uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Hong Guo

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Mobile Certifying Cryptographic File System

TITRE DE LA THÈSE / TITLE OF THESIS

T. Yeap

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Yongyi Mao

Qijun Zhang

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

MOBILE CERTIFYING CRYPTOGRAPHIC FILE SYSTEM

Hong Guo

Thesis submitted to the
Faculty of Graduate and Postdoctral Studies
In partial fulfillment of the requirements
For the degree of Master of Electrical Engineering

Electrical Engineering
School of Information Technology and Engineering
University of Ottawa

Your file Votre référence
ISBN: 978-0-494-18422-6
Our file Notre référence
ISBN: 978-0-494-18422-6

ABSTRACT

This thesis describes the design and implementation of a Mobile Certifying Cryptographic File System (MCCFS), which proposes a solution that allows the mobile user access to remote resources from different locations using either the same device or nearby devices. The MCCFS is aimed to support the host mobility and user mobility in a secure environment.

We separated the key management from the notion of administrative realm based on two factors authentication and certificate-based access control lists. Mobile users can certify themselves in remote domains freely. MCCFS provided transparent encryption support for accessing files not only in local machine but also across Internet on HTTP or FTP servers. Files are encrypted when they are transferred over the network and when saved on untrusted machines.

The MCCFS affords mobile users more flexibility in accessing and sharing their files and does not require the assistance of the local or remote realm's system administrator.

Acknowledgement

I would like to express my gratitude to Prof.Tet Hin Yeap for the immeasurable amount of support and guidance he gave me. I would like to thank the other group members for their cooperation and expert advice. Finally, I am deeply grateful to my family and friends for their support during this process.

Contents

ABSTRACT ii

Acknowledgement iii

1 Introduction 1

2 State of the Arts Survey 5

2.1 Cryptography Concepts 5

2.1.1 Random Numbers Generating 5

2.1.2 Symmetric Encryption 9

2.1.3 Public Key Cryptography 12

2.1.4 Hash Algorithms 15

2.2 Cryptographic File System 17

2.2.1 Local Cryptographic File Systems 17

2.2.2 Network Cryptographic File Systems 19

2.3 Access Control 21

2.3.1 Access Matrix 22

2.3.2 Current ACL Mechanism and Limitations 26

3 Prototype Design 28

3.1 Design Goals 28

3.1.1 Goals 28

3.1.2 Problems to Solve 31

3.2 The Mobile Certifying Cryptographic File System Architecture 32

3.2.1 Overview 32
3.3 Core Security Management 34
3.3.1 Attacks 34
3.3.2 Authorization 35
3.3.3 Authentication 35
3.3.4 Securing data on disk 37
3.4 Access Control 39
3.4.1 Two-Factor-Based Access Control 40
3.4.2 Access Control List 42
3.5 Network File Mapping 43

4 MCCFS Implementation **45**
4.1 MCCFS File Format Specification 46
4.1.1 File Header Format 46
4.2 Cryptographic Implementation 49
4.2.1 Random Number Generator (RNG) 49
4.2.2 Whitening 52
4.2.3 Header Encryption Deriving Function 53
4.3 MCCFS Drive Service 55
4.3.1 File Operations on SVD 57
4.4 Graphic User Interface 58
4.4.1 Create Program 59
4.4.2 Mount Program 60
4.4.3 Recover File 62

5 Performance **64**
5.1 Attack Prevention Capabilities 64
5.2 Test Platform 66
5.3 Local Performance 68
5.4 Network Performance 70
5.4.1 Peer-peer Mode 70
5.4.2 Client-Server Mode 72

6 Conclusions and Future Work **75**
6.1 Conclusion 75
6.1.1 Flexibility 75
6.1.2 Modularity 76
6.1.3 Performance 76
6.1.4 Robustness 77
6.2 Future Work 77

A User Manual **79**
A.1 Setting Up MCCFS Software 79
A.2 Initial USB Token 79
A.3 Mount 80
A.4 Dismount 80
A.5 Change USB Token's Password 81
A.6 Recover Files 82
A.6.1 Client 82
A.6.2 Administrator 83

Bibliography **84**

List of Tables

4.1 Encryption operation mode 50

5.1 Attack Classification 64
5.2 Attack Prevention Capabilities 65
5.3 Disk Bandwidths 74

List of Figures

3.1 MCCFS Design 33
3.2 MCCFS Access Control: Create, read, write 41
3.3 Network Drive Mapping 44

4.1 MCCFS Architecture 45
4.2 MCCFS File Format 46
4.3 MCCFS File Header Format 46
4.4 IV Generation 48
4.5 Random Number Generate Pool Mixing Function 51
4.6 Whitening Generation (Cipher 128) 52
4.7 Secure Virtual Drive Service 56
4.8 Create File 59
4.9 Mount File 61
4.10 Recover File 63

5.1 Local Read and Write Performance 69
5.2 Peer-peer bandwidth by request size: (a) read, (b) write 71
5.3 Peer-peer bandwidth by request size: (a) read, (b) write 73

Chapter 1

Introduction

With the increasing number of computer and Internet applications, the theft of proprietary information consistently causes great financial loss. The security issue has become very important. By now, much of the focus of recent storage security work has been on protecting communication between clients and servers in an untrusted networked world. Cryptographic file systems can protect sensitive or private data from unauthorized disclosure and modification. If data were stored on the server in encrypted form, then it would not be necessary to encrypt it for each transfer on the network.

Storing data in encrypted form was originally proposed in Blaze's Cryptographic File System (CFS) [1] to protect data from untrusted servers. If data is stored on the server in encrypted form, it is protected from leaking by the server, and there is no need to encrypt data again when it is sent on the network. Encryption is done by the original creator of the file and updated by subsequent writers, but the server performs no encryption or decryption.

This thesis proposes that the Mobile Certifying Cryptographic File System (MCCFS) provide mobile certifying to different client machines and efficient accessing to encrypted data saved on remote machine. The mobile certifying is based on a USB token and a user password.

Mobile certifying is to satisfy the storage security requirements of mobile users. This becomes particularly evident when users perform their daily routine with the help of access the application or information on different devices such as the desktop, notebooks, or servers. They require a fast and reliable connection to their home system or the ability to load data to their temporal locations. As the files are transformed and shared over the network, they must defend the usual attacks on it, such as data change or data leak. Remote login facilities often create new opportunities for attackers to access the data illegally. To reduce these risks, cryptographic techniques make it possible to limit data access while still taking advantage of untrustworthy networks and services. This does not only include the protection of the communication itself but also the protection of possibly confidential information and data stored locally on the different locations.

Data access control is to protect against mistrusted access (reading and writing). Nobody would like his or her confidential information to be exposed to other people. For governments and enterprises, there is also much data that has to be kept secure and cannot be accessed by any unauthorized person. MCCFS provides transparent access to encrypted files and protects the sensitive data even when the media is stolen. It works on top of a conventional file system.

Currently, password-based encryption is still in widespread use in such environments. In order to resist the tamper attack, many people choose asymmetric encryption algorithm because it is harder to be broken than symmetric encryption algorithm. However, in the case of encryption-on-disk, asymmetric algorithms are too slow to satisfy the requirement of encrypting a large amount of data.

In addition to mobile certifying and access control, there is a third factor to consider when building any secure system: the level of inconvenience users are willing to tolerate. If users must type in a separate password for every document they open or individually choose access rights for every file they create, they simply circumvent it entirely. If users perceive encryption to slow down their work, they just turn it off.

MCCFS provides a deeper integration between the encryption service and the file system, which results in complete transparency of use to the user application. As a mobile certifying system, MCCFS offers the user the option of saving the encrypted files on any machine. Users can reach any MCCFS file from any MCCFS client in the world. The MCCFS protocol then cryptographically guarantees the integrity of all file data and the secrecy of any data without losing universe-readable ability. These guarantees hold over mistrusted networks and across administrative realms. We expect cryptographic file systems to become a commodity component of future operation systems.

To achieve high-speed encryption, we combine the tamper-resistant USB hardware with the asymmetric cryptographic keys and algorithms, such as PKI, using a secure authentication mechanism for the holder verification. Our work improves the traditional encryption file system by resisting active and passive network attacks while avoiding any form of centralized control. MCCFS establishes and maintains

an on-the-fly encryption file system. On-the-fly encryption means that data are automatically encrypted or that decrypted files can be read without using the correct password or encryption key. Until decrypted, a MCCFS file appears to be nothing more than a series of random numbers. The entire file system is encrypted, such as files names, directory names, contents of every file, and free space. MCCFS never writes decrypted data to any storage devices.

This thesis consists of six chapters. Chapter 1 gives a general introduction on the Mobile Certifying Cryptographic File System. Chapter 2 presents basic relative concepts and analyzes several existing cryptographic file systems. Chapter 3 outlines our design to satisfy the constraints of Chapter 2. Chapter 4 describes the implementation details. Chapter 5 summarizes our experiment results. Finally, Chapter 6 gives the conclusion of our current work and suggestions for future directions.

Chapter 2

State of the Arts Survey

This chapter describes a range of available techniques of encryption data and corresponding application systems.

2.1 Cryptography Concepts

2.1.1 Random Numbers Generating

Security-critical applications often require well-chosen random numbers for purposes ranging from cryptographic key generation to shuffling a virtual deck of cards.

There are essentially three classes of solutions [2]:

Insecure random number generators More properly, these are non-cryptographic pseudo-random number generators. It should generally be assumed that an attacker could predict the output of such a generator.

Cryptographic pseudo-random number generators (PRNGs) These take a single secure seed and produce as many unguessable random numbers from that

seed as necessary. Such a solution should be secure for most uses as long as a few reasonable conditions are met (the most important being that they are securely seeded).

Entropy harvesters These are sometimes "true" random number generators, although they really just try to gather entropy from other sources and present it directly. They are expected to be secure under most circumstances, but are generally incredibly slow at producing data.

MCCFS chose a combination of the second and third classes, which are secure and fast enough for general purpose use. It generates file passphrases, initialization vectors, salts and whitening seeds based on the Random Numbers Generating Function.

Passphrase

A passphrase is a collection of 'words' used for access control, typically used to gain access to a computer system. Passphrases are also used to control both access to, and operation of, special security programs such as cryptographic systems. The origin of the term is analogous with "password". The modern concept of passphrases is believed to have been invented by Sigmund N. Porter in 1982 [3].

Passphrases differ from passwords. A password is usually short – six to ten characters. Such passwords may be adequate for logging onto computer systems, but they are certainly not safe for use with quality security systems. Passphrases are a better choice. First, they usually are (and always should be) much longer – 20 to 30 characters or more is typical, making some kinds of brute force attacks entirely impractical. Second, if well chosen, they will not be found in any 'phrase or quote dictionary', so such dictionary attacks will be impossible. Third, they can be so structured as to be

more easily remembered than passwords without being written down, reducing that risk as well. They can thus be considerably more 'secure'.

Initialization Vector

In cryptography, an initialization vector (IV) is a block of bits that is combined with the first block of data in any of several modes of a block cipher. In some cryptosystems, IV is random and is sent with the ciphertext; in others, such as a disk encryption subsystem, it is based on some information, such as the file's created time, that does not have to be put in the ciphertext.

Initialization vectors are very important when different files are encrypted using the same key. In general, XORing two cipher-texts created using a stream cipher initialized with the same key will yield the same result as XORing two different plain-texts. Recovering both is then simple. Thus, if the same key needs to be reused, a few random bytes, which can be written into the start of the encrypted file, are independent of the key, assuring a different initialization of the cipher-stream for every encryption process.

The situation is different for block ciphers. In Electronic Code Book (ECB) mode same plain-text will encrypt to same cipher-text (for the same key). This reveals patterns in the code. In Cipher Block Chaining (CBC) mode, each block is XORed with the result of the encryption of the previous block. This hides patterns. However, two similar plain-texts will have (for the same encryption key) the same cipher-text up to the block containing the first difference. This problem can be circumnavigated by adding a random IV block to the plain-text. This will make each cipher-text unique, even when similar plain-text is encrypted with the same key in CBC mode.

Salt

In data encryption, salt is an initialization vector of a block cipher, specifically often, salt is an initialization vector used to obscure a pass phrase.

When the list of encrypted passphrase is simply stored in a file, users who shares the same passphrase will also share the same encrypted password. In this simple scheme, if a user is able to find another encrypted passphrase that is a copy of his, he will be able to deduce that he and that other user share the same password.

To get around this problem, we use in our encryption (salt + pass phrase),hence the resultant encrypted pass phrase will be different even if the pass phrase is the same.

In cryptography, salt consists of random bits, typically 12 or more, used as one of the inputs to a key derivation function. The other input is usually a password or pass phrase. The output of the key derivation function is often stored as the encrypted version of the password. It can also be used as a key for use in a cipher or other cryptographic algorithm. A salt value is typically used in a hash function.

The salt value may or may not be protected as a secret. In either case, the additional salt data makes it more difficult to conduct a dictionary attack using pre-encryption of dictionary entries, as each bit of salt used doubles the amount of storage and computation required.

Whitening

Whitening is the name given to the technique of XORing some key material with the input to a block algorithm and XORing some other key material with the output.

This was first done in the DESX variant developed by RSA Data Security Inc.

$$C = K_3 \otimes E_{K2}(M \otimes K_1)$$
$$M = k_1 \otimes D_{K2}(C \otimes K_3)$$

Data whitening is using cryptography to remove statistical correlations between bits, effectively spreading entropy evenly throughout data. This is commonly done with a cryptographic hash function, but can be done with a block cipher.

2.1.2 Symmetric Encryption

There are two classes of symmetric primitives, both of utmost importance. First are symmetric encryption algorithms, which provide for data secrecy. Second are message authentication codes (MACs), which can ensure that, if someone tampers with data while in transit, the tampering will be detected.

For the symmetric encryption, we need to keep an internal representation of a symmetric key. We may need to save this key to disk, pass it over a network, or use it in some other way. We may need some mechanism for securely transporting the key to anyone who needs it. We need the key to be as strong as the cipher we are using, and we want the key to be absolutely independent of any other data in our system.

Cipher

For cryptographic file systems, there are two general types of ciphers:

Block ciphers These work by encrypting a fixed-size chunk of data (a block). Data that is not aligned to the size of the block needs to be padded somehow. The

same input always produces the same output.

Stream ciphers These work by generating a stream of pseudo-random data, then using XOR to combine the stream with the plain text.

Compared with stream cipher, block ciphers are efficient and versatile. Stream ciphers generally are used as designed. This class of ciphers can be made to act as block ciphers, but that generally destroys their best property (their speed), so they are typically not used that way.

There are many different ways of using block ciphers; these are called block cipher modes. Selecting a mode and using it properly is important to security. Many block cipher modes are designed to produce a result that acts just like a stream cipher. Each block cipher mode has its advantages and drawbacks. Such as DES variants, Blowfish, and Rijndael, they are often used for file encryption and are believed to be secure. There are many other block ciphers available, including CAST, GOST, IDEA, MARS, Serpent, RC5, RC6, and TwoFish. Most of them have similar characteristics with varying block and key sizes.

DES

DES is a block cipher designed by IBM with assistance from the NSA in the 1970s [4]. DES was the first encryption algorithm that was published as a standard by NIST with enough details to be implemented in software. DES uses a 56-bit key, has a 64-bit block size, and can be implemented efficiently in hardware. DES is no longer considered to be secure. There are several more secure variants of DES, most commonly 3DES (Triple-DES) [5]. 3DES uses three separate DES encryptions with three different keys, increasing the total key length to 168 bits. 3DES is considered

secure for government communications. DESX is a variant designed by RSA Data Security that uses a second 64-bit key for whitening (obscuring) the data before the first round and after the last round of DES proper, thereby reducing its vulnerability to brute-force attacks, as well as differential and linear cryptanalysis [4].

AES

AES (Advanced Encryption Standard) [6] is a great general-purpose block cipher. It is among the fastest block ciphers, is extremely well studied, and is believed to provide a high level of security. It can also use key lengths up to 256 bits.

AES has recently replaced Triple-DES as the block cipher of choice, partially because of its status as a U.S. government standard and partially because of its widespread endorsement by leading cryptographers. However, Triple-DES is still considered a very secure alternative to AES. In fact, in some ways it is a more conservative solution, because it has been studied for many more years than has AES and because AES is based on a relatively new breed of block cipher that is far less understood than the traditional underpinnings upon which Triple-DES is based.

Nonetheless, AES is widely believed to be able to resist any practical attack currently known that could be launched against any block ciphers. Today, many cryptographers would feel just as safe using AES as they would using Triple-DES. In addition, AES always uses longer effective keys and is capable of key sizes up to 256 bits, which should offer vastly more security than Triple-DES with its effective 112-bit keys.

2.1.3 Public Key Cryptography

The need for a separate, out-of-band secret key exchange step can lead to tremendous difficulties when entities are unknown to each other. Public key cryptography offers a number of important advantages over traditional, or symmetric, cryptography.

Traditional cryptography is done with a single shared key. It needs a more secure out-of-band medium for transport, such as telephone or postal mail. Such a solution is rarely practical, however, considering that we might want to do business securely with an online merchant we have never previously encountered. Public key cryptography can help solve the key agreement problem, although doing so is not as easy as one might hope.

Encryption

With some public-key algorithms, encrypting data with the public key is possible; then the resulting ciphertext can be decrypted only with the corresponding private key. Generally, however, the computations involved in public-key cryptography are too slow, thus impractical for many environments. Typically what is done instead involves a two-step process, as follows:

1. The data is encrypted using a randomly generated symmetric key.
2. The symmetric key is then encrypted using the public key of the intended recipient of the data.

When the recipient receives the encrypted data, a similar two-step process takes place:

1. The recipient decrypts the symmetric key, using his or her private key.
2. The symmetric key is then used to decrypt the actual data.

Even when the total amount of data to be encrypted is very small, the two-step process presented here is typically used rather than direct data encryption/decryption using the public/private-key pair. This serves to keep processing clear and simple so that there is never any confusion about whether the output of a private-key decryption operation is data or a symmetric key.

RSA

The algorithm proposed by Ron Rivest, Adi Shamir, and Len Adleman in 1978 [7], known as RSA, is one of the earliest and most versatile of the public-key algorithms. It is suitable for encryption/decryption, for signing/verification (and, therefore, for data integrity), and for key establishment (specifically key transfer). It can be used as the basis for a secure pseudorandom number generator as well as for the security in some electronic games. Its security is based on the difficulty of factoring very large integers. The current state of factoring research suggests that RSA keys should be at least 1,024 bits long to provide adequate security for the next several years or more.

DSA

The Digital Signature Algorithm (DSA) is a Federal Information Processing Standard (FIPS) publication of NIST of the U.S. Department of Commerce [8]. It is a variant of the ElGamal signature mechanism [9]. The DSA was designed exclusively for signing/verification and therefore also for data integrity. Other algorithms in the ElGamal family can be used for encryption/decryption and therefore key transfer, if what is being encrypted and decrypted is a symmetric key. The security of these algorithms is based on the difficulty of computing logarithms in a finite field. The

current state of research with respect to discrete logarithms suggests that DSA keys should be at least 1,024 bits long to provide adequate security for the next several years or more.

DH

The algorithm that Diffie and Hellman proposed, known as DH, is a wonderful example of elegance and simplicity [10]. The earliest public-key algorithm, it is exclusively a key establishment, specifically key agreement, protocol. Each of the two entities uses its own private key and the other entity's public key to create a symmetric key that no third entity can compute. The algorithm derives its security from the difficulty of computing logarithms in a finite field. As with DSA, the current state of research with respect to discrete logarithms suggests that DH keys should be at least 1,024 bits long to provide adequate security for the next several years or more.

ECDSA and ECDH

The DSA and DH algorithms can also be computed over the group of points defined by the solution to an equation for an elliptic curve over a finite field [11, 12]. The resulting elliptic curve DSA (ECDSA) and elliptic curve DH (ECDH) algorithms have identical uses to their finite field counterparts earlier, but the security now rests on the difficulty of computing logarithms over the group of EC points. This different foundation leads to more complicated implementation and processing but has the benefit of significantly smaller key sizes for a similar level of security. The current state of research with respect to discrete logarithms over EC points suggests that ECDH and ECDSA keys should be at least 192 bits long to provide adequate security

for the next several years or more.

Note that it is possible to do elliptic curve RSA as well; however, due to the different security basis (integer factorization as opposed to discrete logarithms), the key sizes are not significantly smaller than "ordinary" RSA. Thus, the added complication has no perceived benefit, and ECRSA appears not to be used anywhere.

2.1.4 Hash Algorithms

MCCFS adopted a user-selected hash algorithm in the random number generator function.

SHA-1

The Secure Hash Algorithm SHA-1, a slight revision of the original Secure Hash Algorithm SHA, is described in an NIST FIPS publication [13]. This hash algorithm was designed specifically for use with the DSA but can be used with RSA or other public-key signature algorithms as well. Its design principles are similar to those used in the MD2 [14], MD4 [15], and especially MD5 [16] hash functions proposed by Ron Rivest. Current computational capability suggests that the size of the SHA-1 hash value, 160 bits, provides adequate security for at least the next several years. Hash functions are not public-key algorithms but are included here because digital signature algorithms are always used in conjunction with hash algorithms to provide the services of signing/verification and data integrity. Thus, they are an essential component of the digital signature and integrity security services.

Whirlpool

The Whirlpool hash algorithm was designed by Vincent Rijmen, who is one of the authors of the AES encryption algorithm too, and Paulo S. L. M. Barreto. The size of the output of this algorithm is 512 bits. The first version of Whirlpool was published in November 2000. The second version, now called Whirlpool- T, was selected for the NESSIE (New European Schemes for Signatures, Integrity and Encryption) portfolio of cryptographic primitives (a project organized by the European Union, similar to the AES contest). We used the third (final) version of Whirlpool, which was adopted by the International Organization for Standardization (ISO) and the IEC in the ISO/IEC 10118-3:2004 international standard [17].

RIPEMD-160

RIPEMD-160, published in 1996, is a hash algorithm designed by Hans Dobbertin, Antoon Bosselaers, and Bart Preneel in an open academic community, and it represents a valuable alternative to SHA-1. The size of the output of RIPEMD-160 is 160 bits. RIPEMD-160 is a strengthened version of the RIPEMD hash algorithm, which was developed in the framework of the European Union's project RIPE (RACE Integrity Primitives Evaluation), 1988-1992, and in which collisions were found in 2004. No collisions have been found in RIPEMD-160 so far, and no method is known to do so with effort smaller than that required for brute force on average.

2.2 Cryptographic File System

The cryptographic file system is not a new idea. Systems like CFS [1], SFS[18], and BestCrypt[19] allow the encryption of files on disks. In these systems, files are transparently encrypted/decrypted using a key stored in the memory of process operating on the file. These systems use symmetric cryptography, which means that encryption and decryption use the same key. This means that the separation of read and write access must be enforced by other methods, such as the access control mechanisms of the underlying file system. Furthermore, neither of these systems provides the flexibility of file accessing required by mobile users. In this section, we introduce some previously proposed cryptographic file systems.

2.2.1 Local Cryptographic File Systems

Local Cryptographic File Systems (LCFSs) are embedded with the user client file system. Some LCFSs operate below the file system level, such as SFS, and encrypt one disk block at a time. But it is difficult for users to share the encrypted file.

CFS

CFS is a cryptographic file system that is designed as a secure local file system and implemented as a user-level NFS server [1]. It requires the user to create a directory on the local or remote file system to store encrypted data. The cipher and key are specified when the directory is first created. The CFS daemon is responsible for providing the owner with access to the encrypted data via a special attach command. The daemon, after verifying the user ID and key, creates a directory in the mount point directory that acts as an unencrypted window to the user's encrypted data.

Once attached, the user accesses the attached directory like any other directory. CFS is a carefully designed, portable file system with a wide choice of built-in ciphers. However, it lacks features for encrypted file sharing among users. They can only share a protected file by directly handing out file keys to other users.

EFS

The Encryption File System (EFS) is found in Microsoft Windows, based on the NT kernel (Windows 2000 and XP) [20]. It is an extension to NTFS and utilizes Windows authentication methods as well as Windows ACLs [21, 22] and supports encrypting data similar to CFS. Though EFS is located in the kernel, it is tightly coupled with user-space DLLs to perform encryption and the user-space Local Security Authentication Server for authentication [23]. This prevents EFS from being used for protection files or folders in the root or \winnt directory. Encryption keys are stored on the disk in a lockbox that is encrypted using the user's login password. This means that, when users change their password, the lockbox must be re-encrypted. If an administrator changes the user's password, then all encrypted files become unreadable. Additionally, for compatibility with Windows 2000, EFS uses DESX [4] by default, and the only other available cipher is the 3DES (included in Windows XP or in the Windows 2000 High Encryption pack). If without external secure network solution, it is vulnerable to attacks on the wire.

SFS

Secure File System (SFS) is a MSDOS device driver that encrypts an entire partition [18]. SFS is a block-based encryption file system that operates below the file system

level, encrypting one disk block at a time. This is advantageous because it do not require knowledge of the file system that resides on top of it, and can even be used for swap partitions or applications that require access to raw partitions. Also, it does not reveal information about individual files (e.g., sizes) or directory structure. Once encrypted, the driver presents a decrypted view of the encrypted data. This provides the convenient abstraction of a file system, but SFS relies on MSDOS, which is risky because MSDOS provides none of the protections of a modern operating system.

StegFS

Steganographic File System (StegFS) is a local file system that employs steganography as well as encryption [24]. If adversaries inspect the system, then they only know that there is some hidden data. They do not know the contents or extent of what is hidden. This is achieved via a modified Ext2 kernel driver that keeps a separate block-allocation table per security level. It is not possible to determine how many security levels exist without the key to each security level. When the disk is mounted with an unmodified Ext2 driver, random blocks may be overwritten, so data is replicated randomly throughout the disk to avoid loss of data. Although StegFS achieves plausible deniability of the data's existence, the performance degradation is a factor of 6-196, making it impractical for most applications.

2.2.2 Network Cryptographic File Systems

Network Cryptographic File Systems (NCFSs) have two major advantages:

1. They can cooperate with different file systems, and

2. they are more portable than local file systems.

NCFS's major disadvantage is that performance suffers. Security also suffers because NCFS are vulnerable to the weaknesses of the underlying network protocol.

BestCrypt

BestCrypt is a commercially available loopback device driver supporting many ciphers [19]. BestCrypt supports both Linux and Windows and uses a normal file as a backing store. Such a loopback device driver creates a raw block device with a single file, called a container, as the backing store. This device can then be formatted with any file system or used as swap space. Each container has a single cipher key. The administrator creates, formats, and mounts the container as if it were a regular block device. BestCrypt is ideal for single-user environments but unsuitable for multi-user systems. In a single-user workstation, the user controls the details of creating and using a container. In a multi-user environment, however, the user must give the encryption key to a potentially untrustworthy administrator. Moreover, the ability to share containers among groups of users is limited, as BestCrypt gives different users equal rights to the same container. The MCCFS is similar to BestCrypt and other loop-device encryption systems, but it uses a native disk or partition as the backing-store [25].

AFS

Andrew File System (AFS) [26] is one of the first distributed file systems to specifically addressed security issues. AFS detects mistrusted users and uses Kerberos to authenticate users to servers. At the beginning of a session, users obtain tokens from a Kerberos server, which authorizes them to access the storage servers. AFS servers

verify the tokens and then do appropriate authorization based on group information maintained by a group server. A secure version of RPC is used to protect communication. AFS is vulnerable to leak, modify, and destroy attacks in collusion with any of the servers.

NFS

The Network File System Version 4 (NFSv4) is the newest distributed file system similar to previous versions of NFS [27]. It proposes at least three security mechanisms: one using Kerberos and two using a public key infrastructure. All of these essentially set up a secure communication channel and enable mutual authentication. Interestingly, one of these mechanisms - the low infrastructure public key mechanism - exploits the fact that the client authentication can proceed after establishing a secure channel, to reduce the PKI overhead. In this scheme, only the server needs to have a public/private pair that authenticates the server and sets up a secure channel. NFSv4 also greatly expands the use of ACLs for access control.

2.3 Access Control

The purpose of access control is to limit the actions or operations that a legitimate user of a computer system can perform. Access control constrains what a user can do directly, as well what programs executing on behalf of the users are allowed to do. In this way, access control seeks to prevent activity which could lead to a breach of security.

Access control relies on and co-exists with other security services in a computer system. It is enforced by a reference monitor that mediates every attempted access

by a user to objects in the system. The reference monitor consults an authorization database in order to determine whether the user attempting to do an operation is actually authorized to perform that operation. Authorizations can be administered and maintained by a security administrator. The administrator sets these authorizations on the basis of the security policy of the organization. Users may also be able to modify some portion of the authorization database, for instance, to set permissions for their personal files. Auditing monitors and keeps a record of relevant activity in the system.

Here, access control is totally different from authentication. Correctly establishing the identity of the user is the responsibility of the authentication service. Access control assumes that authentication of the user has been successfully verified prior to enforcement of access control via a reference monitor. The effectiveness of the access control rests on a proper user identification and on the correctness of the authorizations governing the reference monitor.

The security of the network file systems mentioned above, such as NFS and AFS, merely provide a weak scheme for access control, but neglect data integrity and confidentiality.

2.3.1 Access Matrix

In the access control model of security, an access control matrix associates rights for operations on objects with subjects. Activity in the system is initiated by entities know as subjects. Subjects are typically users or programs executing on behalf of users. A user may sign on to the system as a different subject on different occasions,

depending on the privileges the users wishes to exercise in a given session. For example, a user working on two different projects may sign on for purpose of working on one project or the other. We then have two subjects corresponding to this user, depending on the project the user is currently working on.

The access matrix is a conceptual model that specifies the rights that each subject possesses for each object. There is a row in this matrix for each subject and a column for each object. Each cell of the matrix specifies the access authorized for the subject in the row to the object in the column. The task of access control is to ensure that only those operations authorized by the access matrix actually get executed. Traditionally, this is achieved by means of a reference monitor, which is responsible for mediating all attempted operations by subjects on objects. In order to simplify the monitoring operations, we combine the access control matrix with the file in encryption format. Note that the access matrix model clearly separates the problem of authentication from that of authorization.

Access Control List A popular approach to implementing the access matrix is by means of Access Control Lists (ACLs). Each object is associated with an ACL, indicating for each subject in the system the access that the subject is authorized to execute on the object. This approach corresponds to storing the matrix by columns. By looking at an object's ACL, it is easy to determine which modes of access subjects are currently authorized for that object. In other words, ACLs provide for convenient access to an object by replacing the existing ACL with an empty one.

Role-Based Policies Role-based policies regulate the access of users to the information on the basis of the activities the users execute in the system. Role-based

policies require the identification of roles in the system. A role can be defined as a set of actions and responsibilities associated with a particular working activity. Then, instead of specifying all of the accesses each user is allowed to execute, access authorizations on objects are specified for roles. Users are given authorizations to adopt roles.

The user playing a role is allowed to execute all accesses for which the role is authorized. In general, a user can take on different roles on different occasions. Also, the same role can be played by several users, perhaps simultaneously. Some proposals for role-based access control allow a user to exercise multiple roles at the same time. Other proposals limit the user to only one role at a time or recognize that some roles can be jointly exercised while others must be adopted in exclusion to one another.

The role-based approach has several advantages. Some of these are discussed below.

- Authorization management: Role-based policies benefit from a logical independence in specifying user authorizations by breaking this task into two parts, one which assigns users to roles and one that assigns access rights for objects to roles. This greatly simplifies security management. For instance, suppose a user's responsibilities change, say, due to a promotion. The user's current roles can be taken away and new roles assigned, as appropriate for the new responsibilities. If all authorization is directly between users and objects, it becomes necessary to revoke all existing access rights of the user and assign new ones. This is a cumbersome and time-consuming task.

- Hierarchical roles: In many applications, there is a natural hierarchy of roles, based on the familiar principles of generalization and specialization. For example, the roles of hardware and software engineers are specializations of the engineer role. A user assigned to the role of software engineer (or hardware engineer) will also inherit privileges and permissions assigned to the more general role of engineer. The role of supervising engineer similarly inherits privileges and permissions from both software engineer and hardware engineer roles. Hierarchical roles further simplify authorization management.

- Least privileges: Roles allow a user to sign on with the least privileges required for the particular task at hand. Users authorized to powerful roles do not need to exercise them until those privileges are actually needed. This minimizes the danger of damage due to inadvertent errors or to intruders masquerading as legitimate users.

- Separation of duties: Separation of duties refers to the principle that no user should be given enough privileges to misuse the system on their own. For example, the person authorizing a paycheque should not also be the one who can prepare the paycheque. Separation of duties can be enforced either statically or dynamically. An example of dynamic separation of duty is the two-person rule. The first user to execute a two-person operation can be any authorized user, whereas the second user can be any authorized user different from the first.

- Object classes: Role-based policies provide a classification of users according to the activities they execute. Analogously, a classification should be

provided for objects. For example, generally a clerk needs to have access to bank accounts, and a secretary will have access to letters and memos. Objects could be classified according to their type or to their application area. Access authorizations of roles should then be on the basis of object classes, not specific objects. For example, a secretary role should be given the authorization to read and write the entire class of letters, instead of giving it explicit authorization. This would be much easier and better controlled. Moreover, the accesses authorized on each object are automatically determined according to the type of the object without needing to specifying authorizations upon each object creation.

- Administrative policies: They determine who is authorized to modify the allowed accesses. This is one of the most important and least understood aspects of access controls. In mandatory access control, the allowed accesses are determined entirely on the basis of the security classification of subjects and objects. Security levels are assigned to users by the security administrator. Security levels of objects are determined by the system on the basis of the levels of the users creating them.

2.3.2 Current ACL Mechanism and Limitations

Existing systems have several shortcomings when used for information-sharing tasks. First, traditional user authentication implies that the user is known to the system before file requests can be processed. Second, file and directory permissions are concepts inherited from multi-user operating systems. Sharing is achieved by either account sharing (which is ill-advised, as it defeats accountability) or through the use

of group access permissions. Such permissions lack flexibility and fine granularity and, perhaps most importantly, extensibility: there is no way of adding new permission mechanisms if existing ones prove inadequate.

Traditionally, fileholders have relied on access control lists (ACLs), stored on the resource server, to present access policy. However, such ACLs typically require a central administrator to maintain all changes, which means both that the administrator must be trusted by all fileholders and that the administrator is potentially a bottleneck to rapid updating of the policy. Also, ACLs usually require the server domain to maintain accounts and other administrative support for both fileholders and users. These problems are all exacerbated when some or all fileholders and users are administratively and geographically remote from the server.

Another problem that arises in network file systems is that there may be multiple principals from different administrative domains who need to have input to the access control policy for a single resource.

Chapter 3

Prototype Design

3.1 Design Goals

3.1.1 Goals

Many existing network file systems encourage the formation of inconveniently large administrative realms, either to maintain security without restricting file sharing or to save users from unreasonable complexity. MCCFS's primary design goals were to balance the often conflicting concerns of security, convenience, and performance. MCCFS's mobile access feature assures users access to any servers or client computers they need from any location. Thus, users can run a single authentication agent at login time and never need to worry about separately authenticating themselves to different servers.

We tried to make our system design satisfying the following requirements:

Lightweight It has been pointed out by many researchers that mobile hardware is, and will always continue to be, resource-poor when compared to their stationary

counterparts [28]. Therefore, any mechanism involving mobile entities needs to ensure that the client-side load is kept to a minimum. This could include simplifying the client module at the expens of a more complex server module, since the server is assumed to be more capable.

Security The mechanism involves controlling access to resources, so it needs to ensure that it is secure. Both the mobile user and the service provider may require guarantees about the token they handle. Users may need assurance that, when acquiring a token, they are doing so from the genuine provider and not someone masquerading as the provider. They may also want to be assured that the token has not been intercepted by an eavesdropper. The service provider needs to be sure that the token is genuine (i.e., not a forgery) and original (i.e., not a duplicate) before allowing access to the service. Security often involves a trade-off with other characteristics (e.g., lightweight). We ensure that data stored using MCCFS remains confidential by using strong encryption.

Convenient Mobile users will come in any shape and form, covering a wide range of performance and capabilities. Additionally, the range of applications and users is likely to be very diverse. Therefore, a mechanism that can flexibly support a wide range of user types, devices, and services is required. If a system is not convenient, then users will not use it or will circumvent its functionality. The inconvenience of current cryptographic systems contributes to their lack of widespread adoption. MCCFS makes encryption transparent to the application: any existing application can make use of strong cryptography with no modifications. We designed MCCFS to be cipher-agnostic so it is not tied to any one cipher.

Simple To be useful to and to be used by a large number of nonspecialist users, the token mechanism must not be complicated.

In designing MCCFS, the most important problems are security and convenience. We were interested in providing a robust security mechanisms with the simplest possible operation to the user.

For the security mechanism, MCCFS must guarantee that secure files should not be readable

- by any user other than the legitimate owner
- by wire tapping the communication lines between the user and the remote file system
- by the super user or the administrator of the file system server (in some special cases)

In MCCFS's security mechanism, the key management and distribution are handled both by the client user and administrator. The advantage of doing this over existing encrypt-on-wire systems is that we can

- protect against data leakage attacks on the physical device, such as by a mistrusted user, a stolen laptop, or a compromised server
- allow users to set arbitrary policies for key distribution (and therefore file sharing)
- enable better server scalability, because most of the computationally intensive cryptographic operations are performed at end systems, rather than in centralized servers

3.1.2 Problems to Solve

Having outlined the desirable characteristics of the mobile certifying and mobile accessing mechanism, we now consider the problems that need to be solved in order to implement authentication based on the USB token.

Duplication The user's authentication is based on the public/private-key cryptography. A public key or private key is typically distributed in the form of a certificate, whereas the private key is always protected separately and distinctly from unauthorized disclosure in transit, use, and storage. So the problem of certificate duplication needs to be considered. Here, two potential scenarios need to be considered. First, there is the dishonest user who legitimately acquires a certificate and then makes copies that are either sold on or used instead of purchasing a new certificate. Second, there is the case of a cracker who intrudes into another's system, acquires a certificate, and takes a copy for him. The first concerns the need to prevent duplication of a valid certificate by unscrupulous users and is a non-trivial problem to solve. The second is related to the security of the application environment and the need to prevent unauthorized entities from being able to access data.

Forgery The aim is to prevent an entity from constructing a certificate such that the local system or remote server accepts it as a valid certificate and hence allows access to the resource. Such as the forgery of the public keys; Tom can easily create new key pairs, and if he can fool Alice into believing that the forged key belongs to Bob, he will be able to access the data Alice sends to Bob using that forged key. So even though public keys can be universally known, some way must be found to prevent their forgery.

Modification The client should not be able to modify the USB token. This is to prevent the client from accessing resources for which they have not been issued authority or for accessing more resources than the authority allows. This can be achieved by using a keyed one-way cryptographic hash function.

3.2 The Mobile Certifying Cryptographic File System Architecture

3.2.1 Overview

Two distinct situations are of particular interest:

- Access within a private enterprise
- Access within a more general environment

The first environment is that of the private enterprise that employs mobile technologies, such as wireless access, in order to facilitate employee mobility. The problem here is that, by introducing mobile access technologies, a significant security problem can exist.

The second situation is a generalization of this that covers mobile service provision for a much larger and more diverse population and range of services. In this situation, the mobile user no longer requires an account per se with a service provider. Instead, the user holds a number of certificates that can be used to access the data saved in different locations. All of the certificates can be saved in one token. Once the user logs into the secure environment correctly, the MCCFS system will choose the

corresponding certificate to access the required data. So there is no difference between the user handling different certificates or handling only one certificate.

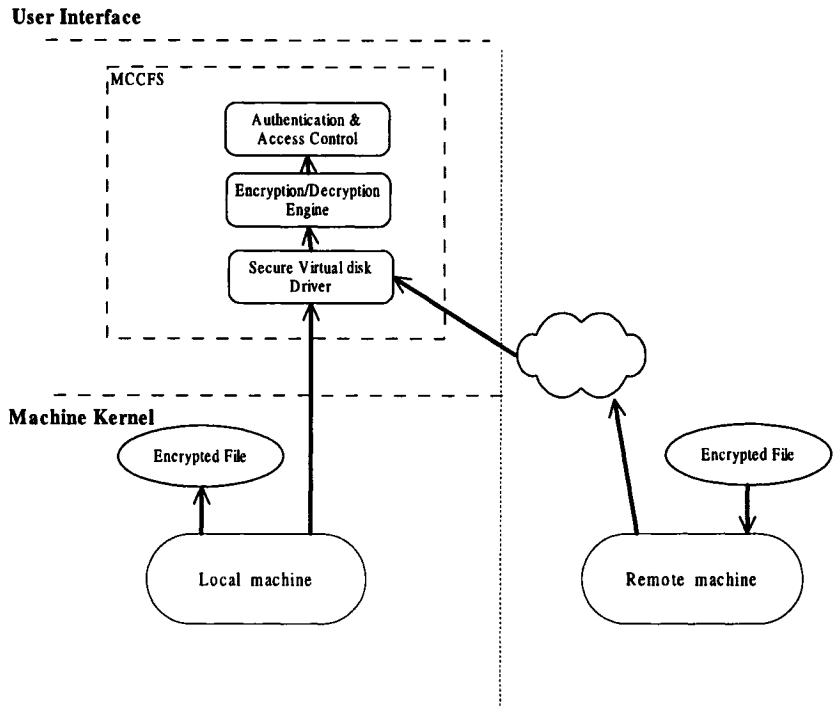


Figure 3.1: MCCFS Design

To maximize the level of security achievable, we keep to the minimum the number of trusted entities. Users need to trust only the kernel of the machine on which their application programs accessing the secure files run. We stress that this minimal level of trust is necessary, as the kernel can read the user space anyway and thus get the secure files after they have been decrypted.

MCCFS files can be stored in any location, such as servers, client computers, and mobile devices, without performing or changing any additional access management. The secure virtual disk driver performs requests, authentication, read/write, and other operations with the remote machines that are transparent for users. This

convenient key management infrastructure make it easy to implement the different files, encryption and combine various key management mechanisms. Finally, MCCFS separates the server's diversity, the key's generation and distribution, and the secure information's structure. In short, MCCFS succeed in being a system that provides mobile certifying, security, and versatility (Fig. 3.1).

3.3 Core Security Management

3.3.1 Attacks

Existing systems for network security have mostly addressed the compromise of short-lived data and the protocols used to communicate them. In addition to securing data on the wire, storage systems must also secure long-lived data on the remote server or local client computer. The attacks can be divided into three kinds, based on the effect they have on the data:

1. Leak attacks are those where the adversary gains access to some data.

2. Change attacks are those where the adversary makes valid modifications to data.

3. Destroy attacks are those where the adversary makes invalid modifications to some stored data. An invalid modification is any change to data that is detectable as incorrect by the owner or readers.

3.3.2 Authorization

The purpose of authorization is to allow the owner of some data to delegate access to the data to another user. In MCCFS, authorization can be done in two ways:

Administrator-handled The administrator of a group can access all of the files created by the group's member; at the same time, they can authorize the read or write rights to other users.

Owner-handled The file, owners can provide readers and writers with keys that they can use to authorize or perform actions. For example, Bob can add Alice to his file's ACL and authorize the read/write right. Then, once Alice accesses Bob's machine, she can mount the network shared file to her local machine and perform the actions authorized by Bob.

3.3.3 Authentication

Because users can not necessarily trust file systems with their passwords, all user authentication for remote or local file systems use private keys stored in a USB token in MCCFS. Every user on an MCCFS client machine logs in the authentication agent process using the MCCFS client software. The authentication agent holds one or more of the user's private keys.

The purpose of authentication is to establish the identity of a particular user in order to authorize their actions. There are two mechanisms to achieve user authentication: a user's password and a password-based USB token containing a private key. The usual concern is about authentication of owners, readers, and writers on storage machines.

In MCCFS we identified four groups of users:

System Administrator The system administrator can enforce usage polices. The system administrator is trusted to properly install the MCCFS kernel and user-space components. However, the system administrator is not trusted with encryption keys.

Owners create and destroy data and control the encryption key for the data. The owners delegate reading and writing permission to other users and revoke another users' privilege to read or write owned data.

Readers and Writers All other authorized users are either readers or writers. The only difference between an owner and a reader or writer is that the owner is implicitly a reader or writer, depending on the permission that the system administrator delegates. For the system to be convenient, readers and writers must be able to use encryption transparently. Authorized readers and writers must also be able to delegate permission received from other authorized readers and writers.

Adversaries Any user who attempts to perform functions exceeding their delegated permissions is considered an adversary. Adversaries include legitimate users attempting to perform actions beyond what they are authorized to.

In MCCFS, users do not need remember the decryption password associated with the files. We adopted two factor authentications, the user's certificate and the USB token. Users just need to remember the password of the USB token, whose password is easier than the file's password, which consists of 64 characters of numbers and letters. This allows the use of random passwords, which are highly secure and difficult to

crack. And each file has its own password, which is transparent to the user. In other words, no matter how many different files the user encrypted, they only need to remember one password.

Once they pass the authentication, they can access the USB token belonging to them. The password of a file is saved in the header of the file, and the header is encrypted by the user's public key. MCCFS decrypts the file's header based on the private key saved in the USB token and gets the file's password from its header.

3.3.4 Securing data on disk

The reasons one may want to encrypt data on the disk are that the server is inherently mistrusted or the server might be compromised, such as a stolen disk or laptop. To guarantee that the data and metadata are not compromised, they must be stored and encrypted on disk. To accomplish this encryption, two types of ciphers may be used here:

- symmetric cipher - a single private-key system, such as DES or AES, that is used to perform bulk data encryption and decryption.
- asymmetric cipher - a system using a pair of keys, such as RSA, that is generally used for authentication and to bootstrap the shared keys to be used by the symmetric cipher.

Since computing asymmetric ciphers is much slower than symmetric cipher ciphers, these operations are used to protect stored symmetric passphrases. At the same time, MCCFS uses the long-lived symmetric passphrase to encrypt all data and file names written to disk. If we used a short-lived key, then whenever the key

changed, all data would have to be re-encrypted. To avoid this performance penalty, we use long-live keys.

In MCCFS, we require that all sensitive meta data should be hidden. So all the information about the file is encrypted.

This means that not only the content of each file but also the filename, if encrypted. All the sensitive information is encrypted and saved as a file under any name, which prevents others from guessing the information from the filename or directory structure. So on the same file system space, there is no difference in appearance to distinguish *private* data from the *normal* data, such as files which do not require encryption. The security of the content of files is guaranteed by means of Advanced Encryption Standard (AES) cryptography.

So users can be authorized in any machine, and the cost is kept to a minimum with high security on the data:

1. MCCFS should be completely transparent to the file system and should not modify structures of the file system itself. After encryption, the whole encrypted fold looks like a normal file, which can be copied, pasted and renamed without any change to the data itself. So the encrypted data can be portable and decrypted at any machine.

2. The access semantics to secure files should not change. Secure files should be accessible using the same standard systems calls. Once the secure file has been mounted by the MCCFS application, no additional flag in the *open* and *create* operation needs to be used. This is necessary to make possible the use of software unaware of MCCFS, without recompiling. When new files are created in the secure virtual drive, they are by default created secure, similarly, to files

created in a non-secure drive. Thus, all temporary files created by applications running in a secure environment are by default created secure.

3. The secure file can be shared on mistrusted storage servers. Most existing secure storage solutions require the creators of data to trust the storage server to control all users' access to this data. MCCFS strives to provide strong security even with an untrusted server. All data is stored encrypted, and all key distribution is handled in a decentralized manner. All cryptographic and key management operations are performed by the clients, and the server incurs very little cryptographic overhead.

3.4 Access Control

We wish to provide an automated system to allow the Access Control List to assert its authority over a file in a flexible manner, consistent with the scope of its authority. In MCCFS, users can access their files from any machine they trust, anywhere in the world. They can share files across organizational boundaries without the intervention of network administrators.

MCCFS makes the assumption that underlying storage media can be read and tampered with, so to ensure file confidentiality, it must be encrypted. MCCFS relies on encryption to protect data, and each file has its own symmetric password, which is used by writers to encrypt and by readers to decrypt data directly at the edges of the system.

Authorization and access control is the process by which the MCCFS determines whether a user should be allowed to perform a certain action. Authorization takes

place after the user has been authenticated. Before the encrypted file is accessed, the user must provide the USB token containing the user's private key and the user's passphrase to MCCFS.

Furthermore, authorization occurs within the scope of an access control policy. In simpler terms, the first step in making an access control decision is determining who is making a request. The second step is determining, based on the result of the authentication as well as additional information (the access control policy), whether that request should be allowed. Once the users pass the two-factor authentication, we need the necessary checks mechanism to make sure they perform correctly. Before users create, read, or write the files, the system will assign their corresponding rights based on the files' ACL and their certificate.

ACL ensures that only authorized readers or writers are able to access the file or create valid new files (Fig. 3.2).

3.4.1 Two-Factor-Based Access Control

The access control mechanism must allow secure, distributed management of certificate-based access to resources and provide transparent access for authorized users and strong exclusion of unauthorized users in an operating environment where fileholders, users, and administrators may never meet face to face.

Each people should be able to make their assertions without reference to a centralized system administrator who must act on its behalf. The mechanism must be dynamic and easily used by all concerned, fileholders and users, while maintaining strong assurances. The solution should scale with the number of fileholders, resource and users.

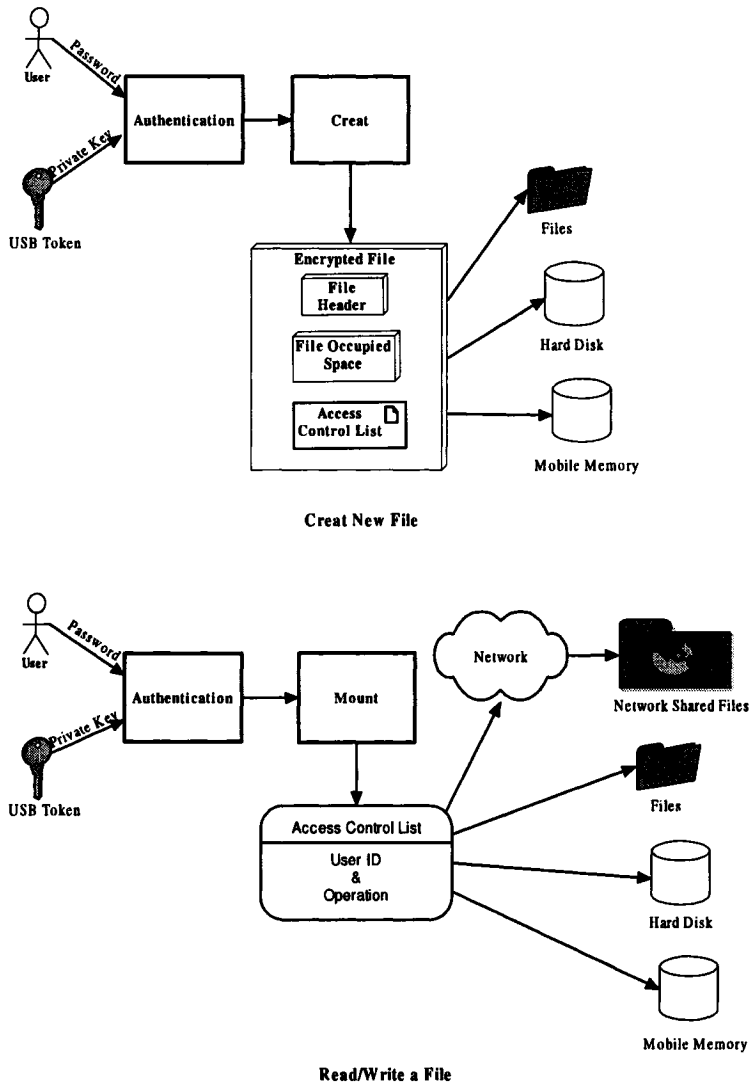


Figure 3.2: MCCFS Access Control: Create, read, write

In particular, the access control mechanism should be able to collect all of the relevant assertions (identity, file-using conditions, and corresponding user attributes) and make an unambiguous access decision requiring an absolute minimum of centrally administered configuration information.

Our approach to two-factor-based access control in a distributed environment is based on digital singed documents, or certificates, that convey identity, authorization and attributes. A digital signature can assert documents' validity without the physical presence of the signer or physical possession of documents signed in the author's handwriting. The result is that the digitally signed documents that provide the assertions of trusted authorities about the attributes of a user may be generated, represented, used, and verified independent of time and location.

Users are authenticated by presenting an X.509 identity certificate and proving that they know the associated private key. These certificates are issued by certificate authorities (CA) that verify the connection between a person or system component and possession of a public key/private key pair. Components that enable the use of these certificates include reliable mechanisms that match use conditions and attributes to decide whether access should be allowed with access methods that enforce policy for the specific resources based on the access control decision.

3.4.2 Access Control List

The ACL is a list describing which access rights a principle has on a file (device). We designed and implemented an access control mechanism that relies on Access Control Lists that provide the usual read and write file system operations. This mechanism has the following properties:

1. ACLs allow fine-grained access control and give fileholders the flexibility to assign different access permissions to different users or groups.
2. ACLs allow access control to extend to any MCCFS user, whether she has an account in the local administrative realm or not.
3. Maintaining ACLs does not require the involvement of the local realm's system administrator.
4. ACLs are compatible with future extensions that will provide support for groups that are defined and maintained by a third party on any MCCFS server.

Since the MCCFS approach allows the access control list to be stored within the encrypted file in distributed environment, it is very convenient for mobile users to access and share their files in any location.

3.5 Network File Mapping

Network File Mapping (NFM) is a file system extension that provides transparent access to files on remote machines. NFM's purpose is to make remote FTP and HTTP file services look like they were local file systems. When an open system call was made on a file, if the file was located on a remote file system, NFM would satisfy the request by downloading the file's header into the local file cache and then change the open request to an open request for the downloaded file's header in the cache. Then the MCCFS can mount the remote file in the same manner as the file saved in the local machine.

From Figure 3.3, we can see that users can map any encrypted files in any remote server to a network drive that can used as a local drive in a secure way. Each secure

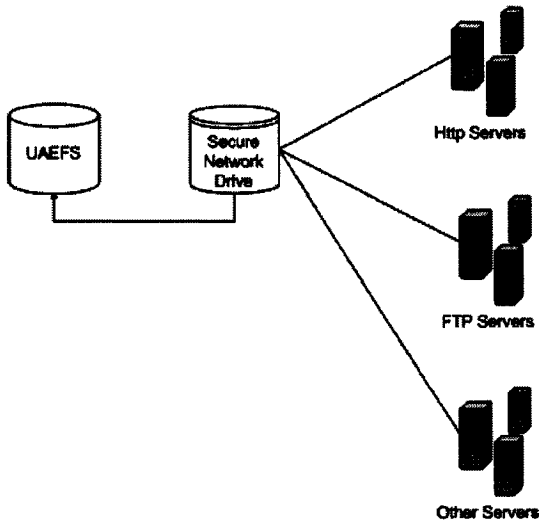


Figure 3.3: Network Drive Mapping

network drive contains the encrypted files and the Access Control Lists. The drive will use it to decide whether or not to grant a write operation to the user. A private key that will be used to validate write requests to the encrypted file can be loaded automatically.

Chapter 4

MCCFS Implementation

We implemented MCCFS in Windows XP with service package 2. For the purpose of quickly getting most of the functionality to work, we adopted some encryption algorithm coding from the TrueCrypt [29].

Figure 4.1 is the architecture of the MCCFS:

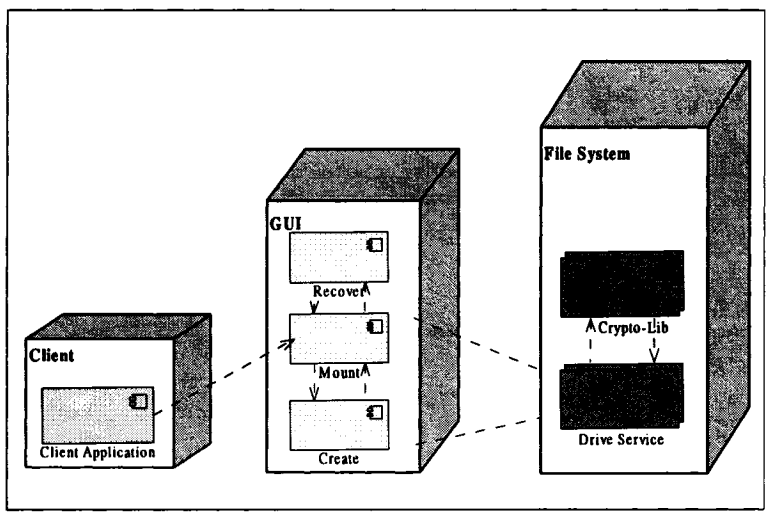


Figure 4.1: MCCFS Architecture

4.1 MCCFS File Format Specification

MCCFS file has no "signature". Until decrypted, it appears to consist entirely of random data. Therefore, it is impossible to identify a MCCFS file or partition. Although the data seems random, the whole file space consists of four parts.

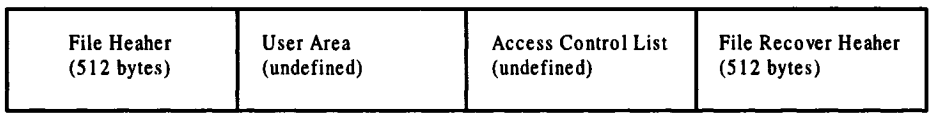


Figure 4.2: MCCFS File Format

4.1.1 File Header Format

Figure 4.3 presents the format of MCCFS's file header.

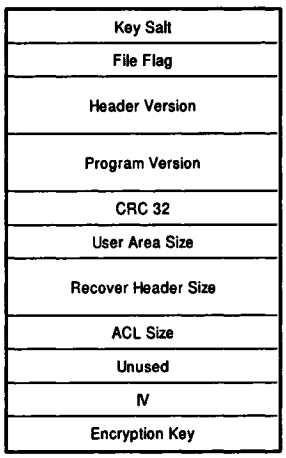


Figure 4.3: MCCFS File Header Format

Key Salt

Key Salt (512 bits) is picked up from the random number pool (we will explain the random number pool in the next section).

File Flag

File Flag is used to check whether the decryption of the file header is correct.

CRC 32

CRC 32 is the checked sum of the Encryption.

Initialization Vector (IV)

IV is always a random value (unknown to an adversary) generated by the random number pool, which is unique to each sector (each sector is 512 bytes long, and sectors are numbered starting 0) and file. This value is generated as follows:

(1) Bytes 256-263 (for a 128-bit block cipher, bytes 256-271) of the decrypted file header are retrieved. If a cipher in a cascade is used more than one 129-bit block ciphers in a cascade in inner-CBC mode, bytes 256-271 are retrieved for the first IV, bytes 272-287 for the second, and 288-303 for the third IV.

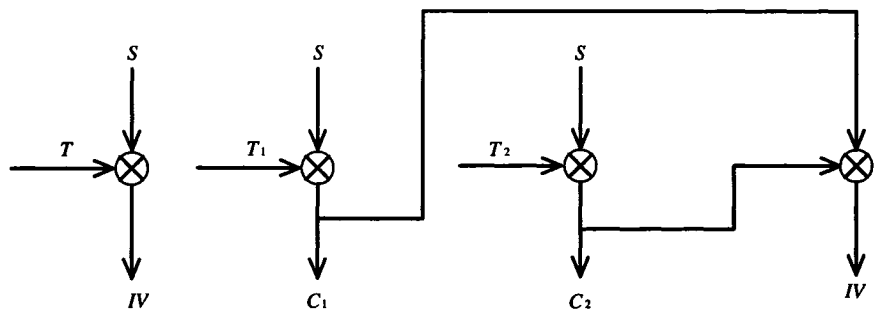
(2) Data retrieved in (1) are XORed with the 64-bit sector number. In case of a 128-bit block cipher, the upper and lower 64-bit words of the 128-bit value retrieved in (1) are XORed with an identical value. The resultant 64-bit value (or 128-bit for 128-bit block ciphers) is the IV (Fig. 4.4).

For a 64-bit block cipher:

$$T = \texttt{64-bit value retrieved in (1)}$$

$$S = \texttt{sector number (64-bit unsigned integer)}$$

$$IV = T \wedge S$$



(a) Cipher 64 (b) Cipher 128

Figure 4.4: IV Generation

For a 128-bit block cipher:

$$T_1 = \text{upper 64-bit word of the value retrieved in (1)}$$

$$T_2 = \text{lower 64-bit word of the value retrieved in (1)}$$

$$S = \text{sector number (64-bit integer)}$$

$$IV = (T_1 \wedge S) \oplus (T_2 \wedge S)$$

Step (1) is only performed once, right after the file is mounted. Then the retrieved data remain in RAM.

Encryption Key

Encryption Key is generated by the random number pool and will be used in encrypting the user area data together with the IV.

4.2 Cryptographic Implementation

In MCCFS, we implemented eight encryption algorithms. Table 4.1 lists their operation modes.

4.2.1 Random Number Generator (RNG)

In MCCFS, the Encryption key, Key Salt, IV, and Whitening seeds are generated by the RNG. The RNG creates a pool of random values in RAM. In order to balance security and time consumption, we combine the pseudo-random number generator and entropy harvester methods together. The random number pool, which is 256 bytes long, is filled with data derived from different sources. We need entropy in a low-entropy environment, so we gather the entropy from the keyboard and mouse events.

For example, we collect the movement of the user's mouse and mix it into the random number pool. There can be a reasonable amount of entropy in mouse movement. The entropy comes not just from where the mouse pointer is on the screen, but from when each movement was made. In fact, the mouse pointer's position on the screen can have very little entropy in it, particularly in an environment where there may be very little interaction from a local user. Most of the entropy will come from the exact timing of the mouse movements. The basic methodology is to mix the on-screen position of the mouse pointer, along with a timestamp, into the random number pool.

Before a value obtained from any of the sources is written to the pool, it is split into bytes (e.g., a 32-bit output of CRC-32 is split into four bytes). These bytes are then individually written to the pool with the modulo 2^8 addition operation (not by

Table 4.1: Encryption operation mode

Encryption Algorithm	Operation Mode	Details of Operation Mode
AES	CBC	$C_i = E_K(P_i \wedge C_{i-1})$;$C_0$ is IV.
AES-Blowfish	Inner-CBC	$C_n = E2_{k2}((S_m \parallel S_{m+1}) \wedge C_{n-1})$; $S_m = E1_{K1}(P_m \wedge S_{m-1})$; C_0 and S_0 are IVs.
AES-Blowfish-Serpent	Inner-CBC	$C_n = E3_{k3}((S_m \parallel S_{m+1}) \wedge C_{n-1})$; $S_m = E2_{K2}(P_m \wedge S_{m-1})$; $T_m \parallel T_{m+1} = Q_n = E1_{K1}(P_n \wedge Q_{n-1}$; C_0, S_0 and Q_0 are IVs.
AES-Twofish	Outer-CBC	$C_i = E2_{K2}(E1_{K1}(P_i \wedge C_{i-1}))$; C_0 is IV.
AES-Twofish-Serpent	Outer-CBC	$C_i = E3_{K3}(E2_{K2}(E1_{K1}(P_i \wedge C_{i-1})))$; C_0 is IV.
Blowfish	CBC	$C_i = E_K(P_i \wedge C_{i-1})))$; C_0 is IV.
CAST5	CBC	$C_i = E_K(P_i \wedge C_{i-1})))$; C_0 is IV.
Triple DES	Out-CBC	$C_i = E_{K3}(D_{K2}(E_{K1}((P_i \wedge C_{i-1})))$; C_0 is IV.

replacing the old values in the pool) at the position of the pool cursor. After a byte is written, the cursor position is advanced by one byte. When the cursor reaches the end of the pool, its position is set to the beginning of the pool. In addition, after a value (byte) is added to the pool, the pool is entirely hashed using a hash function, SHA-1 or RIPEMD-169. After every eighth byte written to the pool, the pool mixing function is applied to the entire pool (Fig. 4.5).

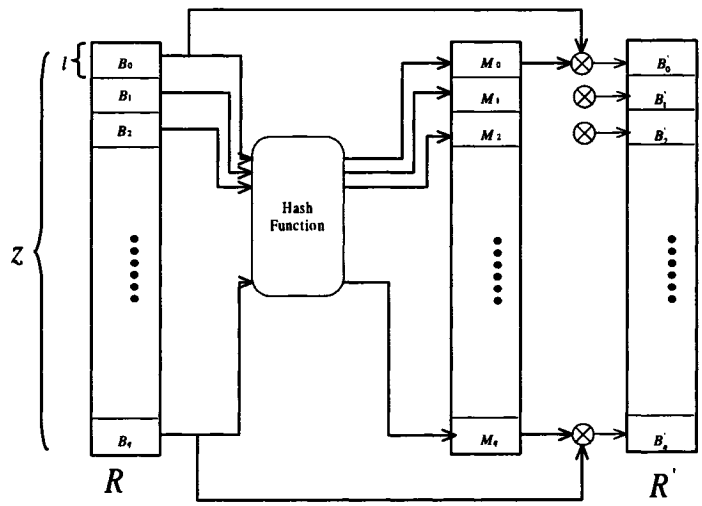


Figure 4.5: Random Number Generate Pool Mixing Function

Here, R is the Random Number Generate Pool, whose size length is z. For the specified hash function, such as SHA-1 or RIPEMD-160, we divide the R into q blocks $B_0, B_1, B-2, ..., B_q$, where $q = z/l - 1$ and l is the byte size of the output of the hash function (H). For each block $B_i, 0 \leq i \leq q, M_i = H(B_i), B_i' = B_1 \wedge M_i$. So we get the mixing pool R' consisting of $B_0', B_1', B_2', ..., B_q'$.

4.2.2 Whitening

In order to prevent an adversary from obtaining a plaintext/ciphertext pair, we applied the whitening algorithm. Every eight bytes of each sector (after the sector is encrypted) are XORed with a 64-bit value, which is unique to each sector and volume (and is unknown to an adversary). The value is generated as follows (Fig. 4.6):

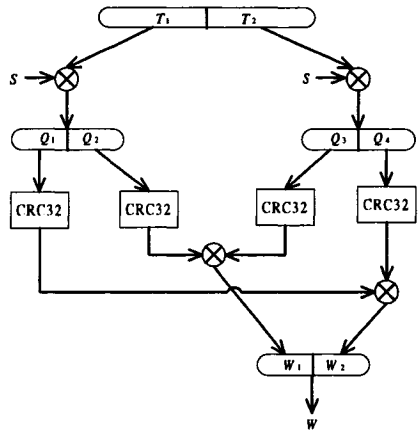


Figure 4.6: Whitening Generation (Cipher 128)

(1) Bytes 264-271 (for a 128-bit block cipher, bytes 272-279) of the decrypted file header are retrieved: T_1.

(2) Bytes 272-279 (for a 128-bit block cipher, bytes 280-287) of the decrypted volume header are retrieved: T_2.

(3) Data retrieved in (1) are XORed with the 64-bit sector number: $T_1 \wedge S$.

(4) Data retrieved in (2) are XORed with the 64-bit sector number: $T_2 \wedge S$.

(5) A 32-bit CRC-32 value of the first 8 bytes of the resultant value (Q_1) in (3) is calculated: ($CRC32(Q_1)$).

(6) A 32-bit CRC-32 value of the second 8 bytes of the resultant value (Q_2) in (3) is calculated: ($CRC32(Q_2)$).

(7) A 32-bit CRC-32 value of the first 8 bytes of the resultant value (Q_3) in (4) is calculated: ($CRC32(Q_3)$).

(8) A 32-bit CRC-32 value of the second 8 bytes of the resultant value (Q_4) in (4) is calculated: ($CRC32(Q_4)$).

(9) The value calculated in (5) is XORed with the value calculated in (8): $W_1 = (CRC32(Q_1)) \wedge (CRC32(Q_4))$.

(10) The value calculated in (6) is XORed with the value calculated in (7): $W_2 = (CRC32(Q_2)) \wedge (CRC32(Q_3))$.

(11) The 32-bit value calculated in (9) is written to the upper 32-bit word and the value calculated in (10) is written to the lower 32-bit word of the 64-bit whitening value: $W = W_1 \parallel W_2$.

4.2.3 Header Encryption Deriving Function

The MCCFS file header is encrypted and decrypted by the header key, which is derived by the PBKDF2 algorithm, specified in PKCS #5 v2.0 [30]. PKCS #5 PBKDF2 was chosen because it is well understood and can generate keys of different lengths safely. The use of PBKDF2 addressed perhaps the most common weakness of otherwise designed crypto-systems, namely dictionary attacks against the passphrase. Since the method uses chained HMACs and includes an iteration count, it is possible to configure enough iterations to make a dictionary attack arbitrarily prohibitive at a

fixed configuration-time cost. PBKDF2 needs a passphrase and a salt to derive a file header key.

In MCCFS, we adopted the user's private key saved in the USB token as the passphrase. Different from the traditional password remembered by users, a user's private key is longer and more randomly includes the numbers and uppercase and lowercase characters. This should provide a reasonable level of protection for the foreseeable future. If we assume that the user's passphrase contains n bits of entropy against a dictionary attack, then it will take 2^n seconds to crack. If Moore's Law is that computer speed will double approximately every 18 months, then in m years it will take $2^{n-2m/3}$ seconds to crack. For $n = 40$, we get that today it should take 2^{40} seconds = 38865 years. Although the work can be spread over many machines, this is still quite a formidable amount of time. When $m = 10$, however, it will take 2^{40-15} seconds = 1.06 years, which is still a considerable amount of time, but within the capacity of an attacker with sufficient resources.

As one of the algorithm parameters, the salt can prevent the off-line attack. Since it is unlikely that the entropy contained in the passphrase that the user can remember will increase with Moore's Law, it is essential that the default iteration count is chosen with care. The PBKDF2 algorithm is run infrequently, so it is acceptable that it takes a reasonably long time to derive the key from the passphrase.

In MCCFS, a 512-bit salt is used, which means there are 2^{512} keys for each password. This significantly decreases vulnerability to "off-line" dictionary attacks. Two thousand iterations of the key derivation function have to be performed to derive a header key, which significantly increases the time necessary to perform an exhaustive search for passwords. The header key derivation function is based on HMAC-SHA-1

or HMAC-RIPEMD-160.

Header keys used by the ciphers in a cascade are mutually independent, even though they are derived from one password. For example, for the AES-Twofish-Serpent cascade, the header key derivation function is instructed to derive a 768-bit key from a given password. This key then splits into three 256-bit keys, out of which the first key is used by Serpent, the second key is used by Twofish, and the third key is used by AES.

The MCCFS also provides the ability to generate additional parameters files, which generate the same key. To do this, it fully evaluates the original parameters file including asking for passphrase and retrieving the keys from a key server producing key K_1. It then generates a new parameters file (which may use different key generation methods) and evaluates it producing key K_2. It then appends a key generation stanza of method storekey where the stored key is $K_1 \oplus K_2$. The new parameters file will generate the same key: $K_2 \oplus (K_1 \oplus K_2) = K_1$.

The ability to generate new parameters files allows the user to change their passphrase without rekeying the disk. It also allows multiple administrators to access the disk with different passphrases.

4.3 MCCFS Drive Service

In MCCFS, when a user wants to mount a MCCFS file, the drive service (DS) will create a secure virtual disk drive (SVD), which is a pseudo device drive that turns a Windows file into a block device. There are several places in the kernel where we could have implemented such functionality.

Figure 4.7 shows the steps involved in mapping a remote file to the local operation

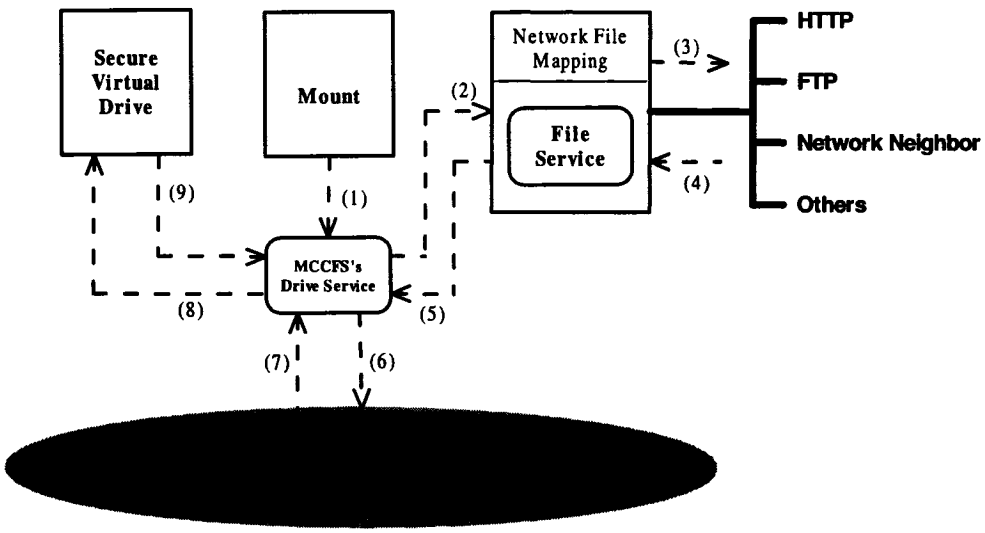


Figure 4.7: Secure Virtual Drive Service

system based on NFM. When the application issues a system call, the following occurs:

(1) If it is file-related, the call gets intercepted by the MCCFS's drive service.

(2) For intercepted calls, the drive service determines whether the system call operates on a remote or a local file.

(3) If the file is remote, the file service sends the file request to the remote machine.

(4) When remote file is requested successfully, the file service creates a local cached copy.

(5) If the file is local, the request proceeds unmodified. If the file is remote, the drive service patches the system call by modifying its parameters.

(6) After the request is serviced in the drive service, it will let the request proceed to the operation kernel.

(7) The result is returned to the MCCFS's drive service.

(8) A new secured virtual drive is created.

(9) All the user operations are on the virtual drive, and all the changing requests to the encrypted file will be sent back to the drive service.

A hook is placed between the system calls and the low-level data storage facilities to effect encryption. The hook is for encrypting/decrypting the data read from or written to a file on the storage.

4.3.1 File Operations on SVD

All the file operations, such as create, read, write, or execute, on SVD will go through the MCCFS's drive service, which executes the corresponding encryption or decryption operation on the mounted MCCFS file.

For each process accessing a file, the DS will acquire a passphrase for this file/process pair. This key will remain in the RAM until the process terminates or the file is closed. As a page may be mapped multiple times into different processes, that might disagree on the point of which key to use to decrypt the same file, the page cache itself must be made aware of the encryption. It must assure that only a process that has the key to a file can map or access one of its pages decrypted. This is done by unmapping the page from all processes that would use the page with a different key when a page is accessed with read/write or mapped into a process address space after a page fault. A process that accesses a page that was unmapped from it will enter a page fault as if the page were never mapped before and will fetch the page back, removing it from the other processes competing with it in turn.

When a dirty page has to be written to disk, the page will need to be encrypted and unmapped from all processes too. Two locks are used to keep the page in an encrypted

or decrypted state, respectively, when buffer IO or read/write system calls are active on this page. One other implementation detail is that, to avoid page bouncing, the encryption has to be deferred for as long as possible to skip over partial modifications to a page. Since the actual asynchronous write is handled by the buffer cache, it was reasonable to move the encryption time back to just before the buffer cache schedules to write the page. Since the write is delayed, the encryption will in most cases skip out of the time interval needed to do the write altogether, and therefore the page is only decrypted and encrypted once.

For the network file accessing based on NFM, which does not use the buffer cache, this procedure is not used, as delayed writes are not done in this case. The write will instead be synchronous, and the host usually has to wait for a reply from the server anyway. The page might be encrypted and decrypted multiple times, but the delays imposed by this are small in comparison to network access times. If performance issues should indicate that another solution to this is necessary, modifications will need to be done on the file system level.

4.4 Graphic User Interface

The graphic user interface (GUI) handles file passphrase management, distribution, storage, and revocation. While the drive service (DS) handles symmetric encryption keys for different files, the GUI counterpart has to somehow figure out where to mount a given file and get the corresponding encryption key and which processes/user should be granted access.

4.4.1 Create Program

The Create Program will collect data that will be used in generating the encryption key, the salt, and the IV's seeds and whitening values for the new file by using the hash algorithm. The collected data include mouse movements, key presses, and other values obtained from the system.

A new MCCFS file is created using the following steps (Fig. 4.8):

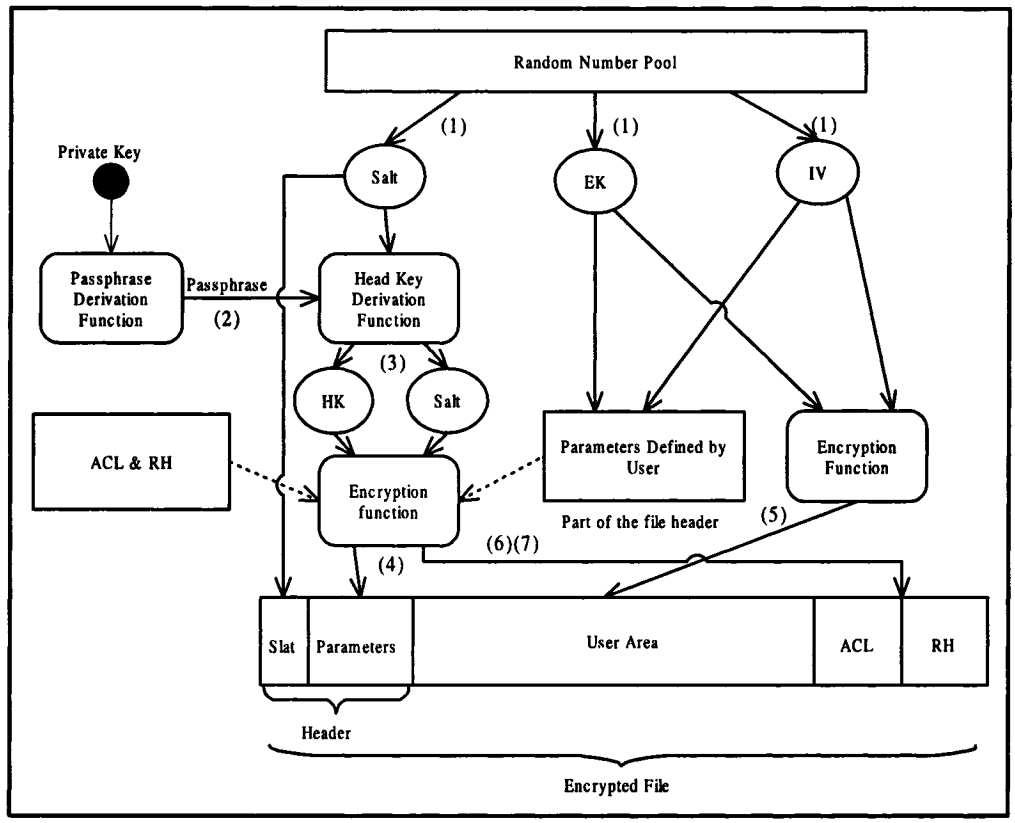


Figure 4.8: Create File

(1) Initialize: The user chooses the parameters: the file space, access control conditions, hash algorithm, and encryption algorithm. A salt, encryption key (EK),

plain text block, IV, and whitening seeds are generated by the random number pool.

(2) The passphrase derivation function generates the file passphrase based on the user's private key.

(3) The head key derivation function generates the Header Key (HK) based on the file's passphrase and salt.

(4) The program initials the file header with the parameters generated by the system or chosen by the user and encrypts the file header, except for the salt.

(5) The program fills the file user area with cipher text blocks based on the encryption algorithm by EK.

(6) The program encrypts the ACL with the user's private key and write it to the file.

(7) The program encrypts the recover header (RH) and writes it to the end of the file.

Once the file is created, the unused user area is filled with random data.

4.4.2 Mount Program

After users click Mount Program , MCCFS will detect whether the USB token is connected to the computer. When a valid USB token existes, it prompts users for a password for the token. If they enter the correct password, the file will be mounted.

When mounting an MCCFS file, the following steps are performed:

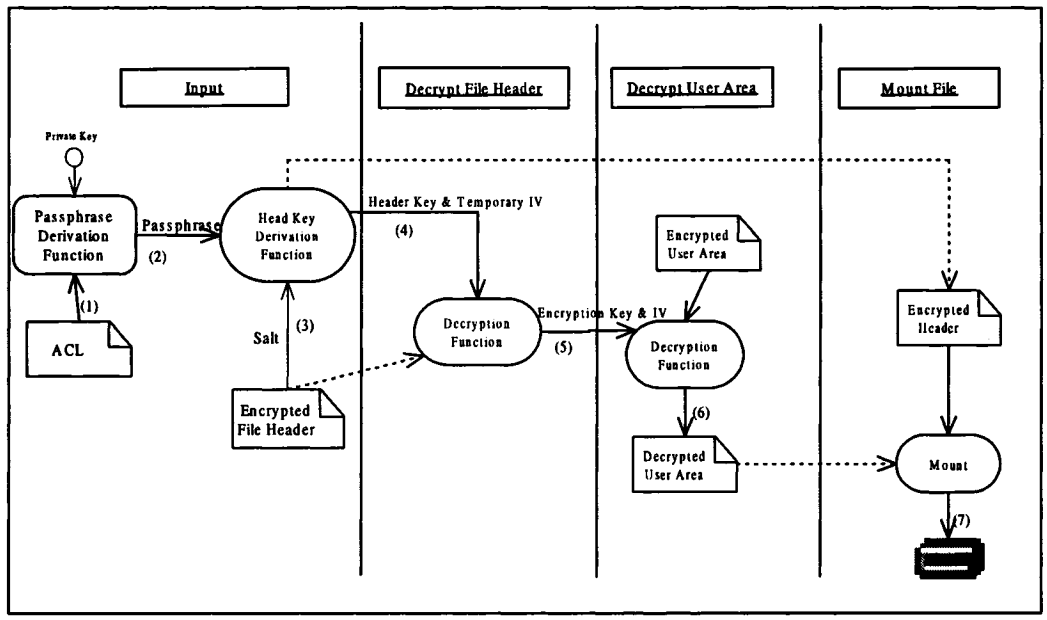


Figure 4.9: Mount File

(1) The passphrase derivation function decrypts the ACLs based on the user's private key.

(2) A passphrase generated is by the passphrase derivation function.

(3) The head key derivation function reads the file header (the first 512 bytes) into RAM.

(4) The passphrase and salt (the first 512 bits read in Step 1) are passed to the header key derivation function. Then the header encryption key and a temporary IV are calculated by the derivation function.

(5) The decryption function attempts to decrypt the MCCFS file header. All data used and generated in the course of the process of decryption are kept in RAM. The pseudo random functions (PRF) generates the header key (HMAC-SHA-1

or HMAC-RIPEMD-160 algorithm). If the first 5 bytes of the decrypted data contain the ASCII string "MCCFS", and if the CRC-32 checksum of the last 256 bytes of the decrypted data (file header) matches the value located at the 8^{th} byte of the decrypted data (this value is unknown to an adversary because it is encrypted), decryption is considered successful. If these conditions are not met, mounting is terminated (either there is a wrong password, a corrupted file, or it is not a MCCFS file).

(6) Reinitialize the encryption routine using the Encryption Key and IV retrieved from the decrypted file header. This key can be used to decrypted the user area.

(7) Check the user's rights based on the user's private key and the ACL. Mount the file through the MCCFS drive service.

4.4.3 Recover File

Loss or theft of USB tokens might leak the user's file information and obstruct these files that have been accessed. The private key in the token might be safe in a relatively short period of time, since the token required a security password being entered before to been used. As for the encrypted files, MCCFS will distribute a new token to recover the user's files and invalidate the previously distributed one.

There is only one recover code (RC) corresponding with an USB token. When a user loses his USB token or forgets its password, he can request a new USB token with a new certificate in it to recover files (Fig. 4.10).

(1) The user specifies that an original file needs to be recovered and gets the recover coder (RC) from the RC generate function.

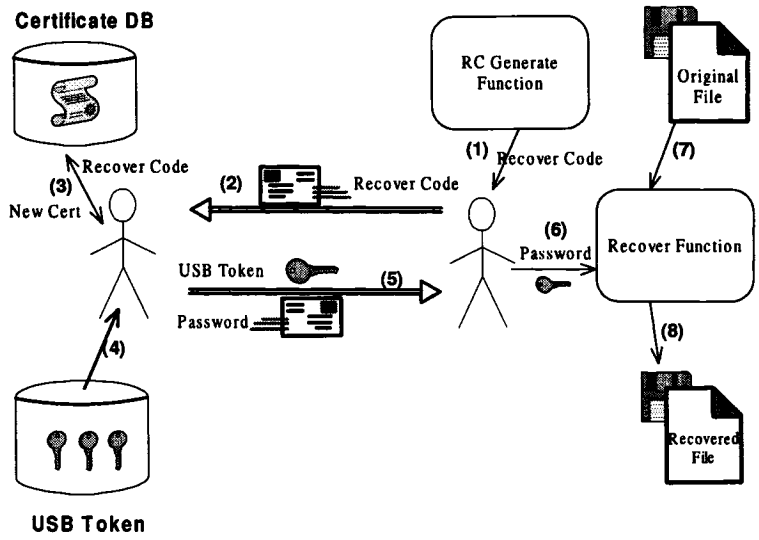


Figure 4.10: Recover File

(2) The user sends the request to the administrator, including the RC.

(3) The administrator inputs the RC to the certificate data base and gets a new certificate to match the RC.

(4) The administrator imports the certificate to the USB token.

(5) The administrator delivers the USB token to the user and sends him the token's password by email.

(6) The user plugs in the token and inputs the password.

(7) The recover function checks whether the USB token matches the original file.

(8) The recover function recovers the file and rewrites the recover header to the end of the file.

Chapter 5

Performance

5.1 Attack Prevention Capabilities

When data is saved in the local machine or transported over the Internet, it is exposed to different kinds of attacks (Table 5.1).

Table 5.1: Attack Classification

Attack	Content
Data Leak	*telecom eavesdropping, system penetration, laptop theft, theft of information, unauthorized access by insiders*
Data Change	*system penetration, unauthorized access by insiders*
Data Destroy	*system penetration, laptop theft, sabotage, virus*

Though the details of the schemes used differ, the core set of security in the cryptographic file systems proposed implement security features to prevent the attacks mentioned above. The comparison in Table 5.2 summarizes the characteristics of

MCCFS and the systems discussed in Chapter 2 and which attacks each system addresses. All of the systems are described for use on a local file system. They could also be used as mounts over a remote file system with protection of the communication to the remote server. We assume the remote server is mistrusted and the encryption keys belong to the owners only.

Table 5.2: Attack Prevention Capabilities

	Local machine			Remote server		
System	leak	change	destroy	leak	change	destroy
CFS	yes	yes	no	yes	yes	no
SFS	yes	yes	yes	no	no	no
Windows EFS	yes	yes	no	yes	yes	no
MCCFS	yes	yes	yes	yes	yes	no

yes means that the system prevents that particular attack; for instance, CFS prevents attacks that leak data by stealing the storage device because it encrypts on the media.

no means that the system fails to handle that particular attack.

In order to support these attack prevention capabilities, MCCFS has the following security primitives:

- Authentication is hybrid (centralized and distributed), and authorization is distributed.
- Data is stored and encrypted on the disk.
- Data is sent in the ciphertext over the Internet.
- A user's private key is used to generate the file's encryption passphrase.

- Revocation does not require re-encrypting the whole file, such as the user area.
- Revocation is achieved by changing the access control list.
- Recovering is achieved by changing the recover header.

5.2 Test Platform

We tested two file-system groups of Windows and Windows+MCCFS by running a certain benchmark at computers, so as to understand the influence of the cryptography file system on the performance of the system and network.

Benchmark The performance benchmark IOZone[31], which is a benchmarking tool used to analyze file system performance on a number of different platforms, was used to measure the performance of MCCFS and Windows without it. IOZone uses file system I/O as its primary load generation, presenting the system under tests with a large range of file I/O requests, running from small to very large file sizes, with varying request sizes, while performing a number of different file access patterns. The IOZone benchmark program used here has the following qualities:

- Highly configurable
- Generates "reproducible" results
- Generates "reproducible" usage scenarios
- Very fine degree of tenability

Test Modes In our experiments, there are three test modes that can be used. The first is the local mode, where all data that a given client thread needs is located

on the local disks to where the thread is running. The one-to-many mode is basically an extreme of the client-server mode, where there are many client threads making requests for data and that have only a server thread. The last mode is peer-to-peer mode where two client threads are collocated in locate disks and request for data between them.

Experiment Environment The three modes' experiment environments are as follows:

local We used IOZone to measure a local file system behaviour throughput on a client computer with an Intel Pentium 4 CPU 2.40GHz, 256M of RAM running Windows XP Professional Version 2002 Service Pack 2.

peer-peer The experiment was on two machines both with an Intel Pentium 4 CPU 2.40GHz, 256M of RAM running Windows XP Professional Version 2002 Service Pack 2.

client-server The performance results of MCCFS are obtained on the Intranet consists of 5 client nodes and 1 file storage server. The nodes were interconnected through a Fast Ethernet. In our experiment, the client side was a PC installed with Window XP, with SP2, 2.4GHz Intel Pentium 4 Processors, 256 Mbytes of memory, and one 40 Gbyte SCSI disk, and the server was a PC installed with Window XP, with SP2, 2.8GHz Intel Pentium 4 Processors, 512 Mbytes of memory, and one 80 Gbyte SCSI disk. Each machine had a 10/100Mbps auto-negotiation Ethernet card and interconnected via a 100Mbps switch.

5.3 Local Performance

Since the throughput was independent of request size, we ran the IOZone benchmark with request sizes from 4 KB to 1024 KB, and read and write (Fig.5.1) experiments were done for file size of 1024 KB and 1 GB. The 1 GB was chosen to ensure that we were not simply reading the disk cache and since 256 MB was the memory of the machine, we could be sure the disk cache would be overfilled. The IOZone results for the other permutations had different ranges, but because the shape remained the same, we did not include them. IOZone determines the possible I/O bandwidth from a user program for both reads and writes to a single disk. We compared the performance of Windows system without MCCFS and with it.

Obviously, a significant performance hit is induced by using IOZone on an encrypted file, at least when large files are used. A slowdown is expected to occur because of the encryption and decryption. It makes sense that the cryptographic overhead when writing is lower than when reading. As writing takes longer than reading and the cost of encryption and decryption are the same in symmetric cryptography, the cryptographic overhead is proportionally smaller.

The difference between the two test cases is file size, and the big performance difference between them is directly related to the way the buffer cache delays writes. For the small file, almost everything from write to reread occurs directly in memory, because the dirty blocks are not left unused long enough to consider them worth writing to disk. Therefore, the blocks are never encrypted, and the MCCFS performance is only slightly reduced in comparison to using an unencrypted file.

For the large file, the kernel will flush out buffers even during the write, which

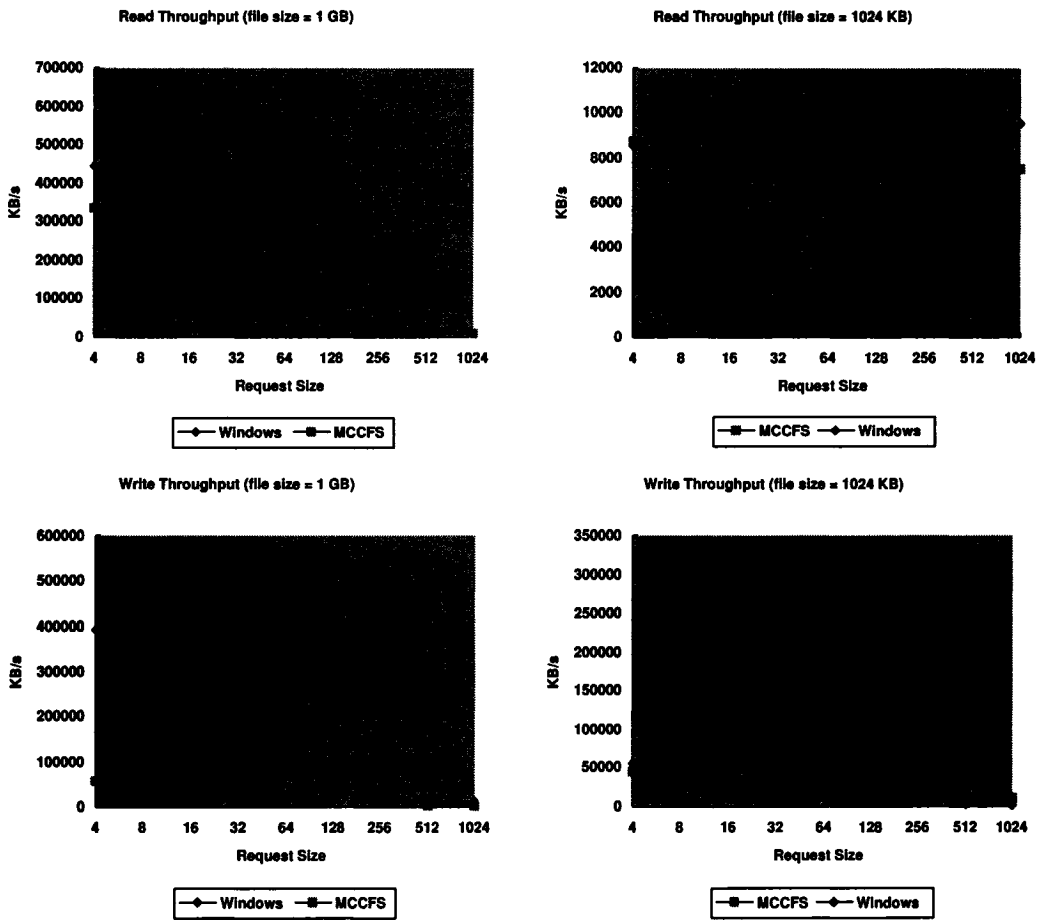


Figure 5.1: Local Read and Write Performance

does impair performance for both encrypted and unencrypted usage. These experiments demonstrated that MCCFS can achieve acceptable performance on application workloads.

5.4 Network Performance

5.4.1 Peer-peer Mode

The single client bandwidth for each file system was obtained using the IOZone filesystem benchmark. The objective of these tests was to evaluate MCCFS's performance with varying I/O request sizes for very large files. These tests were executed on both of MCCFS's clients. Using the IOZone benchmark we measured the peer-to-peer read/write bandwidth. The performance analysis was done by reading/writing a file larger than the amount of RAM available for a system in order to avoid any caching effects. The client tests read/wrote 2 GB files.

From the Figure 5.2, we can see that the single bandwidth is lower than that of Windows. This is partly because of the extra computation of MCCFS in the client machine. The last local performance experiments results had indicated that the extra cryptography computation would not impair MCCFS's speed. We can conservatively assume MCCFS is enough to support the file accessing and sharing from machine to machine for the large files.

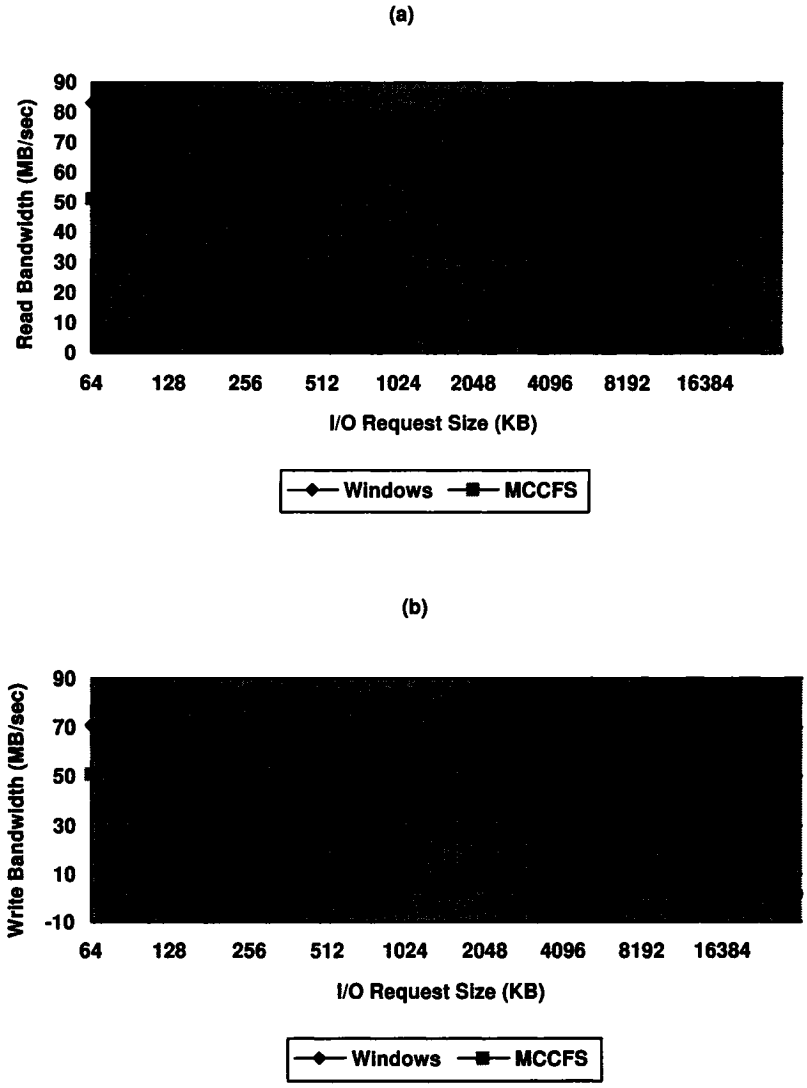


Figure 5.2: Peer-peer bandwidth by request size: (a) read, (b) write

5.4.2 Client-Server Mode

In this evaluation, the performance of the MCCFS is measured and compared with that of Windows. One IOzone process was run on each client, and the process accessed one file, and all files were located in a single directory of the file system. The bandwidth results reported by the IOZone processes were summed up to calculate the total system bandwidth. Comparison measures are aggregate bandwidth when the number of client nodes increases. To exclude the effects of the difference in the total memory size between the experiments, the size of experiment data sets is adjusted; that is, the size of data sets are three times of the total memory size.

The bandwidth is determined by three factors: the network bandwidth, the disk I/O rate, and the cost of MCCFS protocol, such as the cryptography algorithm. We did two sets of tests as before: small-file and large-file tests. In the large-file test, the I/O size was much larger than the cache size of the computer so that the disk I/O was dominant. These tests show how well MCCFS uses the bandwidth.

Figure 5.3 show the aggregate bandwidths of MCCFS for small files and large files tests respectively. In these tests, MCCFS was configured with one server, and each client node ran a test process IOZone that read or wrote a MCCFS file of specific size, which was $\frac{256}{N_c} Mbytes$ for small-file tests and $\frac{6}{N_c} Gbytes$ for large-file tests, where N_c was the number of client computers in use. The processes accessed different files.

The read figure is largely similar to the write figure except with higher values (Fig. 5.3). We noted the notable difference between MCCFS and Windows in the write bandwidths and corresponding read bandwidths in large-file tests. We believe the difference is caused by the disks on the server computer. Table 5.3 shows the aggregate read and write bandwidths of the disk for various numbers of threads in

the server. The aggregate read bandwidth of multiple threads decreased dramatically when the number of threads was more than one. In contrast, there was no distinct decline for aggregate write bandwidth. In the performance tests of MCCFS, there were 4 working threads for each storage server to perform file read and write. That is why read bandwidth was much lower than corresponding write bandwidth in the large file tests. This result suggests that the number of threads on storage servers should be adjustable according to disk behaviour.

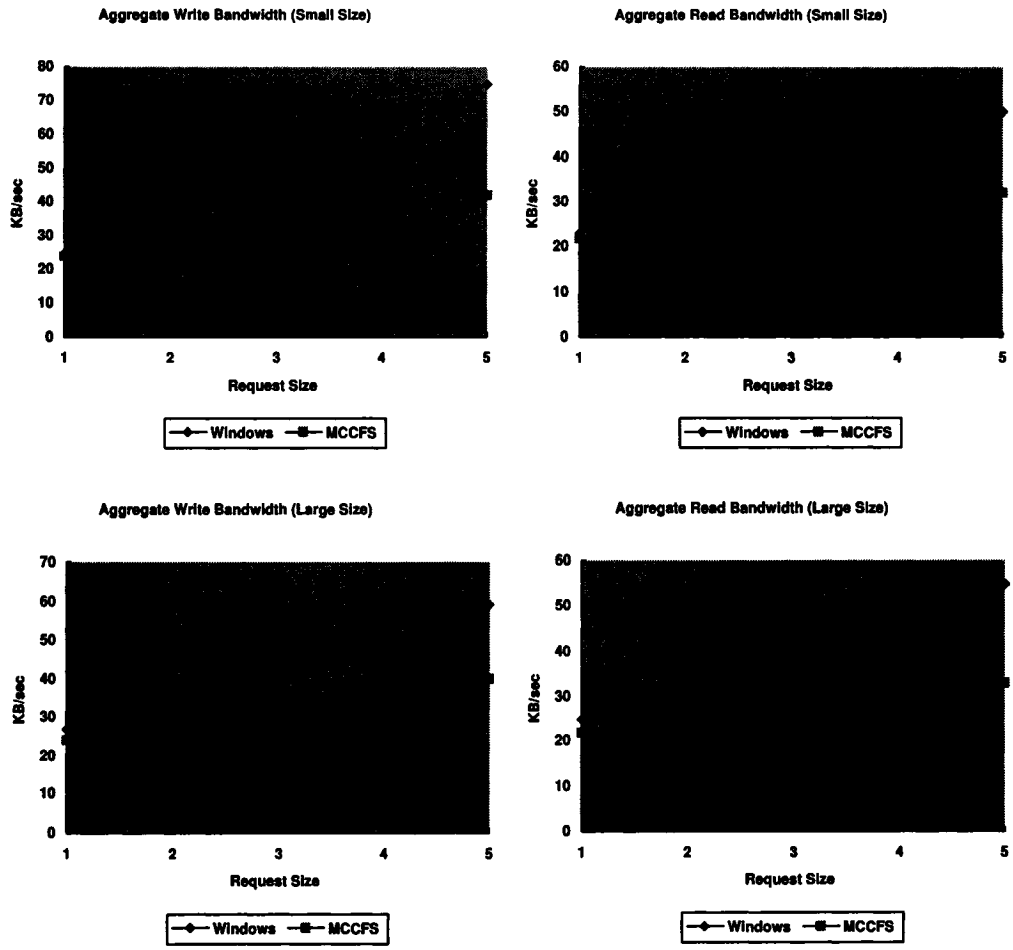


Figure 5.3: Peer-peer bandwidth by request size: (a) read, (b) write

Table 5.3: Disk Bandwidths

Threads Num.	Total Size (MB)	Read Bandwidth (MB/sec)	Write Bandwidth (MB/sec)
1	6144	69.72	65.11
2	6144	49.65	64.98
4	6144	31.27	63.29

The read/write bandwidth is measured by IOZone and the total access size is 6 Gbytes and record size is 16 Kbytes.

Because the network traffic is pre-encrypted in MCCFS, it will not reduce the file server's throughput. In the side of the client, especially for the large files, the encryption operation will reduce the client's throughput. As a result, the MCCFS's bandwidth possession will be a little lower. However, the network bandwidth is still the bottleneck in current network applications other than the local computation. So MCCFS will satisfy most network application requirements.

Chapter 6

Conclusions and Future Work

This thesis presented the design, implementation, and measurements of the Mobile Certifying Cryptographic File System. The MCCFS is an encryption file system that allows a user to access files based on a mobile certifying mechanism. The files can be saved in different locations, such as an HTTP server, an FTP server, or any client machine.

In setting out to create a secure file system, the main design considerations were flexibility, modularity, performance, and robustness.

6.1 Conclusion

6.1.1 Flexibility

File System Compatibility

The flexibility consideration used the secure virtual disk approach. Using the disk-driver encryption gives users the freedom to use any kind of file system they want on

top of it, such as FAT, FAT32, or NTFS.

Mobile Certifying

As for the authentication, the mobile certifying allows mobile users to access files from anywhere. It is more flexible for users to access and share files.

6.1.2 Modularity

It is important that the design be modular, as breakthroughs in cryptanalysis are unpredictable and frequent. Being tied to a single method for generating keys would force users to upgrade critical operating system components upon the discovery of weaknesses or vulnerabilities. Modularity also allows for the user to be tailored to a specific threat model. Allowing multiple key generation methods allows MCCFS to be used under different threat models.

6.1.3 Performance

Secure Virtual Drive

The performance consideration led to the decision to place the cryptographic disk functionality below the buffer cache; that is, create a virtual disk driver that directly accesses the raw disk beneath it. This allows all kernel-level (or even user-level) software that makes assumptions about disk layout to work.

Encryption/Decryption

Unlike the file systems mentioned before, MCCFS is not implemented on the per-file basis. When users want to access only a small part of a large file, the system will not

have the whole file processed.

6.1.4 Robustness

Access Control

The robustness consideration led to the decision to place the most complex code in the configuration utility. The user access control configuration utility performs all key management. MCCFS embeds the ACL in the encrypted file, which makes the file portable for mobile users.

File Recover

The recover header is located at the end of each file. If users forget their password or lose their USB token, they can request that the administrator to insure a new token and recover their encrypted file based on it. MCCFS will read the recover header, check users' identity, and reconstruct the file's encryption passphrase.

6.2 Future Work

More performance analysis of MCCFS is needed, especially under heavy-load conditions of large numbers of clients and under conditions of more file storage servers. Although multiple storage servers in MCCFS improve the server processing performance, they complicate the failure recovery. It is difficult to ensure that the encryption data on all storage servers is consistent when multiple users access the data and failures occur, so a better data distribution policy is needed.

Our future work is in two directions. First, we are planning to optimize the

current certification protocol to be more efficient and adaptable. Second, we are investigating possible extensions to our framework to address the group accessing and sharing problem and to integrate with other security services, such as for ad hoc network security.

Another possibility is to implement the proposed standard authorization and access control API base on XML (Extensible Markup Language) format. XML has the advantages of presenting self-describing documents and being widely used by various scientific disciplines. There are tools available for validating an XML document against its document type definition, which may be useful to the interface programs that are used to create the certificates. For example, MCCFS Client is first called when the client file system sends a file reference to it. Client then determines whether the file is encrypted and, if so, it obtains an XML header for the file. Using the USB token to digitally sign and encrypt all communications, Client sends the Server the XML header and, if the current user has access, according to the XML header, the key to the file is then encrypted to the user's USB token and sent back to client. Client then forwards the encrypted key to the USB token, which decrypts it and sends the key back. Client is then able to decrypt the file and return to the client file system, which transfers the file to the user application. The header is a human-readable ASCII text file using XML. This allows the user to examine the XML header and see exactly what it contains. It contains the author of the file, the owning group, and the Access Control List. The ACL allows the user to explicitly define who can access the data. It also allows the use of key escrow techniques, such as requiring at least two people out of a three-person group to access the data. Embedded in the ACL is the key for the file encrypted to the group server using public/private key cryptography.

Appendix A

User Manual

A.1 Setting Up MCCFS Software

(1) Under the MCCFS software set up folder, double click *Setup.exe.*

(2) Install the hardware driver for the USB token.

(3) Install the MCCFS software.

(4) Insert the USB token to verify the hardware installation.

A.2 Initial USB Token

(1) If the USB token is blank, the *Initialize* dialog will pop up automatically when you plug the token into the USB port. Users can also initialize the token at any time by right clicking on the MCCFS icon in the Windows system tray area and in the menu that appears, clicking *Initialize.*

(2) Choose *Format* to format the USB token.

(3) Choose *Import* to import the user's certificate.

A.3 Mount

(1) Plug the token into the computer's USB port. The password dialog box appears.

(2) Type the password and then click *OK*.

(3) Click *Select File* and choose the file to be mounted from different locations, such as the local machine or remote machine.

(4) Click *Mount* to mount the specified file.

(5) After successful message pops up, and users will find a new secure virtual drive in the local computer.

A.4 Dismount

There two ways users can dismount the secure virtual drive:

(1) Unplug the USB token, log off the current user or shut down the computer, and the SVD will be dismounted automatically. All of the changes users have made will be applied to the original file.

(2) When users finish their work, click the *Dismount* button in *Mount Dialog*. The SVD will be dismounted safely.

A.5 Change USB Token's Password

After the administrator issues the USB token to a user, the token is protected by the default password. Once users plug in their token to the computer USB port, they can change the password any time they want. The password (which is case-sensitive) must adhere to the following criteria:

- Minimum of 8 characters, maximum of 20 characters
- Contains at least 1 lower case character
- Contains at least 1 upper case character
- Contains numbers

Users can change the password using the following steps:

(1) In the Windows task bar system tray area, right click on the MCCFS icon.

(2) In the menu that appears, click *Change Password.*

(3) The **Change Password** dialog appears.

(4) If users did not set their own password, do the following:

   (a) Select the *System Default Password* option.

   (b) Type their new password in the *Type a New Password* and **Confirm the New Password** boxes.

   (c) Click **OK**.

If users want to change their existing password, do the following:

(a) Select the *Your Private Password* option.

(b) Type their new password in the *Type a New Password* and **Confirm the New Password** boxes.

(c) Click **OK**.

A.6 Recover Files

If users lose their USB token or forget their password, they can connect the administrator to issue a new USB token with a valid certificate. When they get their new USB token, they can recover their encrypted file using their new USB token.

A.6.1 Client

In the client side, there two steps to recover files. First, they should send the recover request to the administrator:

(1) Open the **Mount** program, click *Help*, and choose *Recover Code.*

(2) Choose the file that needs to be recovered.

(3) The **Recover Code** dialog pops up.

(4) Send the administrator the request by email including the recover code of the encrypted file.

In the second step, users receive a new USB token from the administrator. They can recover the specified file by doing the following:

(1) Insert the new USB token and input the password.

(2) Open the **Mount** program, click *Tool*, and choose *Recover file.*

(3) Choose the file that need to be recovered.

(4) Once have passed the check mechanism, a successful message pops up.

A.6.2 Administrator

(1) Open the **MCCFS Manager** and input the recover code.

(2) The system will find the common name corresponding to the recover code.

(3) Generate a new certificate with the same common name.

(4) Import the new certificate to a new USB token.

(5) Send the USB token to the user.

(6) Send the USB token's password to the user by email.

Bibliography

[1] Blaze, M., A cryptographic file system for Unix. In Proceedings of the first ACM Conference on Computer and Communications Security, 1993.

[2] Messier, M., Viega, J., Secure Programming Cookbook for C and C++, O'Reilly, July 2003, ISDN 0-596-00394-3 .

[3] Porter, S.N., "A password extension for improved human factors", Computers and Security, 1(1):54-56, January 1982.

[4] Schneier, B., Applied Cryptography. John Wiley and Sons, second edition, October 1995.

[5] National Institute of Standards and Technology. FIPS PUB 46-3: Data Encryption Standard (DES). National Institute for Standards and Technology, Gaithersburg, MD, USA, October 1999.

[6] National Institute of Standards and Technology. "Advanced Encryption Standard Development Effort", http://csrc.nist.gov/encryption/aes, 2005

[7] Rivest, R., A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM 21 (1978): 120-126.

[8] Federal Information Processing Standards Publication 186, "Digital Signature Standard", Springfield, VA: U.S. Department of Commerce, National Bureau of Standards, National Technical Information Service, 1994.

[9] ElGamal,T., "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms", IEEE Transactions on Information Theory 31 (1985): 469-472.

[10] Diffie, W., and M. Hellman. "New Directions in Cryptography", IEEE Transactions on Information Theory 22 (1976): 644-654.

[11] Koblitz, N., "Elliptic Curve Cryptosystems." Mathematics of Computation 48 (1987): 203-209.

[12] Miller, V., "Use of Elliptic Curves in Cryptography." Advances in Cryptology, Proceedings of Crypto '85 (LNCS 218) (1986): 417-426.

[13] Federal Information Processing Standards Publication 180-1. "Secure Hash Standard." Springfield, VA: U.S. Department of Commerce, National Bureau of Standards, National Technical Information Service, 1995.

[14] Kaliski, B., "The MD2 Message-Digest Algorithm." Internet Request for Comments 1319 (April 1992).

[15] Rivest, R. "The MD4 Message-Digest Algorithm." Internet Request for Comments 1320 (April 1992).

[16] Rivest, R., "The MD5 Message-Digest Algorithm." Internet Request for Comments 1321 (April 1992).

[17] International Organization for Standardization (ISO), Information technology C Security techniques C Hash-functions C Part 3: Dedicated hash-functions, ISO/IEC 10118-3:2004, February 24, 2004

[18] Gutmann, P. C., Secure filesystem (sfs) for dos/windows. http://www.cs.auckland.ac.nz/~pgut001/sfs/index.html, 1994.

[19] Jetico, BestCrypt Inc., http://www.jetico.com, 2002.

[20] Microsoft Corporation. Encrypting File System for Windows 2000, Technical report, http://microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp, July 1999.

[21] Nagar, R., Windows NT File System Internals: A Developer's Guide, pages 615-667. O'Reilly, September 1997.

[22] Riedel, E., M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002), pages 15-30, Monterey, CA, January 2002.

[23] Craig, A. N., Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata Efficiency in Versioning File Systems. In Proceedings of the Second USENIX Conference on File and Storage Technologies, pages 43-58, March 2003.

[24] McDonald, A. D., and Kuhn, M. G., StegFS: A Steganographic File System for Linux. In Information Hiding, pages 462-477, 1999.

[25] Dowdeswell, R. and J. Ioannidis. The CryptoGraphic Disk Driver. In Proceedings of the Annual USENIX Technical Conference, FREENIX Track, June 2003.

[26] Howard, M. G., M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham and M. West, Scale and performance in a distributed file system, ACM TOCS 6 (1), February 1988.

[27] Shepler, S., B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler and D. Noveck. NFS Version 4 Protocol. RFC 3010, December 2000.

[28] Imielinski, T. and B.R.Badrinath, "Moblie Wireless Computing: Challenges in Data Management," Communications of the ACM, vol.37, October 1994.

[29] Truecrypt, http://truecrypt.sourceforge.net, 2005

[30] RSA Laboratories, PKCS ♯5 v2.0: Password-Based Cryptography Standard, RSA Data Security, Inc. Public-Key Cryptography Standards (PKCS), ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf, March 25, 1999.

[31] Iozone, http://www.iozone.org, 2005