IP address of a person performing a change, but because the keys can be freely exported, a malicious user can use a public computer which cannot be traced back to him.

Replay Attacks

Servers are vulnerable to Reply Attacks if they do not implement logging. If Alice updates a file and it is then updated by Bob, an attacker can re-send Alice' update and erase Bob's changes. Therefore, the servers should implement some sort of logging to prevent possible Replay Attacks. Another possibility would be to use a timestamp in the messages, but this will require that clients and servers are synchronised, which sometimes can be a difficult task.

Eavesdropping

All sensitive data is encrypted. Thus, an eavesdropper will not be able to learn any sensitive information. An eavesdropper will, however, be able to perform an traffic flow analysis and monitor when a file is updated or created and who accesses the file. This is a not a serious threat to the design, but can be a threat to some systems.

Lock Files for Updating

It is not possible to lock a file on the server when it is updated, i.e. two users can download a file and make changes to it, but only the last submitted changes will be affective. This is not really a threat to the system, but can cause problems, if files are access by many users at the same time.

Denial of Service Attack

One of the central design points of this project is that the servers do not verify the identities of the different clients. Everyone is, therefore, able to request a file from a server. Since network bandwidth is finite and limited in many cases, a malicious person can request so many files simultaneously that the bandwidth will be fully occupied and no one else is able to request any files.

This attack is a problem mainly if the system is distributed on a completely untrusted network such as the Internet. This risk is mitigated if the system is implemented over a local area network where only authorised users on the network will able to request files. Another possibility would be to implement the servers to use some sort of IP address blocking mechanism, so only authorised IP addresses would be allow to request files. However, this would defeat some of the scalability of the design.

Infected Computer

A computer infected with a program that allows another person to access and monitor all actions performed by a user will, of course, be vulnerable to attacks. The private key ring is not encrypted and could easily be copied and misused. Even if the private key ring was encrypted, an attacker would be able to learn the key when a user opens the key ring.

In general, it is almost impossible to do anything against a computer which is infected by such a program. The best solution is to use "one

time" passwords, but this requires some central administration.

Brute Force

The access control mechanism is built entirely on cryptography. If the cryptography is broken, the system is useless. Because everyone is able to request a file and save the encrypted file, the cryptography should be so strong that it will not be possible to break within a specified time period. In the chosen implementation, AES 128 bit is used. It will, at some point, be possible to break this algorithm (see more in section 2.3), perhaps not within the next 30 years, but one should be careful using the implementation for "Top Secret" information that require longer archival timelines.

Side Channel Attacks

As previously mentioned, Side Channel Attacks can be used to retrieve an AES key if an attacker has access to encrypted information and is able to monitor the CPU's cache. This could be the case if, for example, some sort of log-system was running on the same computer to which the attacker has access. He would then be able to monitor the cache and learn its key. With this key, he would be able to modify the log, which would be a threat to the system. One should, therefore, be careful when giving users access to computers running system services or carefully protect the CPU cache from eavesdropping.

7.4 Future Work

The next step in the CryptOS project is to implement the Key Manager and servers on top of a micro kernel to create a operating system completely based on cryptographic access control. This will yield better performance because the current design is implemented on top of the existing access control. The files are, therefore, protected by two access control mechanisms, both of which require resources.

The use of the implemented Key Manager depends on the micro kernel. It will require the micro kernel to provide a standard `C` library with the necessary functions for the Key Manager. If such a library is not available, it will need to be implemented. Furthermore, the Key Manager should be extensively tested, not only from a Shell, but a structural test of all functions should be carried out to verify that the implementation is completely stable.

Besides the extensive test of the Key Manager, it might be possible to optimise

some routines in order to gain better performance. The biggest performance issue is the key generations of the public-private key pair. Elliptic Curves([Sti02], page 247) could be used to replace the existing RSA implementation. Elliptic Curves are known to have better overall performance than RSA and should provide faster key generation.

In order to provide a fully functional Key Manager in the operating system, it is necessary to have services (the servers in this project) which the Key Manager can use to access files, etc. The implemented servers can be adopted for this purpose. They should be expanded to serve the distributed operating system, and the simple implementations should be improved. Currently, communication is handled via RPC. To improve performance, it might be desirable to use a lower level protocol, but this will require much more development.

The servers are vulnerable to some attacks like the Replay Attacks and Denial of Service Attacks. These could, to some extent, be avoided with logging. If the operating system is to be distributed over open networks like the Internet, these attacks should be carefully considered. In a closed network with a limited user base, a simple logging system could be used.

The current Shell implementation is somewhat simple and does not parse commands perfectly. If the Shell is to be used in a project like CryptOS, it should be improved and should undergo a complete structural test. The Shell has some shortcomings, e.g. it only supports one fixed Print Capability Server. This is not satisfactory for a real life scenario and should be improved.

Link Keys are a good feature in cryptographic access control because they allow the mapping of keys to be independent from the access control model. This is possible because Link Keys do not contain any authorisations, only a reference. The performance of Link Keys is acceptable for small and mid-sized trees of key rings, but will be slow in large trees with more than 100 key rings. The current implementation does not support circular trees, which is likely to cause problems in a real life scenario.

7.5 Summary

This chapter evaluated the project and described performance issues, security aspects and future work. The implementation is, for most commands, fast enough for normal use, and users will not experience noticeably decreased performance. The only problem is with the creation of new keys, which currently is too slow. Overall, the implementation is a functional proof-of-concept for

cryptographic access control in a file system.

The design has vulnerabilities to some attacks, like Replay Attacks, which should be handled before the design can be used in a real application. Furthermore, the implementation should be extensively tested. To achieve optimal performance, it would be necessary to migrate the implementation to a micro kernel to avoid redundant access control.

Chapter 8

Conclusion

Today's access control mechanisms are not suitable for large scale distributed file systems and operating systems due to the need for a centralised user administration. To find an authority that all parties trust can be a difficult task and will often lead to decreased performance because principals needs to verify their identity.

The proposed design is a completely distributed cryptographic access control mechanism which, together with key management, can be used to model access control schemes. The design suggests a replacement of existing access control matrices with a cryptographic key scheme. Although, cryptography is more computationally demanding, the extra time and effort is well spent because existing access control mechanisms do not guarantee confidentiality, and this is often a requirement in open networks like the Internet and wireless communication. The proposed key management further allows principals to grant other principals capabilities to access files and services using key rings.

The design has been implemented in `C` and the performance evaluated. In order to complete the proof-of-concept, the implementation of a Key Manager is not enough in itself. It was also necessary to implement four different types of servers and extend the existing command line shell to handle the new functions provided by the Key Manager. All parts of the implementation use a cryptographic library which is based on existing implementations of cryptographic algorithms. The

library can be replaced with newer or faster algorithms as the present algorithms are deprecated. The project has been tested and it demonstrates the possibility to replace existing centralised schemes with a distributed cryptographic access control scheme.

Cryptographic access control has advantages and disadvantages compared to traditional access control. The performance is slightly slower, but is not noticeable under normal usage, except during key generation. Considering that the design is completely distributed and applicable for untrusted networks, the performance is indeed acceptable. In addition, file servers save expensive computation time because they do not need to encrypt or decrypt files. This scheme is, therefore, especially applicable for distributed systems, where server side computations are limited. The implementation uses 33%-50% more disk space than traditional file storages, but this can be avoided using a different communication protocol or simple decoding on the server side. The design distributes the access control from a central administration to the users. This can lead to security problems if the users are not aware of their responsibility to restrict authorisation to confidential information. The implemented servers are vulnerable to known attacks such as Replay Attacks and Denial of Service Attacks, but this can, in many cases, be avoided using logging. All in all, cryptographic access control in operating systems like the CryptOS project is definitely possible, and hopefully this project will provide a foundation for further work in this area.

Appendix A

Appendix

A.1 Key Manager API

The Key Manager has a total of 16 functions that can be used by other applications. They all return integers depending on whether the call was successful or not. The applications built on top of the Key Manager, does not have to handle the keys.

The functions are:

int import(char * filename)
Imports a key with the specified file name.

int export(int i, char * filename)
Exports key *i* to the specified file name.

int RPCrequest(int i, char * reqText)
Sends an encrypted request to server with key *i*. The server has to be a Service Server or a Print Server.

int create(char * filename, char* ip)
Creates a file on the specified server.

int createLocal(char * filename, char *path)
Creates a file on the specified drive. This file will be encrypted and added to current key ring. Because the file is placed on a local drive, not all users will be able to download the file. The file will be under the operating system's access control and users will therefore need authorisation to access the drive. The file is saved using normal file operations and the integrity will, therefore, not be verified by any server.

int createKeyRing(char * name)
Creates a new key ring, with the specified name, in the current key ring.

int updateKeyRing()
Updates a key ring. This function is called by the other functions when necessary, so it should not be used in a normal implementation.

int update(int i, char * filename)
Updates the file with index *i* with the specified file.

int print(int i, char * filename)
Uploads the specified file to the Print Server with index *i*.

int printcap(int i, char * servername)
Encrypts and sends the key (capability) with index *i* to the specified Capability Server.

int download(int i)
Downloads, decrypts and verifies the file with index *i*.

int changekey(int i)
Replaces the key with index *i*.

int removeK(int i)
Removes the key with index *i* from the current key ring.

int open(int i)
Opens the key ring with index *i*.

int back()
Opens the previous key ring (from the key ring stack).

int view()
Display the current key ring.

If a new application is developed, it is recommended to use the existing `Makefile` to include the project files properly.

A.2 Shell User Manual

The Shell can be initiated by typing `./../Shell/Shell keyring` from the `myFiles` directory. It can, of course, be initiated from other directories as well, but the chosen directory will be the default directory for downloaded files and the files to be uploaded. The `keyring` parameter should be the name (and path) of the local private key ring. The Shell can handle commands to the Key Manager and normal command line commands. The normal command line commands should be prefixed with "' ", i.e. if a user wants to display the contents of the `myFiles` directory, he would write:

```
cmd> ' ls
```

The commands to the Key Manager can just be typed. The Shell supports the following commands:

Command	Description
view	Views the current key ring.
view *keyno*	Views the CAC key with index *keyno*.
open *keyno*	Open key ring with index *keyno*.
back	Goes back to previous key ring.
create *filename*	Creates the local file in the current keyring on the same server.
create *op servername filename*	Creates the local file in the current keyring on another server or locally (*op*=l for local, *op*=r for remote).
createkr *keyringname*	Creates an empty key ring with the specified name.
print *keyno filename*	Prints the local file with name *filename* on the printer with key number *keyno*.
printcap *keyno*	Sends key *keyno* to a predefined capability printer.
update *keyno filename*	Updates *keyno* with *filename*.
download *keyno*	Downloads the file with key number *keyno*.
changekey *keyno*	Replaces the existing key.
import *keyname*	Imports the local key with name *keyname* to the current key ring.
export *keyno keyname*	Exports *keyno* to a local file with name *keyname*.
export *keyno op keyname*	Exports the key as a link (*op*=l) or read(*op*=r) only key.
req *keyno message*	Uses key *keyno* to send a request/message.
remove *keyno*	Removes key with index *keyno*.
quit	Quit the cryptographic shell.
help	Displays help.

In the following section, all the different commands will be illustrated and tested.

A.3 Functional Test

This section contains a functional test of the different commands provided by the Shell. The functional test is a documentation of the test described in section 7.1.

Each command is tested in the Shell, and the command line and results are displayed in the text.

First the Shell program is initiated from the `myFiles` directory by specifying the location of the Shell and the initial private key ring (in this case the private key ring is called `new.keyring`):

```
[simon@localhost myFiles]$ ./../Shell/Shell new.keyring

****************************************
*                                      *
*  Welcome to the Cryptographic Shell  *
*                                      *
****************************************
cmd>
```

The contents of the private key ring can then be displayed using `view` command:

```
cmd> view
           new.keyring
        --------------------------------
        1. robin.txt                    (File (RW))
        2. personal                     (Key Ring (RW))
        3. SimpleService                (Web Service (RW))
        4. Printer                      (Printer)
```

Each key can be viewed by specifying it index in the key ring:

```
cmd> view 1
Key 1 found!
Location:       805DA2C
sid:            682501569
type:           2
filename:       robin.txt
path:
ip:             localhost
symkey:         806924216
pubkey
prikey
prev:           805D5D8
next:           805DE80
```

The public and private key is not printed on the screen, because they can contain special control characters.

The key ring contains a key to another key ring(`personal`), which can be opened:

```
cmd> open 2
File verified!
cmd> view
           personal
        --------------------------------
```

The `personal` key ring is currenly empty.

From the Shell it is possible to make standard Unix commands using the escape character. Below the contents of the `myFiles` directory is displayed:

ext

```
cmd> ' ls
ls
cars.xls  default.keyring  new.keyring  old files  robin.txt
roskilde.txt  start
```

A file form the directory can be created in the `personal` key ring:

```
cmd> create cars.xls
Symmetric Key created : 1612750731
Generating keys...(this may take a while)
Public key and private key generated!
File created on server(return code: 0)
File updated on server(return code: 0)
```

Because the file is added to the key ring it is necessary to generate the keys, encrypt the file, create it on the server and update the key ring on the server. The key generation is the slowest command under normal uses of the application. It takes about 10 seconds. All other commands takes less than a second (Link keys can takes longer). By looking at the `files` directory on the File Server it possible to find the encrypted file (`1612750731.file`), the hash of the file (`1612750731.hash`) and the public key (`1612750731.pubkey`).

If the `personal` key ring is displayed again, the file has been added:

```
cmd> view
           personal
        --------------------------------
        1. cars.xls                     (File (RW))
```

It is possible to go back to the previous, in this case the private key ring:

```
cmd> back
cmd> view
           new.keyring
        --------------------------------
        1. robin.txt                    (File (RW))
        2. personal                     (Key Ring (RW))
        3. SimpleService                (Web Service (RW))
        4. Printer                      (Printer)
```

Keys can be removed from key rings:

```
cmd> remove 1
cmd> view
           new.keyring
        --------------------------------
        1. personal                     (Key Ring (RW))
        2. SimpleService                (Web Service (RW))
        3. Printer                      (Printer)
```

A file is created in the private key ring, encrypted, uploaded to the default server and removed from the local drive:

```
cmd> create immText.txt
Symmetric Key created : 1535962235
 Generating keys...(this may take a while)
Public key and private key generated!
cmd> view
           new.keyring
        --------------------------------
        1. immText.txt                  (File (RW))
        2. personal                     (Key Ring (RW))
```

```
        3. SimpleService                (Web Service (RW))
        4. Printer                      (Printer)

cmd> ' rm immText.txt
cmd> ' ls
ls
cars.xls      file.temp  immText.txt.crypt  new.keyring
roskilde.txt  sig.temp   start
```

The file can be downloaded to the local drive, and integrity checked (verified) and decrypted:

```
cmd> download 1
id: 1117117721
File verified!
cmd> ' ls
ls
cars.xls  file.temp    immText.txt.crypt  new.keyring
sig.temp  immText.txt  roskilde.txt       start
```

It can then be modified using a text editor (Emacs) and updated on the server again:

```
cmd> ' emacs immText.txt
cmd> update 1 immText.txt
Add Stream result: 0
File updated on server(return code: 0)
```

The test goes back into the `personal` key ring and exports the key to the previously upload file and creates a new key ring `public` in the private key ring:

```
cmd> open 3
File verified!
cmd> export 2 cars.key
size of pub:260, size of pri:708
cmd> back
cmd> createkr public
Symmetric Key created : 1744916419
 Generating keys...(this may take a while)
```

```
Public key and private key generated!
File created on server(return code: 0)
cmd> view
           new.keyring
        --------------------------------
        1. public                       (Key Ring (RW))
        2. immText.txt                  (File (RW))
        3. personal                     (Key Ring (RW))
        4. SimpleService                (Web Service (RW))
        5. Printer                      (Printer)

cmd> open 1
File verified!
cmd> view
           public
        --------------------------------
```

The `public` key ring is empty, but the exported key can be imported:

```
cmd> import cars.key
File updated on server(return code: 0)
cmd> view
           public
        --------------------------------
        1. cars.xls                     (File (RW))
```

A Link Key to the file is now exported:

```
cmd> export 1 l cars.linkkey
size of pub:260, size of pri:708
cmd> back
```

The Link Key is imported in the `personal` key ring and the original key is removed:

```
cmd> view
           new.keyring
        --------------------------------
        1. public                       (Key Ring (RW))
```

```
        2. immText.txt                  (File (RW))
        3. personal                     (Key Ring (RW))
        4. SimpleService                (Web Service (RW))
        5. Printer                      (Printer)

cmd> open 3
File verified!
cmd> view
           personal
        --------------------------------
        1. cars.xls                     (File (RW))

cmd> import cars.linkkey
File updated on server(return code: 0)
cmd> view
           personal
        --------------------------------
        1. cars.xls                     (Link)
        2. cars.xls                     (File (RW))

cmd> remove 2
cmd> view
           personal
        --------------------------------
        1. cars.xls                     (Link)
```

The Link Key can now be used to download the file. The Link Key is located in the `personal` key ring and the key pointed to in the `public` key ring. It is necessary for the application to search through all keys in the private key ring and then all keys in the next level of the key ring tree:

```
cmd> download 1
In has key: public
In has key: immText.txt
In has key: personal
In has key: SimpleService
In has key: Printer
File verified!
In has key: cars.xls
File verified!
```

The Link Key has now been used to find the original key and download the corresponding file. The file is now located in the `myFiles` directory.

Now a "read only" key is exported from the `public` key ring and imported to the personal key ring:

```
cmd> back
cmd> open 1
File verified!
cmd> view
           public
        --------------------------------
        1. cars.xls                     (File (RW))

cmd> export 1 r cars.readkey
size of pub:260, size of pri:708
cmd> back
cmd> open 3
File verified!
cmd> import cars.readkey
File updated on server(return code: 0)
cmd> view
           personal
        --------------------------------
        1. cars.xls                     (File (R))
        2. cars.xls                     (Link)

cmd> update 1 cars.xls
Error: Could not update file
        Not an update key!
```

If the user tries to update a file with the "read only" key ring, he is rejected.

The private key ring also contains a key to a simple web service, which can contacted with an encrypted request:

```
cmd> back
cmd> view
           new.keyring
        --------------------------------
        1. public                       (Key Ring (RW))
        2. immText.txt                  (File (RW))
```

```
        3. personal                     (Key Ring (RW))
        4. SimpleService                (Web Service (RW))
        5. Printer                      (Printer)

cmd> req 4 hello
Starting decoding with: eFSQuAmThW0ppEciV7pKIsb01tQa4TTk...
Decoded!        Decrypted!
Response from server: Got your message, thanks!
```

The output from on the Web Service Server is:

```
Key ID: 1374217172
Recieved from client: LKi8DY8+HwvfgQdn+94bLX3WMY...
Using key: 1374217172
Starting decoding with: LKi8DY8+HwvfgQdn+94bLX3WMY...
Decoded!        Decrypted!
Decrypted from client: hello
Replying: Got your message, thanks!
Sending: eFSQuAmThW0ppEciV7pKIsb01tQa4TTk...
```

A file from the local drive (the file must be downloaded) can be uploaded to a Print Server and the print process initiated using a Printer Key:

```
cmd> print 5 immText.txt
Hash is: L8SjZoq+7l2OOqp5sbhonut...
Session to server created (Id: 2117032453)
File sent on server(return code: 0)
cmd> req 5 print
Sending: llgshnYFpD/8OGnEWckG+vyryl1...
Recieved: v4cPSEz2yANfJoB9Cz/lFuHrQit7OZGUc...
Starting decoding with: v4cPSEz2yANfJoB9Cz/lFuHrQit7OZGUc...
Decoded!        Decrypted!
Printer reply: Job printed!
```

The simple Printer Server now has now recieved the encrypted file, decrypted it and printed it on the screen.

Instead of printing a downloaded file, a capability (a key to a file) can be send to a Print Capability Server, which then downloads and decrypts the file. The print process can then be initiated using the `print` command.

```
cmd> printcap 2
size of pub:260, size of pri:708
Connecting to: localhost
Session to server created (Id: 1436110418)
File sent on server(return code: 0)
cmd> req 5 print
Sending: x2NbTptEoxHAHd8GVx08juAoYM8R9cx...
Connection to server created on 'localhost'
Recieved: v4cPSEz2yANfJoB9Cz/lFuHrQit7OZGUcohc...
Starting decoding with: v4cPSEz2yANfJoB9Cz/lFuHrQit7OZGUc...
Decoded!        Decrypted!
Printer reply: Job printed!
```

The file has now been printed to the screen by the Print Capability Server. This demonstrates that the server is able to use the Key Manager to retrieve a file from a File Server.

```
cmd> quit
[simon@localhost myFiles]$
```

The Shell is terminated. Functional test complete.

Bibliography

[BL76] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. *The Mitre Corporation*, 1976.

[Cry06] Website: CryptOS. *Christian D. Jensen: CryptOS.* http://www2.imm.dtu.dk/ cdj/CryptOS/index.html, 16/2-06.

[DAOT05] A. Shamir D. A. Osvik and E. Tromer. Cache attacks and countermeasures: the case of aes. 2005.

[DH96] F Saunier D Hagimont, JMXR de Pina. Hidden software capabilities. *Distributed Computing Systems*, 1996.

[ea00] Furguson et al. "improved cryptanalysis of rijndael". 2000.

[Gro06] System Architecture Group. *The L4Ka Project.* http://www.l4ka.org/, 16/02-06.

[Har01] Anthony Harrington. Cryptographic access control for a network file system. Master's thesis, Trinity College Dublin, 2001.

[HJ03] Anthony Harrington and Christian Jensen. Cryptographic access control in a distributed file system. In *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 158–165, New York, NY, USA, 2003. ACM Press.

[HK03] S. Hjalvig and J. Kampfeldt. Kryptografisk adgangskontrol i peer-to-peer netværk. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2003. Vejleder: Christian D. Jensen.

[Jen00] Christian D. Jensen. Cryptocache: A secure sharable file cache for roaming users. *Proceedings of the Ninth ACM SIGOPS European Workshop*, 2000.

[KR] Lars R. Knudsen and Vincent Rijmen. *The Block Cipher Lounge - AES*. http://www2.mat.dtu.dk/people/Lars.R.Knudsen/aes.html. Website maintained by L.R.Knudsen and V.Rijmen about attacks on AES.

[Mau96] Ueli Maurer. Modelling a public-key infrastructure. *ESORICS*, 1996.

[McD01] Charlie McDowell. *On to C*. Addison Wesley, 2001.

[Mic94] SUN Microsystems. "sun rpc". *Distributed Systems, Edition 2*, 1994.

[oST01] National Institute of Standards and Technology. *Specification for the Advanced Encryption Standard AES*. FIPS, 197, 2001.

[Pfl03] C. P. Pfleeger & S. L. Pfleeger. *Security In Computing*. Prentice Hall, 3rd edition, 2003.

[RS] Website: RSA-Security. *RSA-200 is factored*. http://www.rsasecurity.com/rsalabs/node.asp?id=2879.

[Sch] Website: Bruce Schneier. *New Cryptanalytic Results Against SHA-1*. http://www.schneier.com/blog/archives/2005/08/new_cryptanalyt.html.

[Sha] Declan Patrick Shanahan. Cryptosfs: Fast cryptographic secure nfs.

[Sla] Website: Slashdot. *Factors Found in 200-Digit RSA Challange*. http://it.slashdot.org/article.pl?sid=05/05/10/1955226.

[ST03] A. Shamir and E. Tromer. On the cost of factoring rsa-1024. *RSA CryptoBytes*, 6(2):10–19, 2003.

[Sti02] Douglas R. Stinson. *Cryptography: Theory and Practice*. CRC Press, 2nd edition, 2002.

[Tan99] Andrew S. Tanenbaum. *Structered Computer Organization*. Prentice Hall, 4th edition, 1999.

[THCR99] C. E. Leiserson T. H. Cormen and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill Company, 33 edition, 1999.

[Wik06] Wikipedia. *Advanced Encryption Standard*. http://en.wikipedia.org/wiki/Aes, 11/01-06.

[XWY05] Yiqun Lisa Yin Xiaoyun Wang and Hongbo Yu. Finding collisions in the full sha-1. -1, 2005. Technical paper to appear.

[ZH99] Lidong Zhou and Zygmunt J. Haas. Securing ad hoc networks. *IEEE Network, special issue on network security*, 1999.