Group Sharing and Random Access in Cryptographic Storage File Systems

by

Kevin E. Fu

B.S. Computer Science and Engineering
MIT, 1998

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

Author .....
Department of Electrical Engineering and Computer Science
May 18, 1999

Certified by .....
Ronald L. Rivest
E. S. Webster Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Group Sharing and Random Access in Cryptographic Storage File Systems

by

Kevin E. Fu

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 1999 in Partial Fulfillment of the Requirements for the Degree
of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Traditional cryptographic storage uses encryption to ensure confidentiality of file data. However, encryption can prevent efficient random access to file data. Moreover, no cryptographic storage file system allows file sharing with similar semantics to UNIX group sharing. The Cryptographic Storage File System (Cepheus) provides confidentiality and integrity of data while enabling efficient random access and file sharing using mechanisms similar to UNIX groups. Cepheus uses a delayed-write-encryption policy for caching, delayed re-encryption for distributed re-encryption, and a hash tree structure beneath the inode for integrity. While maintaining confidentiality and integrity, the cost of reading a block is $O(1)$ amortized over a sequential read of the entire file of n blocks. Writes execute in worst-case $O(\log n)$ time.

Thesis Supervisor: Ronald L. Rivest
Title: E. S. Webster Professor of Electrical Engineering and Computer Science

Acknowledgments

Sivaramakrishnan Rajagopalan and Bill Aiello guided my work on cryptographic storage at Telcordia Technologies (formerly Bellcore) in Morristown, New Jersey. Ron Rivest endowed much advice as my on-campus thesis advisor. Their guidance and suggestions greatly improved the design of Cepheus. Telcordia Technologies supported my research with a graduate fellowship in the MIT VI-A program.

Portions of Cepheus code derive from David Mazières' Secure File System of the Parallel and Distributed Operating Systems Group at the MIT Laboratory for Computer Science[22]. Parts of the low-level file system originated from my 6.033 team's X-File System[12]. Drew Samnick implemented many of the Remote Procedure Calls for his Advanced Undergraduate Project. Other persons deserving credit for help or guidance include Derek Atkins, Giovanni Di Crescenzo, Neil Haller, Sam Hartman, Jeff Hu, Frans Kaashoek, Marcus Kuhn, Anna Lysyanskaya, and Jerome Saltzer. Finally, my fiancée, Teresa, deserves much gratitude for putting up with my midnight coding and bitter moods during this long ordeal. Just one more degree, I promise.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> Very frequently, a user of crypt will forget to remove a cleartext file after producing an encrypted version. Such cleartext can only be described as 'gold'[25]. –Robert H. Morris

Cryptographic storage file systems can protect long-term information from unauthorized disclosure and modification. This thesis proposes the Cryptographic Storage File System (CSFS – pronounced Cepheus), a file system to provide secure group sharing and efficient random access[1]. Cepheus expands upon existing cryptographic file system models by cryptographically enforcing traditional UNIX® file system semantics without imposing significant performance penalties[2].

This thesis consists of five chapters. Chapter 1 gives a general background on the UNIX file system, analyzes several existing cryptographic storage file systems, and motivates research in cryptographic storage. Chapter 2 outlines our requirements for Cepheus and states our assumptions of trust. Chapter 3 presents our design to satisfy the constraints of chapter 2. Chapter 4 describes the implementation details. Chapter 5 summarizes future directions and unresolved questions regarding cryptographic storage. Finally, chapter 6 concludes our results.

[1]Cepheus was the king of Ethiopia, husband of Cassiopeia, father of Andromeda, and fellow Argonaut with Jason. His name is also the only decent thing that sounds at all like Cepheus.

[2]UNIX is a registered trademark licensed exclusively through X/Open Company Ltd.

1.1 Background: UNIX File Systems

This section describes the basic ideas and terminology behind a simplified UNIX file system. We explain the file system details necessary to understand the rest of this document. For more precise definitions, see the Berkeley Fast File System paper[23]. If you are already familiar with file system internals, you can skip ahead to the next section.

To introduce the concepts in the UNIX file system, we begin with with a list of terminology. A file is the basic unit of storage in a file system. For instance, `/etc/passwd` is a name which refers to a file containing account information as data. A *file system* consists of software to store and retrieve files.

A *partition* refers to the stored data referenced by a file system. It typically resides on a disk consisting of rotating magnetic platters. Partitions are also known as disk partitions or volumes. Partitions divide into many consecutive *blocks* typically of a fixed size (e.g., 4KB). The block I/O interface contains methods to manipulate a partition down to the granularity of a block. That is, one cannot write a single byte to a partition. Rather, one must rewrite the entire block that contains the changed byte. Regions of blocks fall into several structures: a superblock, a freemap, an inode table, and data blocks. Figure 1-1 shows the arrangement of structures on a partition. We explain each structure below.

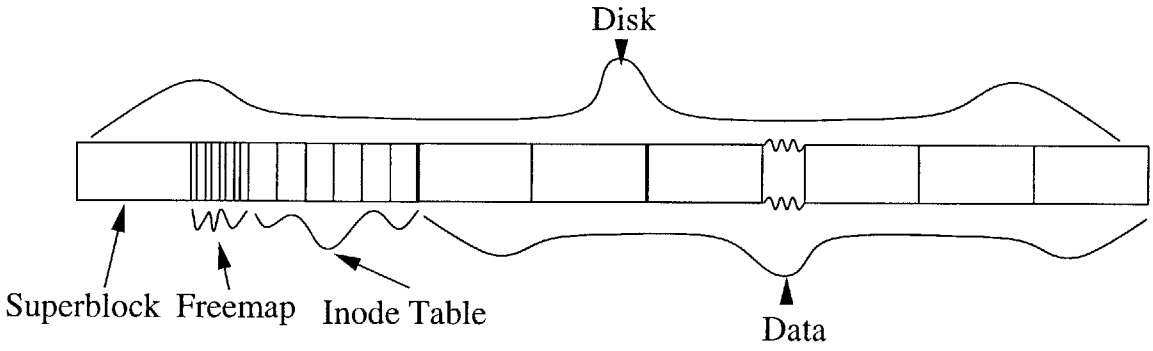

Figure 1-1: The layout of data and metadata on the physical disk.

Superblock

The *superblock* acts as a road map to the rest of the disk partition. It keeps track of critical information necessary to mount the file system. For instance, you might find the following in a superblock:

- magic number
- number of blocks
- location of freemap
- location of inode table
- number of free inodes
- location of data blocks
- last time of a file system consistency check
- number of mounts since last consistency check
- dirty bit

The information in the superblock is established during initialization of the partition. Should the partition fall into an inconsistent state (e.g., from a power failure), the superblock can help to reconstruct the disk partition.

The superblock also records a magic number which can help to determine whether or not the data are valid. If this magic number is not intact, we know something disturbed the superblock. It is also useful to check whether values in the superblock make sense. For example, if the location of the inode table is past the end of the disk, we can assume that corruption has occurred and the superblock data should not be used.

The mount count and the dirty bit signal mandatory consistency checks. The mount count keeps track of how many times we have remounted the partition. Typically there is a threshold number of mounts before a mandatory consistency check takes place. We set the dirty bit during every mount operation. During a shutdown, the file system clears the dirty bit to denote a cleanly unmounted partition. If the file system finds a set dirty bit when mounting a partition, it knows that the file system was shut down unexpectedly. An investigation would proceed to check the

consistency of the partition. The `fsck` program attempts to recover data which may have been corrupted by partially completed operations.

Freemap

The *freemap* follows the superblock. It contains a simple table with one bit representing each block of the disk partition. Bit i in the freemap indicates whether block i is currently free. In this way, the file system can readily find free blocks without having to search the entire disk. There are more scalable ways of keeping track of free blocks. For instance, XFS uses a B+ tree to keep track of extents of contiguous data[40].

Inode Table

Storage in a file system falls into two categories: *data* and *metadata*. Data make up the essential cargo of a file system. The whole point of a file system is to store data in an easily retrievable fashion. Metadata consist of the bookkeeping necessary to maintain the file system. For instance, a file in the UNIX file system is represented by an *inode* which contains metadata necessary to locate the contents of the file. An inode stores the attributes and pointers to data of one file. The *inode table* consists of a sequence of blocks which contain all the inodes. Since inodes are often statically allocated during initialization, the number of inodes corresponds to the maximum number of files that can exist. Allowing a large number of inodes provides for the storage of many files, but requires more space for the inode table. However, modern file systems allow some flexibility with dynamically allocated inodes. An inode often has the following attributes:

- index number
- access rights
- reference count
- file type
- generation number for NFS file handles

- most recent access time
- most recent modification time
- creation time
- owner
- group
- file size

The access rights define read, write, and execute permissions for the owner, group, and others. The reference count keeps track of the number of directory entries pointing to an inode. When this number reaches zero, the file system can release the inode and data. The file type denotes how to interpret the data of the file. Common file types include: regular file, directory, and symbolic link. The generation number becomes useful when using network file systems. It helps to prevent client workstations from accidentally overwriting changes made by another client.

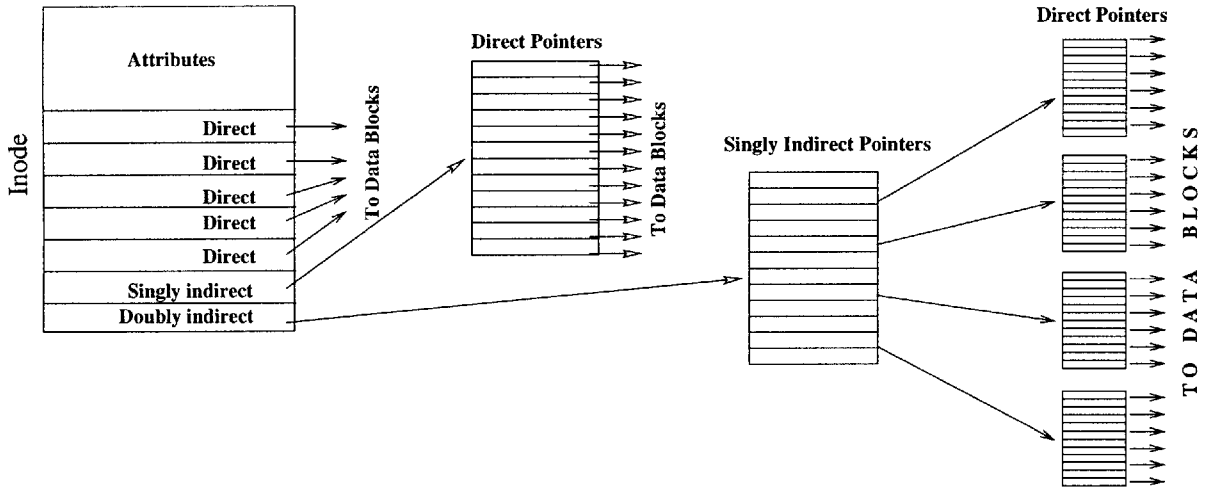

Figure 1-2: The inode contains file attributes and pointers to data blocks. Singly and doubly indirect blocks allow a small inode to address a large amount of file data.

In addition to keeping attributes, an inode maintains pointers to data blocks as shown in figure 1-2. Data pointers are either direct, singly indirect, or doubly indirect. Indirect pointers allow inodes to remain relatively small[3]. A singly indirect pointer references a data block which contains many direct pointers. Doubly indirect

[3]Because common operations such as `ls` read several inodes at a time without needing data pointers, the inodes need to be small.

pointers work similarly. Hence, an inode can address large amounts of data while directly housing only a small number of pointers.

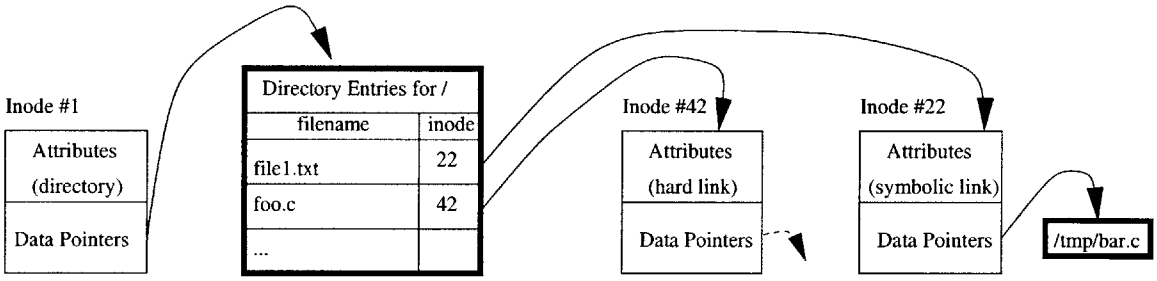

Figure 1-3: A complex example of a directory mapping file names to inodes. Thick boxes represent data blocks while thin boxes represent inodes.

Directories

Note that an inode has no location for a file name. Rather, directories keep track of the file name to inode mapping. In figure 1-3, inode #1 represents the root directory. In the context of this directory, `file1.txt` and `foo.c` refer to inodes #22 and #42 respectively. `foo.c` represents a typical *hard link* to an inode. A hard link points directly to an inode which in turn points to the file data. A second kind of link, a *symbolic link*, involves an extra level of indirection. The associated file data actually reference a path to another file name as in the case of inode #22. The path `/file1.txt` resolves to the path `/tmp/bar.c`. Note that a symbolic link may point to any file path, whereas a hard link can only point to inodes within the same partition.

1.2 Motivation

We motivate the use of cryptographic storage through several examples and trends. Cryptographic file systems offer several security advantages over existing information protection methods. For instance, manual file encryption is cumbersome. One may encrypt a file and then decrypt whenever necessary. However, this clearly offers no transparency to the user. Moreover, manual file encryption protects only what the user knows about. Manual file encryption cannot protect temporary files or swapped

out memory. Building cryptography into a file system allows transparent operation and helps prevent such accidental disclosure.

Reportedly 20,000 credit card numbers were stolen from Netcom, a leading Internet service provider[21]. Had a cryptographic file system been used to store the information, the incident may have been prevented. Tony Liss and Paul Tipton experienced information theft of their draft paper on the top quark[19]. After storing the draft in an obscure directory, another person at Fermilab managed to read the file. Shortly thereafter, some data were posted in a facetious report to an electronic bulletin board. Although the incident was minor, one learns that "security through obscurity" alone is not sufficient and existing file systems are ineffective against the insider threat.

Some researchers argue that a machine should cryptographically protect all of its resources (e.g., memory or the bus). However, today's general-purpose workstations cannot easily protect its resources against malicious local users. For instance, emacs run over an X connection would disclose the text of a document. There is cleartext Interprocess Communication (IPC) between the X server and X client. Moreover, research has yet to produce cryptographic protection of memory with tolerable access delays. For instance, the MemGuard program protects individual words in memory. The overhead ranges from several thousand to several hundred thousand percent of a normal memory access[10]. The benefit of securing every part of a client workstation is much less than gained from simply using a cryptographic file system and trusting the local machine.

According to Ross Anderson, insiders cause the majority of problems in the UK medical record system[1]. A recent ASIS report says that 75% of intellectual property theft involved an insider with a trusted relationship[26]. Marcus Ranum gives a high estimate that 80% of recorded security incidents are inside jobs[30]. While the exact percentage of insider influence varies greatly, one would agree that insiders are responsible for a significant portion, if not the majority, of privacy invasions. A cryptographic file system can transparently protect against many inside attacks. Bribing a system administrator would no longer be of use. Even if an adversary

managed to obtain encrypted files, the files are useless without the key. This separates the responsibility of storage management from information privacy[32].

Network traffic encryption can prevent outside attacks, but does little to prevent inside attacks. Traffic encryption does nothing to prevent a file system administrator from abusing privileges. On the other hand, cryptographic storage provides truly end-to-end encryption[33]. Cryptographically secure storage decreases the value gained from traffic encryption. Moreover, file servers become the bottlenecks in traffic encryption, especially when public key cryptography is involved. For every client workstation, the file server will have to perform encryption and decryption of data. In cryptographic storage, however, much of the computation moves to the client side, thereby removing bottlenecks from the server. The clients perform all the encryption and decryption. The file server must only check for authentication and authorization of requests.

Cryptographic storage can also provide for secure recovery of lost data. One could send in a hard drive for repair without worrying about leakage of information[32]. The data recovery can take place without compromising confidentiality. Another good reason to use a cryptographic file system is for protection against your computer or hard disk being lost or stolen. This is especially true for laptops or other mobile machines[9].

1.3 Focus

Most research in cryptographic file systems centers on fast and reliable storage for a single user. In addition to providing confidentiality, integrity, and availability, Cepheus focuses on group sharing and random access.

1.3.1 Group Sharing

Existing cryptographic file systems either do not address file sharing or discourage file sharing. We recognize that group sharing in the UNIX file system is rare, but allowing sharing adds tremendous power[38]. If sharing in the UNIX file system were

augmented with semantics similar to that of access control lists (ACLs) in AFS, we believe sharing would be more widely used. Currently, creation and membership maintenance in the UNIX file system and Sun's Network File System require intervention by a system administrator. Moreover, a user can be active in just one group at a time, and a file can have permissions for just one group. Cepheus allows secure sharing within cryptographic storage.

1.3.2 Random Access

Encryption adds an obvious overhead to reading and writing files. Cepheus exploits caches, delayed re-encryption, and encryption modes to diffuse the impact of encryption. Much of the motivation for efficient random access comes from research on incremental cryptography[3].

1.4 Related Work: Case Studies

A cryptographic file system does everything a traditional file system does, but in a secure manner. For instance, file data may be kept confidential or protected from unauthorized modification. Under reasonable cryptographic assumptions (e.g., it is computationally infeasible to decrypt a block without the key), we can prove properties of security.

Cryptographic file systems come in two varieties. *Cryptographic network* file systems protect the information sent between a user's workstation and the file server. For instance, Dave Mazières' Secure File System stores plaintext on the file server, but protects the link to the client[22]. Cryptographic network file systems are appropriate when the file server is trusted not to disclose or alter stored data. On the other hand, *cryptographic storage* file systems keep files encrypted on the file server. The users need not trust the file server to protect confidentiality. Below we focus our case studies on cryptographic storage file systems.

1.4.1 Cryptographic File System

The Cryptographic File System (CFS) introduced a relatively robust cryptographic storage system for the UNIX operating system[4]. Matt Blaze from AT&T Bell Laboratories developed CFS in 1993. Development continues, but not on a regular basis. CFS pushes confidentiality into data storage. Although several other file systems had already incorporated cryptography to secure network file transmissions, CFS is the first well-documented UNIX cryptographic storage file system. CFS investigates the question of where encryption should be placed in a file system: at the low-level hardware or the user-level. Several goals guided the CFS design including the issues of key management, transparency, and portability.

In a manual file encryption system, the user inputs a password or key whenever encryption or decryption takes place. Instead of requiring a password for each encryption operation, CFS asks the user once per login for a password. This password acts as a seed to compute a key stream. CFS eliminates the problem of re-entering passwords, but can generate new difficulties in password management. A user may need to remember several passwords to protect unrelated directories.

Transparent performance and access semantics allow users to perform necessary tasks without knowledge of or interference from the underlying cryptography. That is, one should not notice significant differences from a traditional UNIX file system. CFS succeeds in hiding most of the cryptography. When using one encryption password, CFS does not incur noticeable performance penalties after an initial pause to compute the key stream. Moreover, access semantics work as in a normal UNIX file system. However, simultaneous use of several passwords will cause many key streams to be loaded into memory. In the extreme case, CFS can cause thrashing by filling up all memory with key streams.

Because CFS runs in user mode, no kernel modification is necessary to use CFS (see figure 1-4). Moreover, the code has been ported to several UNIX flavors including SunOS and Linux. On the other hand, if CFS were implemented in the kernel itself, much context switching could be avoided. But an in-kernel implementation would

have prevented the ease of portability.

CFS provides end-to-end encryption from the client back to the client. All encryption operations take place on the client machine. The server is trusted to reliably store and retrieve information and not to alter storage. The user must completely trust the client machine and anyone who can gain root access on the client machine. For instance, the root user could access the password or key stream by searching `/dev/kmem`.

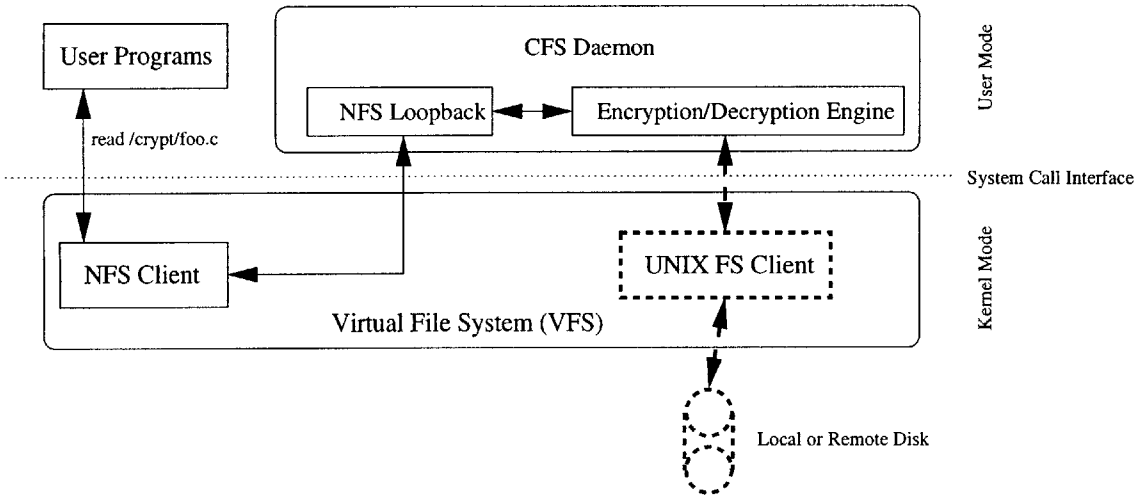

Figure 1-4: This diagram of CFS is based on Blaze's paper and an article on cryptographic file systems[4, 35]. Dashed lines represent a confidential area protected by encryption.

Directories serve as the atomic level of access protection. To reduce the number of key streams, CFS tags an entire directory with one key. That is, one specifies access permissions to files grouped together in a directory rather than to individual files. This differs from traditional UNIX file systems, but is similar to access semantics of the Andrew File System (AFS).

CFS protects data, file names, and symbolic links from disclosure outside the trusted computing base. However, no protection is given to metadata such as file sizes and time stamps. This allows existing utilities such as the `fsck` partition salvaging program to perform normally. CFS offers no protection against unauthorized modification. If a bit of the stored ciphertext flips, the user might notice a corrupted 64-byte block of plaintext. The goals of the CFS design do not include emergency

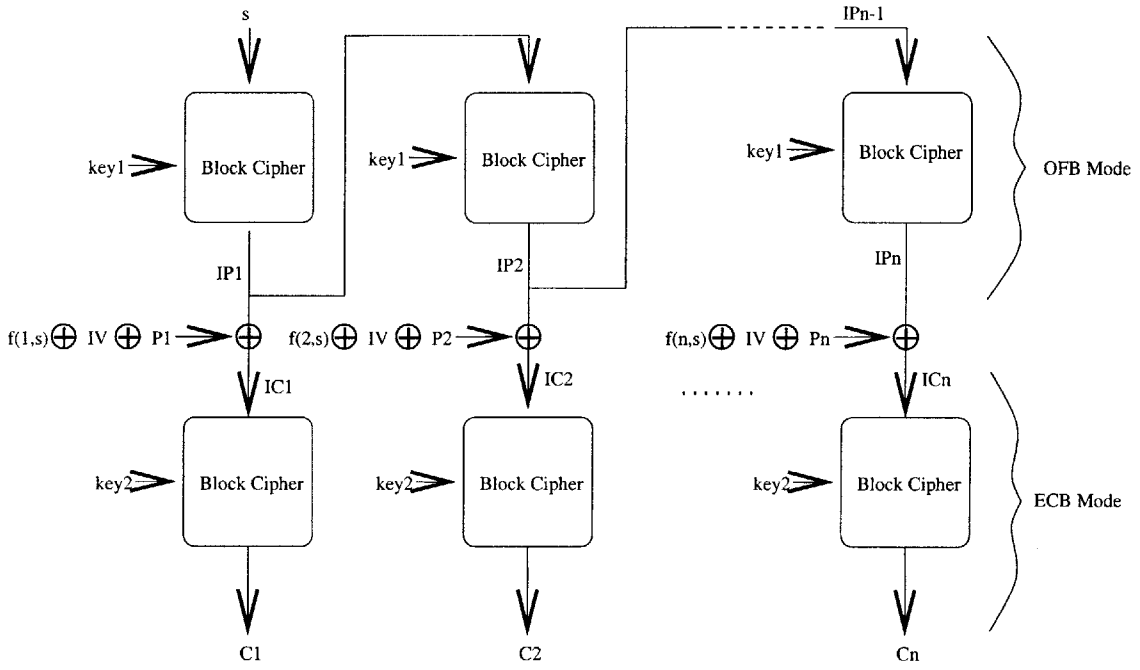

Figure 1-5: In Blaze's OFB+ECB mode, the OFB mode first computes intermediate plaintext blocks IP_i offline. When a plaintext block P_i needs encryption, it is first XORed with IP_i, IV, and a function of the block number and seed. This is encrypted in ECB mode for the final ciphertext.

key recovery or protection against denial of service.

Under the hood CFS employs the DES block cipher in Output-Feedback and Electronic Codebook (OFB+ECB) mode to create a key stream from a password. Figure 1-5 depicts this operation. All files in a directory use the same key, but each file has its own initialization vector (IV) tied to the inode number. CFS crunches the password into two 56-bit DES keys which determine the key stream in the OFB+ECB mode. By computing the key stream offline, CFS trades memory for speed. To my knowledge, no literature analyzes the OFB+ECB mode. However, Yoav Yerushalmi independently developed a similar OFB+ECB mode as shown in figure 1-6[43]. The OFB+ECB method is considered a natural solution.

For individual users, CFS is stable, easy to use, and transparent. However, some desirable goals are not addressed. The notion of file sharing does not exist. In order to share a directory, a user must disclose the directory key. In fact, the CFS documentation states, "The system is designed to be installed on individual single-

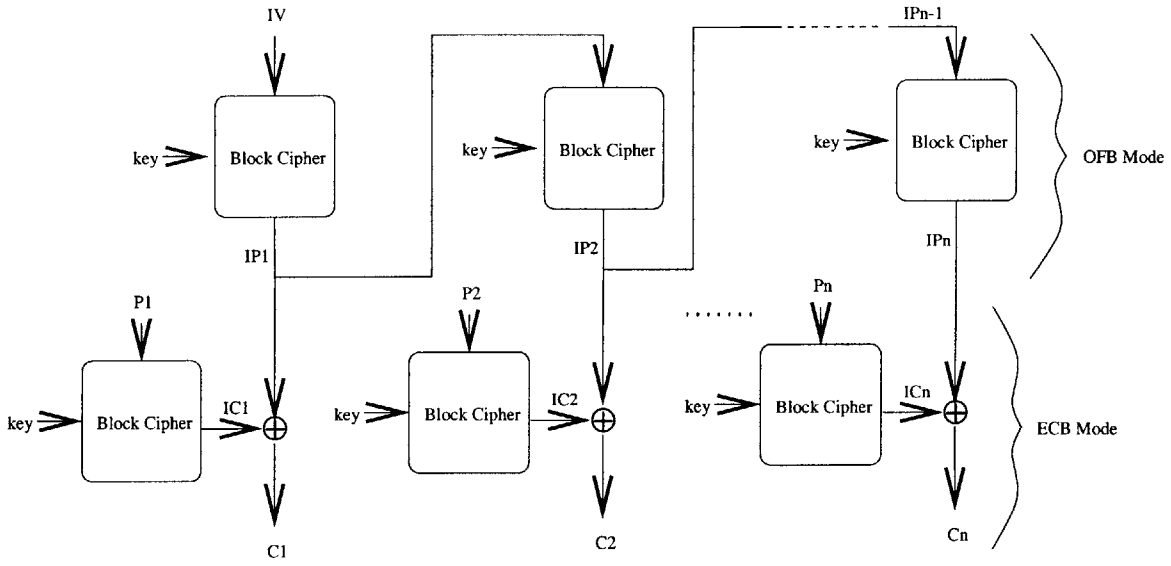

Figure 1-6: In Yerushalmi's OFB+ECB mode, the OFB mode first computes intermediate plaintext blocks IP_i offline, as is done in Blaze's mode. When a plaintext block P_i needs encryption, it is first encrypted in ECB mode, then XORed with IP_i. This allows for easier incremental changes.

user workstations. You really should not install it on a shared file or compute [sic] server, even though such a configuration is technically possible."

Because passwords directly encrypt files, CFS has the advantage that only the holder of the password can access files. No key ring waits to be stolen by an adversary. However, this makes re-encryption and emergency access difficult. No native password changing procedures exist. You would have to copy the appropriate files to a new directory, which would re-encrypt the files with a new password. In chapter 3, we explain a delayed re-encryption technique to avoid some of this latency. Emergency access or key escrow could help recover lost passwords, but it would also give adversaries an easier task to acquire a key (e.g., through bribery or the "rubber-hose" method).

Blaze explains several other important issues in two papers concerning his cryptographic file system[4, 5]. Anyone wishing to research cryptographic file systems should read these papers.

1.4.2 Secure FileSystem

The Secure FileSystem (SFS) implements a cryptographic storage file system for MS-DOS[4]. Despite its underlying operation system, SFS has a surprising number of creative and useful features. The primary goal of SFS is to protect bulk data stored on a disk[13]. Peter Gutmann, a graduate student at the University of Auckland in New Zealand, developed SFS until 1995. Although discussion about SFS appears occasionally in Usenet, all development has ceased. As Gutmann also worked with the early developers of PGP, a lot of paranoia rubbed off onto SFS. For instance, the SFS documentation explains that "35 separate overwrite passes" help to prevent the leakage of decrypted information. In order to be free of intellectual property problems and export restrictions, SFS avoids patented algorithms and was developed outside the United States. Gutmann does not release the source code because companies developing other DOS-based cryptographic file systems could copy it[5].

Since SFS operates entirely on a single machine, the user must only trust the local operating system and hardware. No concept of a "superuser" exists in the early versions of MS-DOS. SFS protects against disclosure of data to unauthorized persons who may gain physical access after files are encrypted. From the documentation, it is unclear whether SFS protects sensitive metadata or provides security against unauthorized modification. The protection granularity consists of an entire logical partition of data.

The most interesting feature of SFS is the emergency access mechanism. To safeguard against data loss, this mechanism can recover lost passwords. The emergency access mechanism employs Shamir's secret sharing scheme in which trusted escrow agents receive n key fragments[37]. Any m-sized subset of the n agents can recover the key. However, no smaller subset can feasibly recover the key. Unlike the big-brother-ish escrow mechanisms of the Clipper Chip, SFS does not give any agent the immediate ability to easily recover a key. Because m of the n pieces are needed to

[4]The Secure FileSystem by Gutmann and the Secure File System by Mazières are completely unrelated.

[5]Gutmann also blames any design problems on DOS.

harvest the master key, m users must collude to surreptitiously recover a key.

SFS uses the Cipher Feedback (CFB) mode of the Message Digest Cipher with the Secure Hash Standard (MDC/SHS) encryption algorithm, designed by Gutmann himself. The password generates an intermediate key by iterating a one-way hash function over the password several hundred times. This intermediate key can decrypt the master key which in turn decrypts/encrypts the data on the disk. Bruce Schneier comments that using a hash function for encryption is dangerous since the hash function designers did not intend for such use[36].

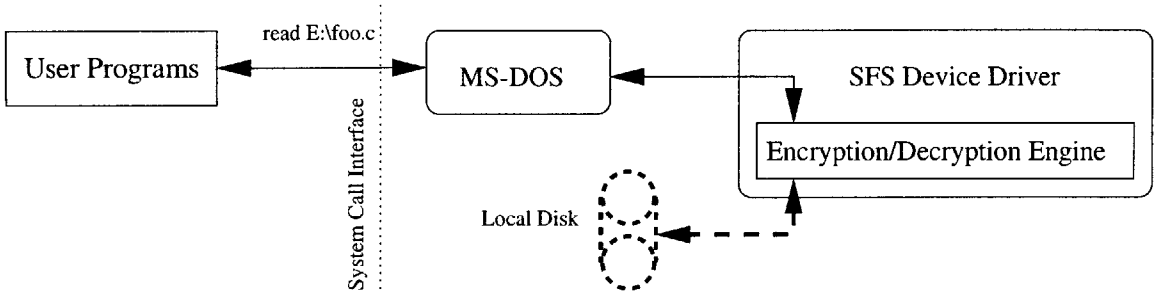

Figure 1-7: SFS interfaces to MS-DOS as a device driver with raw access to the physical disk. The thick dotted line represents confidential storage.

The SFS documentation boasts of moderate performance and excellent memory usage. On an average 486 desktop system, SFS requires as little as 7.5KB of RAM. Unlike CFS and TCFS (the next case study), SFS uses direct access to IDE and SCSI hard drives for better performance. Figure 1-7 depicts the model of operation. This probably hurt portability as SFS exists only for MS-DOS. However, Gutmann notes that the SFS design does not limit itself to any particular operating system.

SFS offers two timeout mechanisms to deter compromises. Passwords have finite lifetimes. When a password expires, SFS requires a new password to protect the master key. However, this does not protect against direct attacks to the master key encrypted with the old password. Changing a password just reduces the risk of compromising the currently encrypted master key. A second timeout can unmount the file system after a period of inactivity. Should a user absentmindedly walk away from the console, the file system will automatically unmount itself to prevent unauthorized access from the console.

In unimplemented design ideas, Gutmann explains an anti-duress measure such that adversaries cannot prove you know the key, and you can provide a reasonable proof that you do not know the key[14]. Gutmann's scheme is a form of secret sharing. Assuming you are dealing with a reasonable adversary, you can prevent the disclosure of your key while staying alive. More recent literature describes theoretical techniques for deniable encryption where a user can later disclose a fake key to reveal an alternative plaintext[6].

On the downside, SFS is not easily portable. With the advent of Windows95, some of the features of SFS require extra effort. Because MS-DOS is dreadfully simple, SFS does not need to worry about many of the security problems found in the UNIX operating system (e.g., multiple users or root access). Currently all development on SFS has ceased. Gutmann has instead pursued the development a cryptographic library called CryptLib. At least one commercial cryptographic storage file system uses this library[42]. Consult the SFS documentation for further information on its use and implementation[13].

1.4.3 Transparent Cryptographic File System

The Transparent Cryptographic File System (TCFS) seeks to improve upon the security model set by CFS[7]. Several students and faculty developed TCFS at the Universitá di Salerno in Italy during 1997. The early development was lead by G. Cattaneo, G. Persiano, A. Del Sorbo, A. Cozzolino, E. Mauriello, and R. Pisapia. New releases of TCFS appear on a regular basis. TCFS aims to offer the feel of the Network File System (NFS) ® without the feeling of insecurity[6]. TCFS works fundamentally the same as CFS in figure 1-4, except that TCFS is an in-kernel implementation.

One of the principle tenets of TCFS is not to trust the server except to store files. When a client machine attempts to access an encrypted file, the encrypted file blocks are sent over the network. Upon arrival, the blocks are decrypted on the client

[6]NFS is a trademark of Sun Microsystems, Inc.

machine. TCFS extends the UNIX file attributes to include an encryption bit. When this bit is set, TCFS protects the file with encryption.

As is with CFS, the metadata are not encrypted. Although some confidentiality may be lost to file sizes and file names, existing utilities such as the `fsck` disk recovery program will continue to operate normally. The documentation does not explain whether mechanisms exist to protect against unauthorized modification. Each user's standard login password decrypts a key ring to access files. Therefore, all files with the encryption bit set are encrypted by the owner's key. No built-in data loss protection exists. TCFS currently supports compile-time options for ciphers including DES, IDEA, and RC5. Keys are stored in a global key file which can be accessed by a user's password.

A recently released version of TCFS implements a group secret sharing protocol. A file can be opened only when a certain threshold number of group members log in simultaneously. All group members must log into the same workstation. This is not similar to the concept of group sharing in the UNIX file system.

TCFS has a strong following, but is somewhat less robust and well documented compared to CFS. For instance, a likely buffer overflow in an old key generation program could potentially expose root access. The program uses the `getpass` system call and is Set-User-ID (SUID) root. An SUID root program will run with root permissions, regardless of what user executed the program. Because TCFS requires changes to the `login` binary, it does not interoperate well with other authentication systems such as Kerberos or S/Key one-time passwords. On the other hand, TCFS is easily available because Italy currently has no export restrictions on cryptography. TCFS has regular releases and will likely become more robust as development continues.

1.4.4 SecureDrive

SecureDrive cryptographically protects entire partitions and disks in MS-DOS[39]. Edgar Swank and Mike Ingle worked on SecureDrive as a freeware project from 1993 until 1996. The model of SecureDrive matches that of SFS in figure 1-7. Unfortunately, the available design documentation for SecureDrive consists of a lot of source

code and an installation guide. SecureDrive hopes to encourage the use of encryption by making encryption easy to use. SecureDrive works on a limited basis in Windows95.

One interesting feature of SecureDrive is the key file. One can create a file of random bits in lieu of a password. This greatly increases the entropy to better thwart brute force attacks. The key file can be generated by any program. For instance, PGP 2.6.2 offers an undocumented feature to create cryptographically strong random bits. Swank explains two desirable features of a key file. First, the key file better represents the notion of a physical key. You can carry the key file on a disk. This also means that if you lose the disk, you lose your files. Or if someone finds your key file, that person could decrypt your files. Second, a key file allows anti-duress measures, popularly known as the anti "rubber hose" technique. A court order may require you to reveal your password. However, by erasing your key file, you yourself can no longer decrypt the files.

As with SFS, this file system enjoys the simplicity of MS-DOS. The user must trust the local operating system and hardware. SecureDrive protects against disclosure. It is unclear whether SecureDrive protects against unauthorized modification. The issue of denial of service does not apply since SecureDrive does not consider network usage. For data loss protection, SecureDrive suggests an out-of-band mechanism for emergency key recovery. If you use a key file, you can use any available escrow program to split your key amongst trusted third parties. This is completely left to the discretion of the user. For secure tape backups, SecureDrive has no native methods. But you can use a raw disk writing program to copy ranges of cylinders to a file, then backup the file.

The granularity of protection is the volume or disk partition. That is, an entire partition will share the same password and cryptographic key. Everything on the disk partition except the boot sector is encrypted. The documentation suggests creating multiple partitions if several users plan to use the same computer. However, SecureDrive is not designed to handle secure file sharing.

SecureDrive uses the IDEA cipher in CFB mode. The MD5 hash function converts

the user's password into a 128-bit IDEA key. If a key file is used, the key file is XORed with the key derived from the optional password. The disk serial number, track numbers, and sector numbers are used as part of the IV to make encryption of each sector unique.

SecureDrive requires minimal memory – the Terminate and Stay Resident (TSR) program consumes only 2.7KB of RAM. Since encryption takes place at the sector level, the encryption routines are completely transparent to the user. Changing passwords or key files can move slowly. An entire drive must be decrypted with the old key while encrypting with the new key. This may be more secure than storing a password-protected key file on the disk itself, but the process is slow. SecureDrive comes with source code.

1.4.5 SecureDevice

This freeware device driver for MS-DOS is a direct descendent of SecureDrive. Hence, its model is the same as in figure 1-7. SecureDevice uses the IDEA cipher to protect volumes of data[20]. Max Loewenthal and Arthur Helwig developed SecureDevice in the Netherlands in 1994. SecureDevice uses a single file as a virtual disk volume. This allows much more flexibility, but suffers performance drawbacks because of its location above another file system. SecureDevice uses the same trust model as SecureDrive and SFS.

Very little documentation explains the internals of SecureDevice. The unit of granularity is the virtual disk volume. In other words, one key protects everything. According to the source code, the CFB mode of IDEA protects the virtual disk volume. Encryption takes place on a per-sector basis. Each sector on a volume is encrypted separately with a different IV. To generate the IDEA key, SecureDevice takes the MD5 hash of the user's password. A different master IV is used for each volume created. SecureDevice then mixes the master IV with the 32-bit sector number to produce a unique IV for each sector. As with SecureDrive, changing a key suffers performance drawbacks. Since the password is directly used for encryption, an entire volume must go through a lengthy decryption and encryption routine.

Because SecureDevice stores the encrypted disk volume as a single file, backups are extremely simple. Just copy the file to a remote device such as a tape drive. However, fragmentation of a large volume can lead to instability. If a file is fragmented into more than 50 pieces on the disk, DOS will become unstable. A disk defragmentation program is necessary to correct this problem.

1.4.6 CryptDisk

CryptDisk ensures the confidentiality of entire partitions on the Macintosh[29]. This is a shareware product written in 1995 by Will Price, then a graduate student at the University of Southern California. CryptDisk has since become a commercial product called PGPdisk[28].

As in SecureDevice, a special container file acts as a virtual disk. To mount an encrypted partition, one simply drags the icon of the container file to the icon of the CryptDisk application. One can then access individual files of the encrypted partition – as if they were regular files. As with SFS, CryptDisk will unmount all volumes after an inactivity timeout. An emergency recovery option exists to decrypt an entire volume should it ever have problems mounting. The drag-and-drop nature of the Macintosh makes the file system almost completely transparent to the user.

The trusted computing base is the same as the DOS-based cryptographic file systems previously discussed. Since the Macintosh operating system does not have the concepts of multiple users or a root account, CryptDisk must only trust the local operating system and hardware. The model is virtually the same as in figure 1-7.

CryptDisk protects files using the IDEA encryption algorithm in CFB mode with an IV that varies every 512 bytes. CryptDisk uses a master key derived from a function of a password. The password may be up to 128 characters in length, and is hashed an arbitrary number of times and salted with random data to insure a good key. Session keys protect individual files. A file's session key derives from the master key and a publicly known, random salt value. The salt varies for each file. Because CryptDisk uses a master key to obtain file keys, it is easy to change the password.

Price warns that buffers in memory could be swapped to disk by virtual memory.

In other words, an IDEA key may be inadvertently written to disk, allowing anyone at the console to grab the key. Bruce Schneier points out that Norton Diskreet suffered from this problem as well[36]. Peter Gutmann discusses such swap file issues in archives of the sfs-crypt mailing list[13].

1.5 Common Features

This section examines the common features in cryptographic storage file systems. Table 1.1 summarizes differences among the case studies. In this section, "the file systems" refers to all of the case studies.

File System	Granularity of Protection	Operating System	Unique Feature
CFS	Directory	SunOS, Linux	Networkable
SFS	Partition	MS-DOS	Emergency Recovery
TCFS	User Account	Linux	Threshold Sharing
SecureDrive	Partition	MS-DOS	Key Files
SecureDevice	Partition	MS-DOS	Easy Backup
CryptDisk	Partition	MacOS	Drag-and-Drop

Table 1.1: A comparison of cryptographic storage file systems.

1.5.1 Key Management & Sharing

Most of the file systems leave key management up to the user. That is, sharing involves disclosing your *personal* password. If a cryptographic file system promotes sharing, it should have a transparent mechanism to obtain keys when necessary, rather than forcing the user to juggle many passwords. Furthermore, to integrate a cryptographic file system with existing authentication systems such as Kerberos or SecureID, a design must not mandate a specific method for user authentication.

1.5.2 Portability

Based on the amount of online discussion, CFS is by far the most widely used cryptographic file system. One of the driving forces behind CFS is portability. Because

CFS uses an NFS loopback server rather than the VFS/Vnode interface of the kernel, CFS quickly ports to other UNIX systems. However, such portability comes at the cost of context switching between kernel and user mode. When CFS is used over a network, files must travel through several NFS servers.

1.5.3 Transparency

General users will not welcome a cryptographic file system unless the internal cryptographic routines are transparent. Fortunately, most of the file systems have a good level of transparency. Changes in access semantics and performance are not usually noticeable for small files. Traditional file system utilities such as `fsck` would otherwise require significant modification. By making the cryptography transparent, users can perform their tasks unhindered, and file system utilities can operate without modification.

1.5.4 Encryption Costs

Since encryption can add significant delays for file I/O, cryptographic file systems try to reduce the amount of encryption and decryption. For instance, CFS uses a novel method of encryption in the OFB+ECB mode[4]. This mode allows pre-computation of part of the key stream. When a block requires encryption, the file system must simply XOR a pre-computed key stream and encrypt a block.

1.5.5 Fine-grained Access Control

Cryptographic file systems must choose an appropriate level of protection granularity. By grouping files together protected by one key, significantly fewer cycles are spent creating key streams. However, the grouping also increases potential damage from a lost or stolen key. The examined file systems have granularity ranging from directories to partitions.

1.5.6 Key Changes & Revocation

Changing a key or revoking group membership should not consume an excessive amount of CPU time. Users demand that normal operations run in a reasonable amount of time. For instance, TCFS uses a login password to decrypt a key ring. If the login password changes, one need only re-encrypt the key ring. The files protected by the keys in the key ring do not necessarily require re-encryption. By using indirection, it is possible to minimize the cascading effects of a key change or membership revocation. However, this may result in a penalty to security. An adversary could mount an attack on the system by using old key rings. Therefore, it is not appropriate to use simple key indirection if an adversary steals a key ring.

1.5.7 Intrusion Tolerance

Cryptographic file systems must tolerate some level of intrusion. For instance, a user should not be able to use his or her password to decrypt a global password file for every user. File systems should also tolerate collusion amongst system administrators[11]. Often secret sharing schemes can drive the probability of compromise to near zero.

1.5.8 Reliability

Escrow is a touchy issue. But some form of key recovery is necessary since users often forget passwords. This may not be important for temporal data such as email, but it is important for long-term storage. A lost key implies lost data. Escrow or highly reliable backups forgo some security for the persistence of data.

1.5.9 Trust in System Administrators

One feature of a cryptographic file system is that the user needs not trust the system administrator to keep files confidential. However, the system administrator should perform his or her job of maintaining hardware and performing backups. In all of the case studies, the file systems trust the system administrator to maintain file integrity and keep files available.

1.5.10 Integrity

Most cryptographic file systems provide confidentiality, but not integrity[17]. That is, an adversary could undetectably change bits on the disk. The legitimate user might only notice a corrupted block of plaintext. The file systems do not detect unauthorized modification. A user might notice modifications when garbled files appear. In networked file systems, some mechanism to detect tampering is necessary. MACs or digital signatures may help, but re-signing files continuously will have serious performance drawbacks.

Chapter 2

Design Requirements

Here we list the necessary properties and features of our cryptographic storage file system. First, we state the general criteria of Cepheus. Next, we classify failures as tolerable or intolerable. Finally, a trust model is developed and justified. In the next chapter, we propose a design to meet the criteria.

2.1 Design Criteria

Cepheus protects the confidentiality, integrity, and availability of storage. The following criteria guided the design of Cepheus:

- Ensure the *confidentiality* of file data and directory contents
- Maintain *integrity* of file data, directory contents, and metadata
- Provide for *availability* of file data, directory contents, and metadata
- Facilitate simple *group sharing* with similar semantics to the UNIX file system
- Allow efficient *random access* to file data, directory contents, and metadata

2.1.1 Confidentiality

Cepheus must protect file data and directory contents from unauthorized access. This includes information such as file names and data blocks, but not metadata such as

modification times, data block pointers, and other `stat` information found in the inode. Only users explicitly granted access should be able to read the plaintext contents. For instance, a system administrator does not implicitly have access to plaintext. This separates the privilege of storage management from that of confidentiality[32].

We do not protect confidentiality of metadata in order to enable better crash recovery. For instance, if data block pointers were encrypted, the file server could not piece together a partially written file. While this may allow for limited traffic analysis, we consider the ability to recover from failures more important.

2.1.2 Integrity

Cepheus must detect damage to integrity of file data, directory contents, and metadata. However, this does not require *prevention* against unauthorized alteration. We explain prevention in the next paragraph. For integrity we merely stipulate unauthorized alterations be *detectable*. This allows users to know with certainty that a file is what it claims to be. An adversary cannot add to file contents or change metadata without being detected.

2.1.3 Availability

We explain the difference between integrity and availability in subsection 2.3.4. Because integrity can only detect unauthorized alterations, we further require prevention of unauthorized alterations. For instance, if an unauthorized user deleted all the files, the requirements of subsection 2.1.2 still hold. The deletions are detectable. File data, directory contents, and metadata must be readily available and not subject to simple denial of service attacks.

2.1.4 Secure Group Sharing

Cepheus must allow for group sharing with similar semantics to those of the UNIX file system. However, the permissions must be cryptographically enforced. As an example, consider a file with group read access, but no write access. A group member

should not be able to create authentic writes even though the group member can decrypt and read file contents. Only a group owner should have the ability to add new members to the group.

2.1.5 Efficient Random Access

Cepheus file reads and writes must perform similarly to that of a traditional file system. For instance, reading the last or first byte of a file should take approximately the same time. Moreover, encryption should be used conservatively by minimizing online cryptographic operations. Operating on a small file should execute quickly. Operating on a large file should perform efficiently.

2.2 Failure Conditions

All systems are subject to uncontrollable failures. However, tolerable failures should be recoverable. Below we classify failures as tolerable or intolerable. For tolerable events, we discuss necessary properties of recovery.

2.2.1 Intolerable Failures

We expect all hardware to operate correctly or to recover on its own. Cepheus does not attempt to recover from catastrophic failures beneath the block I/O level. For instance, the underlying media (i.e., the hard drive) should ensure reliable storage of information. Strategies such as the Redundant Array of Inexpensive Disks (RAID) can make the likelihood of unrecoverable failure extremely low.

2.2.2 Tolerable Failures

Several events can cause a file system to enter an inconsistent state. In such tolerable failures, Cepheus must recover to a consistent state. In a typical UNIX file system, an unexpected power failure could leave a partially written file. This may cause the inode table to be inconsistent with the freemap, for example. Also, writes to the

cache may not have been fully flushed to disk, leading to lost file data. Kernel panics and other software crashes are treated just like power failures since the end result is the same. Cepheus must also recover from incomplete operations resulting from client crashes, file server crashes, and group database server crashes. The next section describes these modules in more depth.

Since Cepheus introduces the idea of cryptographic integrity built into a file system, it must gracefully recover from integrity check failures. If a file contains unauthorized modifications, the file system should return an error and refuse to serve the file. The user agent can recover damaged files through reconciliation with the user. Similar constraints apply when a key is lost or decryption fails.

2.3 Cryptographic Storage Trust Model

This section defines and defends a trust model for Cepheus. We incrementally build up our trust model, ensuring that Cepheus stays as secure as conventional file systems and wherever possible more secure. Recall that Cepheus has five criteria for storage: confidentiality, integrity, availability, group sharing, and random access. Random access to file data is independent of the principals in our trust model. This issue appears in the next chapter. We first define the principals in our model, then begin with a trust model for confidentiality.

2.3.1 Trust Model Principals

Below are the principals in our trust model, as depicted by figure 2-1.

- A *client machine* is a multi-user workstation. A client daemon and one or more user agents run on this machine.
- A *client daemon* communicates with user agents and file servers.
- A *user agent* acts on behalf of the user and retrieves file and group keys. It responds to requests from the client daemon and communicates with the group database server. Each user has a user agent.

- A *file server* stores and retrieves files for client daemons. It also communicates with the group database server.
- A *group database server* maintains group membership information, user public keys, and group symmetric keys. It responds to requests from file servers and user agents.
- A *network* connects client machines, file servers, and the group database server. The network is publicly accessible.
- A *group* consists of a list of users which share a common privilege for file access.

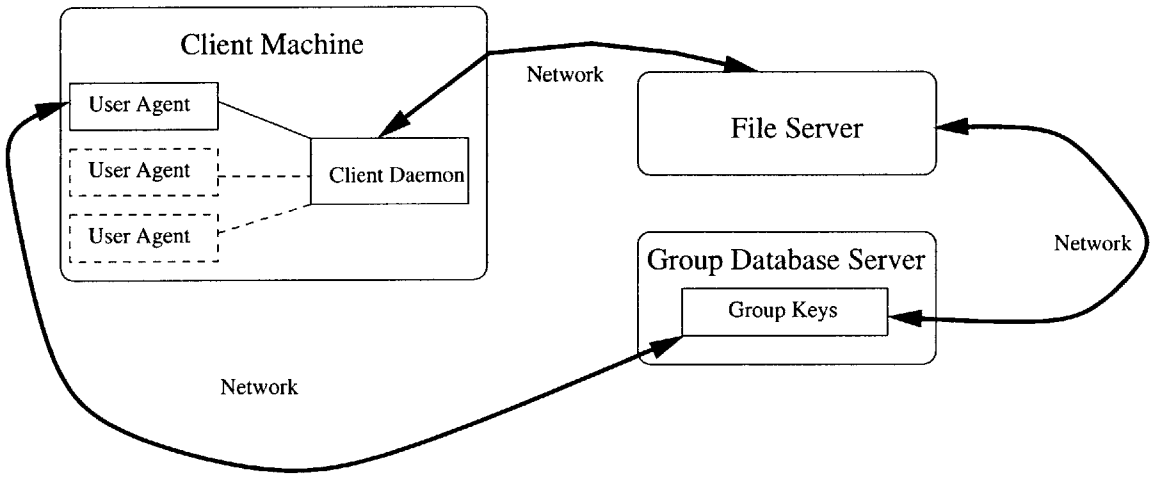

Figure 2-1: Interactions among the trust model principals in Cepheus.

2.3.2 Confidentiality

Files remain confidential if and only if users with explicit permission can understand the stored file data. For instance, a file server administrator should not implicitly be able to decrypt the files; encrypted files are opaque to the server. Client-side encryption can easily provide for confidentiality. A simple trust model for the confidentiality of file data and directory contents is as follows:

1. A *file server* maintains the reliable storage of files.

2. A *user agent* is completely trusted by the user. It performs decryption and encryption at the request of the client daemon. It trusts that client daemon requests are authentic and that the client daemon will not disclose plaintext to unauthorized parties.

3. A *client machine* will not snoop in the buffer cache or keys of the user agents. It is either a single-user machine, or the users trust all persons who effectively have root access.

4. A *client daemon* trusts that file system requests from the kernel are authentic. It maintains a buffer cache of plaintext and ciphertext for each user, and asks user agents to perform decryption and encryption.

5. The principals trust the *network* only for reliable packet delivery.

Clearly, this model is more secure than that of a conventional file system. Only the user agent has access to plaintext. This is the model used by most conventional cryptographic storage file systems.

2.3.3 Integrity

To provide for integrity, it must be possible to verify that no unauthorized person has modified the file data, directory contents, or metadata. For instance, if the file server or network were to flip a bit of the encrypted file, the user agent should detect the alteration. Cryptographic hashes, message authentication codes, or digital signatures can provide for integrity. In addition to the trust model for confidentiality, we require:

6. At the request of the client daemon, *user agents* create and verify an integrity field that is not easily forged.

7. The *file server* and *group database server* will not collude with users to obtain unauthorized privileges.

8. The *client daemon* will not request a user agent to create an integrity field for bogus data.

Since only the user agent can create a valid integrity field, the user agent can detect unauthorized changes. Other users and the file server cannot feasibly construct an authentic integrity field on their own. But a user with read access could collude with the file server to gain authenticated write access. For this reason, our trust model forbids such collusion. It is possible to omit this requirement with digital signatures, but we have not discovered an efficient mechanism to do so. Note that these requirements do not hurt or improve confidentiality. Moreover, we still maintain a more secure system than a conventional file system.

2.3.4 Availability

In network link security, one works to secure a link between two end points. If an integrity check fails, the client can simply ask for a retransmission. But in cryptographic storage, the client is both the starting and ending point. There is no opportunity for retransmission if an encrypted file is lost. Consequently, cryptographic storage must rely on *prevention* rather than *detection* to preserve integrity. This leads to our third goal, availability.

Denial of service attacks are notoriously difficult to prevent. For instance, a malicious user could consume all the resources of the file server by making bogus requests. Such an attack reduces the availability of a service. The next paragraphs explain what is necessary to achieve reasonable availability.

Cryptographic storage can separate the responsibility of storage management from that of confidentiality[32]. Unfortunately, we cannot entirely separate storage management from availability. Consider two trust models of a server administrator: the first model trusts the server administrator merely for reliable storage. The server acts on requests to read and write files. A second model additionally trusts the server administrator to verify authorization of user requests.

In both models, the client can maintain confidentiality and detect failures of integrity. But availability requires cooperation with the server. If the file server performs no authorization, a benevolent server cannot distinguish authorized users from unauthorized users. Hence, the server cannot prevent unauthorized reads and writes.

If the server performs authorization, we can prevent unauthorized requests to modify files and metadata. In the worst case (a malicious server), the trust models result in the same availability; a malicious server can ignore authorization. But in the expected case (a benevolent server), the second model can prevent unauthorized requests. Consequently, unauthorized users cannot simply remove files. Moreover, we can improve confidentiality. Unauthorized users must exert effort to break the authorization mechanism before they can attack the confidentiality. In addition to the trust model for confidentiality and integrity, we require:

9. The *file server* reliably stores files and verifies authenticity and authorization of requests by consulting with the group database server. The file server believes in the authenticity of user information from the group database server. There is an integrity-protected link to the group database server.

10. The *group database server* responds correctly to authorization requests and maintains authentication and authorization information of users.

If the encryption keys are not derivable from the authentication process, confidentiality, integrity, and availability remain independent of each other.

2.3.5 Group Sharing

With group sharing, a single key allows a group member to access files assigned to the group. Unfortunately, giving away a group key also gives away the ability to add new group members. However, the file server can prevent this unauthorized spread of privileges. The file server verifies that a user belongs to a group by consulting with the group database server. Our trust model additionally requires:

11. The *file server* is trusted not to obey requests for which the group database server denies access. The file server believes in the authenticity of group membership information from the group database server.

12. The *group database server* keeps up-to-date membership lists and distributes group keys.

13. Only *user agents* with group write access can create authentic integrity fields.

14. *Users* are trusted to be responsible and not disclose private information through covert channels.

We now have all the features of a conventional network file system, but we have provided for each in a secure manner. We note that for strict confidentiality with group sharing, we must include item 14 in the trust model. Otherwise group sharing could affect confidentiality of files. A user could simply redistribute plaintext in an out-of-band channel. Furthermore, a group member with read access could collude with a file server to create valid integrity fields.

Our final trust model enables confidentiality, integrity, and availability in a network file system with group sharing. We were not able to tighten the trust model any further without losing confidentiality, integrity, or availability. With this model in mind, we can now design Cepheus in the next chapter.

Chapter 3

Cepheus Design

We first discuss fundamental concepts for group sharing and random access. Then we describe the four modules of Cepheus: the user agent, the client daemon, the file server daemon, and the group database server. Figure 3-1 shows how the modules interact. We rationalize how the design of Cepheus complies with the requirements given in chapter 2. Finally, we summarize design alternatives.

3.1 Group Sharing

Group sharing consists of two ideas: maintaining group membership lists and maintaining group access rights. A group database server facilitates for the former while the file server and file structures take care of latter. Section 3.3 explains the details of the file structures. We discuss the group database server in section 3.7.

Much of the group sharing depends on the concept of a *lock box*. We use this term in reference to a key encrypted with another key. The lock box metaphor corresponds to the lock box used by real estate agents. A realtor can attach a small lock box to the door of a house for sale. Within the box is the key to the house. Anyone who knows the combination to the lock box can indirectly obtain the house key. In this way, the lock box can exist out in the public, but only authorized persons can open the lock box to reveal the protected key. Lock boxes are similar to master key systems[8].

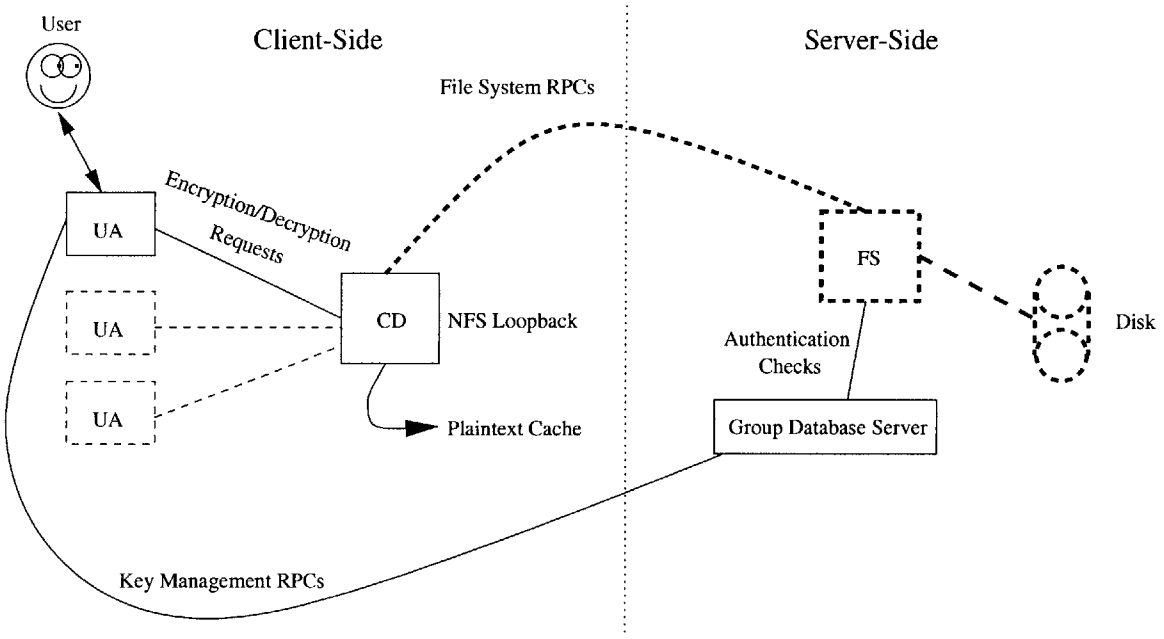

Figure 3-1: The client daemon (CD) acts as an NFS loopback server on the client workstation. The CD asks the appropriate user agent (UA) to encrypt, decrypt, sign, or verify data. Each CD communicates with the appropriate file server (FS). The FS checks with the group database server (GDS) for authentication and authorization of a user agent. The thick dotted line represents confidential storage.

3.2 Random Access

In a traditional file system, the running time of a read or write request is mostly independent of the location of the data in the file. For instance, reading the last block of a 40MB file should take about as long as reading the first block of a 40MB file. Below we explain how Cepheus preserves this independence.

Cepheus encrypts each file data block separately. Figure 3-2 depicts the encryption of one file data block. Because a block cipher typically has an input size of 8 bytes, we must split the 8KB file data block into 1024 smaller blocks, $P_1 ... P_{1024}$. We use RC5 with the cipher block chaining (CBC) mode to encrypt these smaller blocks[31]. The CBC mode allows one to chain several encryptions together. Before encrypting a plaintext block, the mode first XORs the previous ciphertext with the current plaintext. When using an initialization vector (IV), this mixing action lets a single key securely encrypt several chained blocks. Since all blocks within a file use the same key, we use IVs to make sure similar plaintext blocks in a file do not encrypt to

similar ciphertext blocks. Each 8KB block in Cepheus has its own 8-byte IV, stored in the data pointer as shown in figure 3-3.

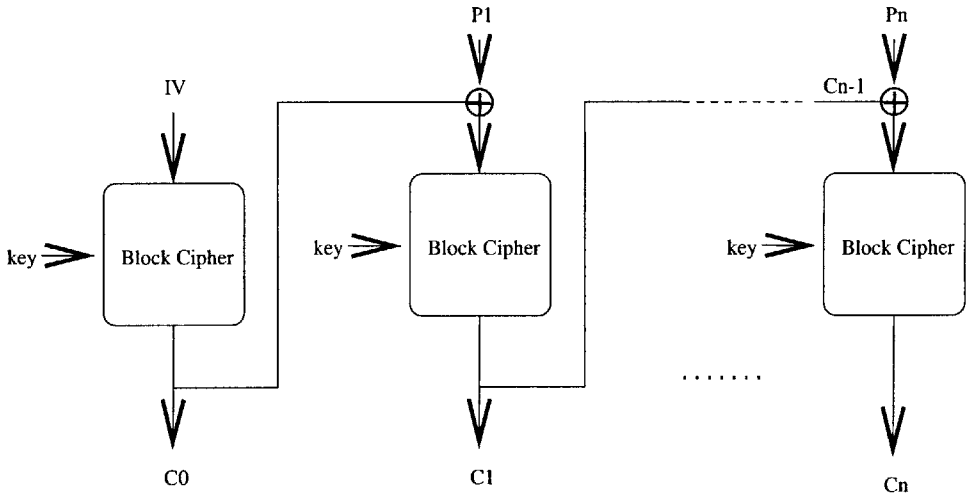

Figure 3-2: In Cepheus, file data is encrypted in CBC mode on a per block basis. Block ciphers typically have two 8-byte inputs.

Reading file data involves two operations: decrypting data blocks and verifying an authenticity/integrity check field (AICF). After Cepheus obtains the encrypted block, IV, and file key, the block can be decrypted. With the plaintext block and AICF, Cepheus can verify the integrity of the file data.

Similarly, writing file data involves encrypting data blocks and creating an AICF. However, the exact writing procedure, which we explain in subsection 3.3.2, is more involved. Whenever Cepheus writes a block, the IV changes as well. This prevents an adversary from analyzing the history of a file (e.g., what part of the file changed). We note that Blaze's CFS uses IVs for a different reason. CFS uses an IV to prevent similar blocks from appearing in *different files* because several files within a directory share the same key.

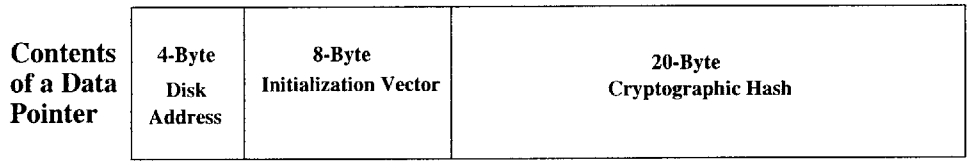

Figure 3-3: The data pointer contains an IV and hash node in addition to the pointer to the data block.

3.2.1 Delayed Re-encryption

When a group member departs from a group, the corresponding group key must also change. Moreover, the contents of lock boxes protected by that group key must change. In other words, all files associated with the group require re-encryption. Re-encrypting thousands of files at once would introduce significant delay. To avoid this delay, we relax the requirements of re-encryption due to group membership changes.

Re-encryption results from two basic causes: group reorganization and key compromises. For each cause, we recognize a satisfactory way to perform re-encryption[15]. In casual group reorganization, we can simply mark a file to be re-encrypted, putting off the re-encryption as long as possible. Such *delayed re-encryption* permits a former group member to read old cached data, but not new updates. AFS uses a similar model because little can be done to disallow the client from reading its own cache. On the other hand, a key compromise requires more immediate attention. *Expedited re-encryption* would quickly re-encrypt files in the case of serious emergencies. Since we expect most causes to have a benign nature (group reorganization), Cepheus uses delayed re-encryption by default.

To invoke delayed re-encryption, the owner first marks the cryptographic dirty bit of the file and sets up a new file key in a lock box. Then any group member with write access can later re-encrypt the file and clear the dirty bit. The file does not need re-encryption until someone makes a change to the file. Note that the entire file must be re-encrypted at once. We do not allow one portion of the file to remain encrypted in an old key. We can get away with this lazy behavior because of the same reason AFS does not prevent a client from reading its cache. We perceive one problematic issue with transparency. For example, delayed re-encryption may cause unexpected delay when a re-keyed file is opened in append mode. Cepheus would have to re-encrypt the whole file, not just the appended data.

3.2.2 Buffer Cache: Delayed Encryption

Almost all file systems use a buffer cache of recently read blocks. The cache allows fast access to blocks we expect to read again soon. With the addition of cryptography, a cache becomes even more useful. Cepheus uses a plaintext and ciphertext buffer cache on the client-side to absorb file writes and redundant encryption. A delayed write policy waits for a set period of time before writing the dirty blocks to the file server[41]. Since a newly-written block is often overwritten or deleted within a few minutes of its creation, the delayed write policy can absorb many unnecessary writes[27]. In a similar manner, our buffer cache delays encryption of newly-written blocks. Hence, a delayed encryption policy can significantly reduce the amount of encryption for file writes. On the other hand, delayed writes can make recoverability more complex. If a client machine crashes before flushing buffers to disk, files may become inconsistent.

3.3 File System Structures

Cepheus augments the metadata found in an inode. We add a few twists to the traditional UNIX file system for cryptographic storage. We did not intend to squeeze every cycle using techniques from the Berkeley Fast File System or XFS[23, 40]. However, no part of the design prohibits the use of advanced structures found in such file systems. Much of the structure is based on the X-File System[12].

All data on the disk is written in 8KB blocks. For instance, inodes are grouped together to fill a block. Ideally, structures should be a power of two to pack the blocks tightly. We use a large block size to reduce the overhead of encryption and to absorb several writes into one re-encryption operation.

3.3.1 Inodes

In addition to the standard fields, our inode contains the following attributes:

- owner lock box (file key locked for owner's use only)

- group lock box (file key locked for group's use only)
- owner old lock box (old file key locked for owner's use only)
- group old lock box (old file key locked for group's use only)
- authenticity/integrity check field
- group key ID
- file key ID
- old file key ID

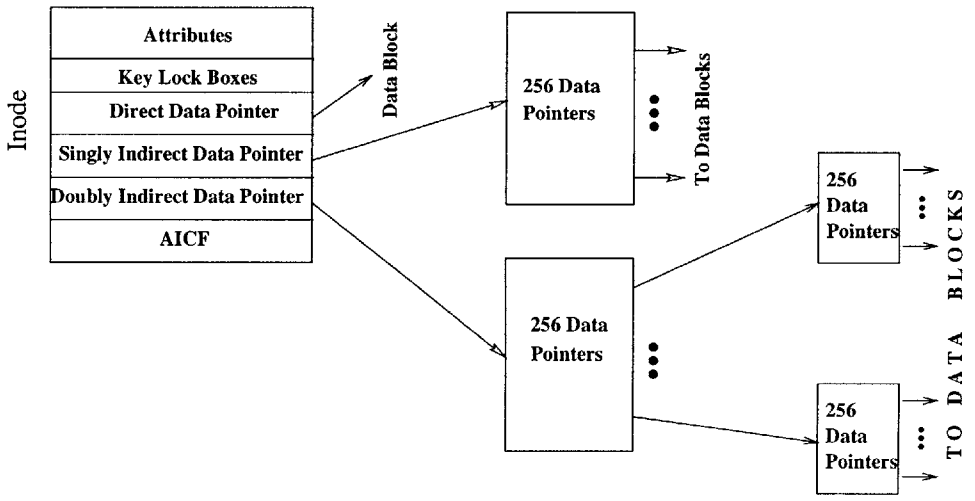

Figure 3-4: Using one direct, one singly indirect, and one doubly indirect data pointer, a single inode in Cepheus can address up to 514MB of data.

The lock boxes contain encrypted file keys. Only authorized users can obtain the key to open the lock boxes. We keep the previous version of the lock boxes for purposes of delayed re-encryption. A file owner updates the current lock boxes to re-key a file. If the old and current lock boxes differ, a user agent realizes the file requires re-encryption. Once the file is re-encrypted with the new key, the old lock box is set to the contents of the current lock box.

3.3.2 Authenticity/Integrity Check Field

Cepheus uses an authenticity/integrity check field (AICF) to verify the integrity of file data and metadata[17]. The slanted tree structure of the AICF parallels exactly the structure of the direct, singly indirect, and doubly indirect data pointers in figure 1-2. This leads to a natural structure that flows conveniently with existing file system

operations. For instance, the IV tends to be read immediately before reading its related data block. This significantly reduces the number of disk reads at the cost of a large branching factor.

Each data pointer contains a 20-byte hash of its data. Figure 3-5 denotes this 20-byte value with a circled H. At the top of the tree, the AICF is the keyed hash of its children.

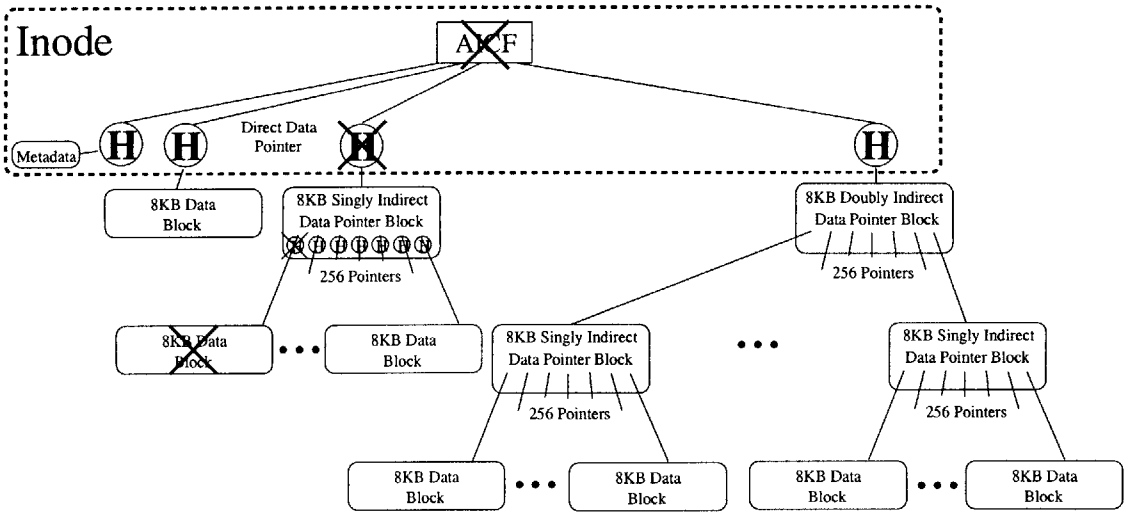

Figure 3-5: The authenticity/integrity check field (AICF) for the file data consists of a cryptographic hash tree. Contents of the dashed box appear in the inode.

We use the HMAC construction of Bellare, Canetti, and Krawczyk as our keyed cryptographic hash[2]. The HMAC function is defined as

$$\mathrm{HMAC}_k(x) = F(k, \mathrm{pad}_1, F(k, \mathrm{pad}_2, x))$$

where the commas represent concatenation, k is the key, pad_1 and pad_2 are sequences of a known constant, F is cryptographic hash, and x is the data being authenticated. In Cepheus, we use SHA1 as F and the file key as k.

Only someone with the file key can verify or create the AICF. If the file server colludes with an outsider (breaking our trust model), our design does not guarantee integrity. We would like to loosen this trust requirement, but we have not been able to avoid public-key techniques for such a model.

Reading an integrity-protected block works as follows. The client daemon obtains

the encrypted block in question and the inode containing the AICF. The client daemon then requests all the data pointer blocks which contain siblings to the nodes on the hash path from the block in question to the AICF. Next, the client daemon passes this information to the user agent for verification. The user agent verifies the hashes along the path from the block to the AICF. If the AICF is correct, the user agent returns the plaintext to the client daemon.

By caching these answers, we can avoid much recomputation for subsequent reads[18]. The client daemon caches data pointers in 8KB blocks. Moreover, the user agent will not recompute a hash for already verified paths. Hence, the cost of reading a block without interleaved writes is $O(1)$ amortized over a sequential read of the entire file of n blocks. We will read each data pointer once and compute each hash node exactly once.

Writing an integrity-protected block works in a similar manner. After writing a block to the buffer cache, the user agent schedules the block for encryption and a new AICF. The client daemon will eventually propagate updated data pointer blocks and the AICF to the file server. In the worst case, a write will take $O(\log n)$ time to recompute a path from the block in question to the AICF. However, the large branching factor (256) makes the depth of the tree at most three levels. This is effectively a constant factor.

3.3.3 Crash Recovery

Since our file server uses its own file structures, we must create our own version of `fsck`. As the structures are very similar to conventional file systems, it should not be extremely difficult to make modifications to `fsck`. Because we keep old lock boxes in the inode, we can attempt recovery of partially re-encrypted files.

If the plaintext cache is not flushed to disk because of a crash, the changes will be lost, but the file system will remain consistent. We could try other caching policies for better reliability at the cost of performance. We expect some problems related to integrity when the file server attempts to salvage a damaged partition. Since parts of the disk may be lost, the AICFs will report spurious errors.

3.4 Client Daemon

The client daemon acts as a dummy process to intercept local file system calls (see figure 3-6). The client daemon acts on requests from the kernel, makes requests of the user agent, and communicates with the file server. By using an NFS loopback server, we can avoid writing any in-kernel code. The loopback server pretends to be an NFS server, acting on file requests to a virtual directory.

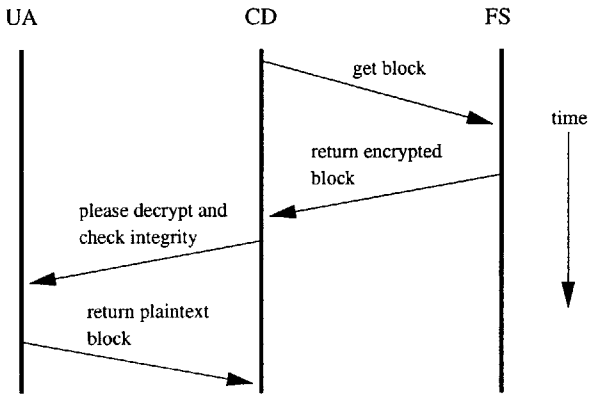

Figure 3-6: An example of reading a block: after the client daemon intercepts a read request from the NFS loopback server, it obtains a block through the following method. Had the block already existed in the cache, the client daemon would skip these steps and simply return the cached data.

3.4.1 Buffer Cache

An important decision involved placement of the plaintext and ciphertext cache in the client daemon. However, we made several arguments to keep the cache in the user agent. In the end, all of the arguments to keep the cache in the user agent were either impractical or obviated by the interface between the client daemon and user agent.

We decided to place the cache in the client daemon for three reasons. First, the client daemon will unavoidably see the flow of plaintext through the NFS loopback server. Hence, the user must trust the client daemon anyway. Because only one process on a machine can intercept NFS requests, one client daemon must intercept the file requests for all the users. Second, keeping the cache in the client daemon reduces

context switching and unnecessary memory copies. All decryptions and encryptions can happen in-place. The user agent can easily encrypt and decrypt information using shared memory by request of the client daemon. Third, we reduce the number of client RPC handles. Each client of the file server must obtain a client RPC handle by contacting the file server. The handle acts as a name to make requests of the file server. If each user agent were to contact the file server, the user agents would each need a client RPC handle. By making all file server requests flow through the client daemon, we reduce the number of client RPC handles. Moreover, the client daemon is a more natural place to keep track of client RPC handles.

While the client daemon controls the cache, the user agent can easily specify a storage location for the cache. For instance, the agent could ask the client daemon to store the cache in a locally encrypted file, a PCMCIA disk, or in memory.

We could have used other policies for flushing dirty blocks to disk. For instance, the write-on-close policy waits to write buffers until the file closes. We avoid this policy because with large files, the encryption can be costly if we wait for the file to close. The delayed encryption allows us to flush all writes related to a particular file after a set period of time.

3.4.2 File Server Communication

A second decision involved who should talk to the file server. We decided to let the client daemon function as a NFS loopback server – listening for file requests. Because the client daemon now talks to several different processes, both local and remote, the implementation involves multitasking and shared memory. We explain these details in section 4.2.

3.5 User Agent

The user agent responds to requests from the client daemon and communicates with the group database server. The user entrusts the user agent with his or her long-term private key. Whenever the agent needs to retrieve a file key or group key, it contacts

the group database server and opens the appropriate lockboxes with the user's private key.

3.6 File Server

The file server responds to Remote Procedure Calls (RPCs) from the client daemon. The server also has a raw interface to a disk and implements 13 system calls:

- `getattr`: get file attributes
- `setattr`: set files attributes
- `lookup`: look up file name
- `read`: read from file
- `write`: write to file
- `lock`: lock or re-lock a file for a bounded duration
- `unlock`: unlock a file
- `read_dp_block`: read an indirect data pointer block
- `write_dp_block`: write an indirect data pointer block
- `statfs`: get file system attributes
- `create`: create file
- `link`: create link to file
- `unlink`: decrement a file's reference count

Many of the above RPCs come from the NFS protocol. However, noticeably absent are the `readlink`, `remove`, `rename`, `symlink`, `mkdir`, `rmdir`, and `readdir` RPCs. Because the file server has no access to plaintext directories, the client must take care of all operations to directory entries. For this reason, we added a locking mechanism to achieve atomicity of client daemon RPCs which each generate several file server RPCs.

Note, we must also authenticate and authorize individual RPCs. If an adversary could forge authentic file server RPCs, we could not ensure availability of data. Hence, the RPCs require cryptographic authentication in addition to the confidentiality and integrity provided by Cepheus.

To check authorization, the file server obtains information from the group database server over an integrity-protected link. The group database server and file server can exchange long-term public keys through an out-of-band mechanisms. Each could then use the public keys to negotiate session keys.

The authentication of read requests is optional. Since file data is encrypted, one should not have to rely on access control, but we do not want to tempt fate. We set as default authentication of read requests, but the file server administrator can toggle this option to tune performance.

The file server must also make requests to the group database service. To reduce the online communication cost with the group database server, the file server periodically requests a complete dump of the authorization database. This allows for quick membership verification. Moreover, the file server can send the locked group keys when a user agent requests a file. In this way, the user agent need not always communicate with the group database service to obtain a group key.

3.7 Group Database Server

The group database server maintains the master list of group membership and user public keys. It also facilitates the distribution of shared symmetric group keys. We assume there exists a mechanism to initially add authenticated public keys of system administrators to the group database server. Further operations use the following RPCs:

- Add User: If authorized, given a name and public key PK, return a new UID and store PK. If name already used, error.
- Delete User: If authorized, mark UID and PK_UID as disabled. Remove UID from all group lists. Error if UID is currently a group owner.
- List users: If authorized, list all names of users.
- User 2 UID: If authorized, given a user's name, return the UID.

- UID 2 User: If authorized, given a UID, return user's name.
- Get User PK: If authorized, given a UID, return the PK_UID.
- Set User PK: If authorized, given a UID and PK_UID, store the PK.
- Get User Permissions: If authorized, find a bitmask representing the group database permissions of a user.
- Set User Permissions: If authorized, set the group database permissions of a user according to a bitmask.
- Add Group: If authorized, given a new group name and group key encrypted with creator's PK, return a group ID. Group is owned by the creator.
- Delete Group: If authorized, given a GID, delete all info about the group.
- List Groups: If authorized, list GIDs
- Group Name 2 GKID: If authorized, given a group name, return its group key ID. If does not exist, error.
- GKID 2 Group Name: If authorized, given a group key ID, return its name. If does not exist, error.
- Get Group Key: If requester is in the membership, return G_PK_UID. The user can then decrypt with his secret key SK. [authorization depends on UID and GID of requester]
- Add Group Member: If authorized, given a GID, UID, and group key encrypted in UID's PK, add UID to the membership (G_PK_UID).
- Delete Group Member: If authorized, given a GID and a UID, remove UID from the membership and delete G_PK_UID. [authorization depends on UID and GID of requester]
- List Group Members: If authorized, given a GID, return the UIDs of membership. [authorization depends on UID and GID of requester]

- Change Group Key: Complex. Every member needs the new key. If authorized, given a new key encrypted with owner's PK, and the same key encrypted with each member's key, flush the old membership and replace. [authorization depends on UID and GID of requester]
- Change Group Owner: Bestow another user with ownership of a group.
- Get Member's Groups: If authorized, return a list of all the groups which contain the member.
- User in Group: Returns whether a user is a member of a group.

The server operates in a manner such that it never sees a symmetric group key in the clear. When a user agent encounters a file protected with a group key, it contacts the group database server for a lock box containing the group key. The group key leads to another key which can decrypt the file. Since only the file owners and group members can open the lock boxes, files data and directory contents remain confidential.

This authentication system uses public key cryptography, but it could could have been implemented using secret key cryptography. Secret key cryptography may execute faster, but public key cryptography is more straightforward to conceptualize and design.

3.8 Design Alternatives

This section discusses alternatives to our design and reasons for their non-inclusion.

3.8.1 Ciphers

We selected RC5 because it can quickly generate key schedules, but no part of the design prohibits the use of another cipher (e.g., DES or the coming AES). We could have also used VRA or SEAL in a pseudo-random function mode instead of a block cipher. That is, the cipher is given a block number, a key, and a length. The cipher

then returns the appropriate pseudo-random mask for that extent. The mask is simply XORed with the plaintext. A drawback here is that re-encryption with the same key is subject to a chosen plaintext attack. If an adversary were to chose a plaintext, obtain the encryption, then see the encryption of a changed block, the adversary could simply XOR the old plaintext with the old encryption to obtain the key. The key would reveal the new plaintext.

3.8.2 Initialization Vector

We considered using one IV per file rather than one IV per block. This led to a few options. We could use one file IV as a seed to generate a pseudo-random IV for each block of that file as shown in figure 3-7. This would conserve space and allow the data blocks to be well-aligned on 8KB boundaries. However, changing any IV would require changing *all* IVs.

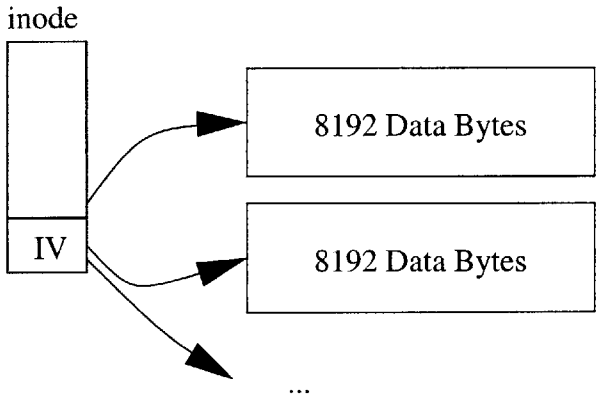

Figure 3-7: A pseudo-random function seeded with a block number and one shared IV could generate new IVs.

3.8.3 Authenticity/Integrity Check Field

We considered another mode for integrity. In his discussion on cryptographic protection of storage, Stephen Kent described the plaintext-ciphertext block chaining (PCBC) mode[17]. Figure 3-8 explains the structure of this mode. The nice feature about PCBC is that one change in the ciphertext will affect the rest of the file after

the change. The plaintext feedback propagates errors forward. This allows easy detection of unauthorized modification when the integrity field is the last block, but it also makes the storage much more brittle. One corrupted bit could ruin a file.

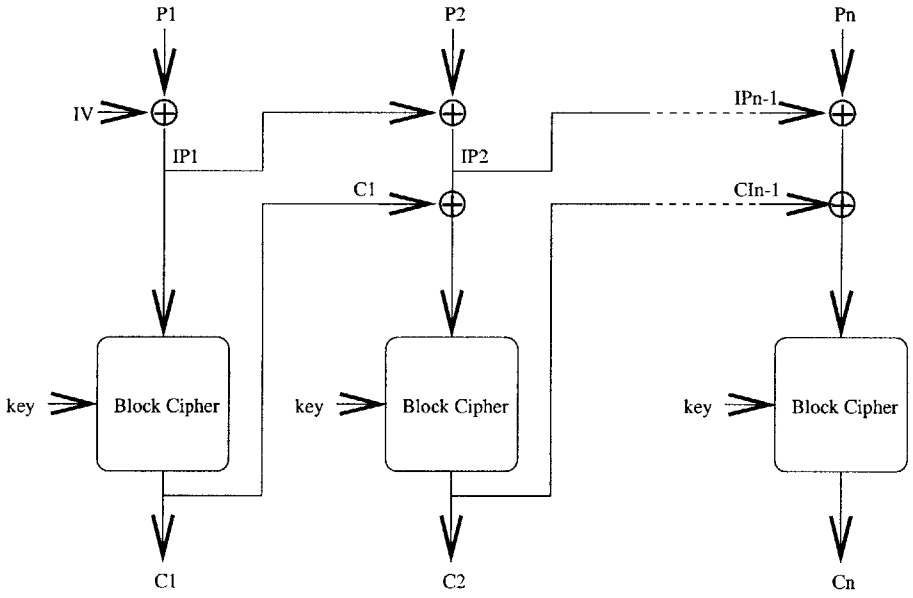

Figure 3-8: In PCBC mode, C_i = the encryption of $P_i \oplus IP_{i-1} \oplus C_{i-1}$.

If we were not to use a hash tree, the entire file would need to be downloaded, decrypted, and hashed in order to verify the integrity of just a single block. This would seriously prevent efficient random access for file reads.

We could have used a plain cryptographic hash, rather than a keyed hash for the integrity file of the file data. However, this requires that the plaintext remain confidential. Since a chosen plaintext attack is easy to mount and the inode metadata is *not* kept confidential, we kept the root of the AICF tree as a keyed hash.

Another alternative to a keyed hash is a digital signature. Our design tries to avoid overuse of public key cryptography because it is significantly slower than most symmetric key cryptography. However, we estimate that most files will not often require re-signing because most large binary files are read-only.

3.8.4 Delayed Re-encryption

Our initial design allowed parts of a file to be encrypted by many different keys. That is, we would only re-encrypt a block with the new key if *that* block changed. We opted for a simple scheme whereby the entire file is re-encrypted when the key changes. If different blocks within a file can have different keys, our inode will have an unbounded size and maintaining security will become more complex.

3.8.5 Buffer Cache

We considered three policies for cache consistency[41]. The simplest strategy, write through, writes a block to the file server immediately. The block remains in the client's cache for future reading. This achieves consistency, but does not reduce the amount of encryption or number of writes to the file server. A second strategy, delayed write, waits to write a block to the file server until the block is about to be ejected by the replacement policy[27]. But a client may not see another client's changes immediately – depending on timing. A third strategy, write-on-close, waits to flush buffers until a file closes. For implementation reasons, we chose the delayed write strategy. However, the third strategy would have a better balance between consistency and performance.

Chapter 4

Implementation Details

We implemented Cepheus in Red Hat 5.1 with the Linux kernel version 2.0.34. The implementation uses the standard Linux RPC package with the UDP protocol. For the purpose of quickly getting most of the functionality to work, we made several simplifications to the model of Cepheus. In particular, we integrated most of the user agent and client daemon as one process. This allowed the file system to quickly come online. The user gives the client daemon his or her secret key. To intercept file system requests, the client daemon acts as an NFS loopback server for the 15 standard NFS RPCs[24]. The NFS loopback client is built-in with the Linux kernel.

4.1 User Agent

We implemented most of the user agent functions in the client daemon. However, the user agent still contains the command-line interface to maintain users and group membership.

4.2 Client Daemon

The client daemon actually consists of two related processes. When the CD starts, it forks into a user agent registration process and an NFS loopback process. The CD_REGISTER and CD_NFS_LOOPBACK processes have shared memory and a file

descriptor for message passing (see figure 4-1).

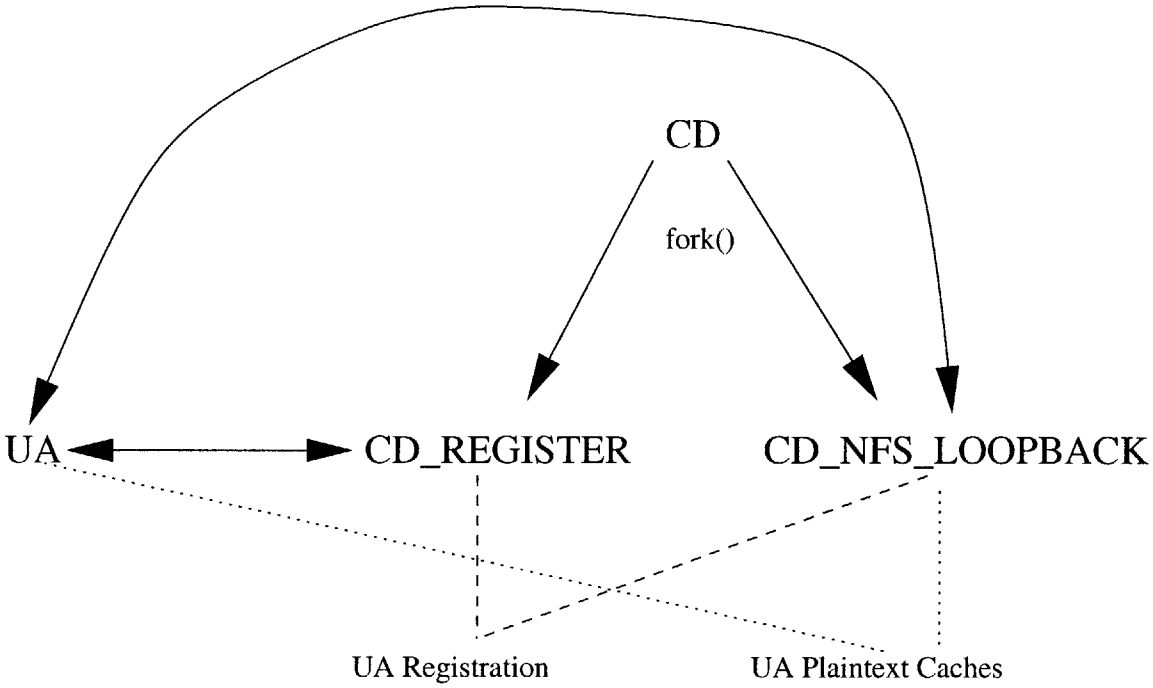

Figure 4-1: The client daemon (CD) forks into two processes. CD_REGISTER and CD_NFS_LOOPBACK share memory for a registration data. The user agent (UA) and CD_NFS_LOOPBACK share memory for the buffer cache.

CD_REGISTER listens for new UAs and handles the mutual authentication of the UA. The UA executes a file setgid to the group `csfs` to connect to a well-known, named stream pipe for a UNIX domain socket. When a UA connects, CD_REGISTER adds the UA to the state shared with CD_NFS_LOOPBACK. For each registered UA, this state includes a user ID (UID), semaphore on UID's buffer cache, pointer to shared memory of UID's buffer cache, length of memory, and a pointer to the next UA.

CD_NFS_LOOPBACK listens for NFS RPCs from the local kernel. When a request comes in, the CD_NFS_LOOPBACK determines the UID of the request and sends a message to the appropriate (and already registered) UA. The UA and CD_NFS_LOOPBACK share memory. In particular, they share memory of the buffer cache. CD_NFS_LOOPBACK asks the UA to encrypt and decrypt this shared memory. In this way, the UA does not have to disclose its keys to the CD.

NFS Loopback Server

There are a few tedious items to please the NFS loopback server. First, the client must run a modified `mountd`. The `mountd` program responds to mount requests (e.g., `mount -t nfs localhost:/csfs /csfs`). We use `mountd` only to please NFS semantics and to get the initial file handle of the virtual file system, `/csfs`.

With `mountd` and the client daemon running, one simply types `cd /csfs/hostname` to mount a remote Cepheus file server. The kernel notices that `/csfs` is a NFS mount point and forwards the request to the client daemon. The daemon then checks if `/csfs/hostname` is already mounted. If not, the client daemon connects to the remote file server.

4.3 File Server

The file server offers 13 RPCs as mentioned in the previous chapter. Most of the RPCs parallel that of the NFS RPCs. We require the RPC library to authenticate the RPCs. The file server also has its own file system. We store an entire Cepheus partition as a virtual disk. That is, we store the partition as a file in the standard Linux file system.

Because the NFS loopback server is stateless, we have no way to determine if a file is open or closed. Hence, we surrendered to using a delayed write policy instead of the encrypt-on-close policy. Had we implemented Cepheus in the kernel, we could detect a close operation. We expect a future version of Cepheus to operate partially within the kernel.

4.4 Group Database Server

The group database server keeps lists of groups and the group keys. For key distribution, a group owner sends the group key encrypted in each group member's public key. While this does not scale well as group sizes increase, it greatly simplifies key distribution and keys are not disclosed to the group database server itself.

In addition to the AICF for file data, we must also authenticate the requests themselves. For instance, the group database server must verify the authenticity and authorization of an ADDUSER request. We had several options of where to place the authentication:

- In the RPC library
- In additional RPC arguments
- Embedded in arguments needing authentication and authorization

We chose the first case because it can easily authenticate the all the RPC arguments. In the second case, the authentication mechanism is less transparent. Moreover, RPC already has a special argument for authentication. Had we chosen the last case, we would only authenticate parts of an RPC request. We might forget authentication of a seemingly innocent argument.

In SunRPC, each RPC request includes a generic authenticator. We simply use a standard authentication protocol in place of the default authentication mechanism. In security, it is better to err on the more secure side. By protecting all the arguments in each RPC request, we guarantee the authenticity of the entire request.

4.5 Status

At this time of writing, most of the code is written. We still have to finish the buffer cache on the client. Reads from the buffer cache work, but the delayed write policy has not been implemented yet. Preliminary performance results give an optimistic view of integrity protection of cryptographic storage. Amortized reads add very little overhead compared to the network delays from NFS.

Cepheus borrowed parts of the code from the Secure File System[22]. The cryptography involves the GNU multi-precision number package to handle public key operations. We use the Rabin-Williams public key algorithm for signatures, SHA1 for cryptographic hashes, and RC5 for the block cipher.

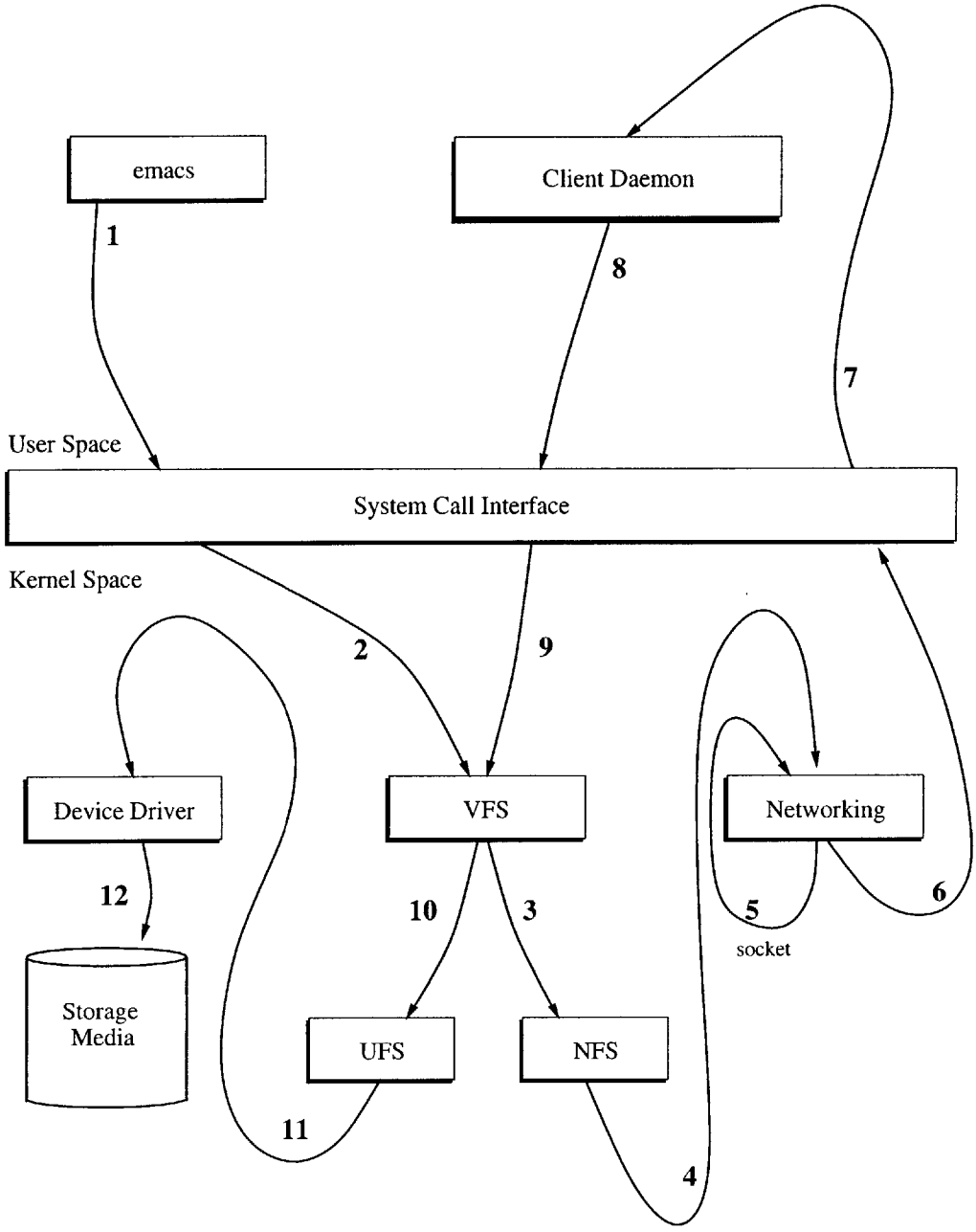

Figure 4-2: This picture shows the order of operations for emacs to read a block from Cepheus. Step 8 actually goes across the network to the file server's kernel[16].

Chapter 5

Questions for Cryptographic Storage

In the course of designing Cepheus, we considered several questions regarding cryptographic storage. In this chapter we briefly discuss some of the interesting questions.

5.1 Network Versus Storage Encryption

What are the advantages of cryptographic storage file systems versus those of cryptographic network file systems? Cryptographic network file systems protect a link between two end points and work securely if both end points are non-adversarial. Cryptographic storage file systems protect a link from users to users and performs well for read-only data. Cryptographic storage also separates confidentiality from storage management[32].

The Secure File System by Dave Mazières (same name as SFS, but different file system) implements two kinds of cryptographic file services: read-only and read-write[22]. Binary system files do not change often and therefore do not need to be re-encrypted. Moreover, SFS uses client caching and time leases on files to offset the cost of encryption. For read-only data, a cryptographic storage file system makes more sense than a cryptographic network file system. A cryptographic storage file system provides true end-to-end encryption. That is, user to user rather than user to

file server.

5.2 File Permissions

The UNIX file system provides three kinds of access permissions: read, write, and execute. An interesting problem exists when granting read access, but not write access, in a cryptographic file system. If a single key protects a file, how does one grant the ability to decrypt, but not modify and re-encrypt? One answer is to have multiple keys, one for encryption/decryption and one for the AICF. A similar but less harmful situation exists when a user has write access, but not read access (e.g., appending to a log file).

5.3 Group Sharing

How important is sharing for cryptographic storage? Sharing is uncommon, but adds valuable functionality if designed well[38]. Group sharing in the UNIX file system is not well suited to network file systems. If you have a stand-alone machine, it is easy to use groups. You edit `/etc/group` as necessary. But in NFS, system administrators must intervene for all group membership changes: creating a group, adding group members, removing group members, and so on. In the Andrew File System (AFS), users take advantage of group sharing because there is minimal intervention by system administrators. We would like to see a cryptographic storage file system that uses access control lists rather than the permissions in the UNIX file system.

The semantics of group sharing is also unclear. For instance, who owns a group? The question in the UNIX file system is mute: root owns and maintains the groups. But in a cryptographic file system, we need to redefine group ownership and group administration abilities. Users must take on roles previously held by omnipotent system administrators.

5.4 Incremental Cryptography

Incremental methods could help alleviate some of the burden of encryption[43]. Incremental encryption can speed up file encryption because most files are created by people who tend to follow patterns. Such patterns include appending more often than modifying files, reading more often than writing, creating small files, and not sharing too often[34]. Most files are decrypted more often than encrypted. In the case of public key cryptography, the decryption should be made to run faster than the encryption. Unfortunately, the UNIX file system does not operate in an incremental fashion. The UNIX file system does not allow inserts as does a database. To fully exploit incremental cryptography, we may need to implement security at the application level[43]. However, incrementally changing keys may alleviate costs of re-encryption in cryptographic storage file systems. Incremental methods would require significant changes to the file system interface.

5.5 Delayed Re-encryption

What consistency problems are expected from delayed encryption? Cepheus uses delayed writes and delayed re-encryption to avoid unnecessary encryption. Problems could arise if a set of files has not been fully re-encrypted after two key changes in a row. Should the file system wait until the first re-encryption finishes? There may be other ways to delay or avoid encryption. For instance, one could throttle the amount of encryption.

5.6 Integrity

The XFS file system from Silicon Graphics uses an adaptive approach to storing file data[40]. B+ trees keep the every file structure balanced. A similar strategy could reduce the average running time for creation and validation of integrity fields. For instance, we could reorganize the hash tree structure of the AICF such that data that changes often tends to affect the same path in the tree.

5.7 Orphaned Directories

An orphaned directory results when two directories with a parental relationship have different cryptographic keys. This is best illustrated by an example. Consider a directory `/home` and a subdirectory `/home/fubob`. Recall that a directory maps file names to inode numbers. To list the contents of the `/home/fubob` directory, the file system must first locate the inode for `/home` by reading the contents of the `/` directory[1]. With the inode for `/home`, the file system can read the directory contents to find the mapping from `/home/fubob` to another inode. Finally, the file system uses the inode for `/home/fubob` to list the directory contents.

Now assume that different cryptographic keys protect the `/home` and `/home/fubob` directories. The following table results:

	Have `/home/fubob` key	Not have `/home/fubob` key
Have `/home` key	Access to both directories	Access to `/home`
Not have `/home` key	Undefined	No access

Table 5.1: Access depending on available keys. When a user has the key to a directory, but not to its parent directory, access semantics are undefined.

Having the key for `/home/fubob` is useless if the key for `/home` is not available. There would be no way to locate the inode for `/home/fubob`. A cryptographic file system will have to define the access semantics under such cases. To have directory access independent from a parent directory, the file system cannot secure directory contents.

5.8 Brittleness

How can we avoid making the file system brittle[32]? If someone forgets a password, data may be lost. A single bit error can render a file useless under some chaining techniques[32]. We need to take extra precautions to avoid accidentally losing data. Data recovery and the ordering of operations is important. This thesis did not concentrate on the issue of brittleness, mainly because it is an extremely hard problem.

[1]For bootstrapping, the / directory has a hard-coded inode number.

5.9 Encrypted Swap File

How useful is an encrypted swap file? Once power is lost, the swap file could be quickly purged by forgetting the key. We have yet to find much documentation of encrypted swap files.

5.10 Key Rings

Where should keys be stored? We do not want to lose our keys, but we also do not want to carry large key rings. Cepheus uses the authorization server as a central repository for keys. But users could modify the user agent to use a local key ring. Key rings are more easily stolen, but a repository becomes a central point of failure.

5.11 Trust of System Administrator

Can integrity and availability be achieved without trusting the file server? With public key cryptography, we can separate integrity from storage management. But we have yet to find a way to separate availability from storage management. It seems obvious that the two are inseparable, but there could be a way to partially separate availability from storage management.

5.12 Unattended Access

How can daemon processes access encrypted areas of a disk? Should they? To print a postscript document, you must use `lpr`. Your file travels across the network to the printer in the clear. Can and should unattended programs be able to safely acquire cryptographic keys without user intervention? What are the end points?

Chapter 6

Conclusion

Cepheus provides confidentiality of file data and directory contents. It also maintains integrity and availability of file data, directory contents, and metadata. Cepheus implements a secure group sharing mechanism. If file systems become more ACL-based, we expect sharing to become more common.

We introduced delayed re-encryption. File owners can re-key a file while letting group members perform the re-encryption. A plaintext and ciphertext buffer cache uses a delayed write strategy to avoid unnecessary re-encryption.

In order for cryptographic storage to be widely used, cryptographic file systems will need to be compatible with existing file system interfaces. Moreover, the cryptographic file system must be at least as good as existing file systems in speed and functionality. The author also discovered that adding cryptography to a file system complicates the design several times over! In the future, we expect to develop Cepheus into a fully functional system. Moreover, we hope to use an asynchronous RPC package to process requests more efficiently.

Bibliography

[1] R. Anderson. Personal Communication, August 1997.

[2] M. Bellare, R. Canetti, and H. Krawczyk. Keying Hash Functions for Message Authentication. In N. Koblitz, editor, *Advances in Cryptology – Crypto 96 Proceedings*, number 1109 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[3] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental Cryptography: The Case of Hashing and Signing. In Y. Desmedt, editor, *Advances in Cryptology – Crypto 94 Proceedings*, number 839 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[4] M. Blaze. A Cryptographic File System for UNIX. In *Proceedings of 1st ACM Conference on Communications and Computing Security*, pages 158–165, Fairfax, VA, 1993.

[5] M. Blaze. Key Management in an Encrypting File System. In *Proceedings of the Summer USENIX Technical Conference*, 1994.

[6] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky. Deniable Encryption. In *Advances in Cryptology – Crypto 97 Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[7] G. Cattaneo, G. Persiano, A. Del Sorbo, A. Cozzolino, E. Mauriello, and R. Pisapia. Design and Implementation of a Transparent Cryptographic File System for UNIX. `http://tcfs.dia.unisa.it/`, 1997.

[8] G. Chick and S. Tavares. Flexible Access Control with Master Keys. In *Advances in Cryptology – Crypto 89 Proceedings*, number 435 in Lecture Notes in Computer Science, pages 316–322. Springer-Verlag, 1990.

[9] D. Coskun. Personal Communication, July 1998.

[10] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[11] Y. Deswarte, L. Blain, and J. Fabre. Intrusion tolerance in distributed computing systems. In *Proceedings IEEE Symposium on Security and Privacy*, pages 110–121, Oakland, 1991.

[12] K. Fu, P. McCormick, and J. Nicholson. A Simple File System: The X-File System. 6.033 Lab Paper. Contact fubob@mit.edu, 1997.

[13] P. Gutmann. Secure File System v1.20. `http://www.cs.auckland.ac.nz/~pgut001/sfs/`, 1995.

[14] P. Gutmann. Key safeguarding/anti-duress measures in cryptosystems. `http://www.mit.edu:8008/bloom-picayune/crypto/1389`, August 1997.

[15] S. Hartman. Personal Communication, 1997.

[16] J. Hu. Personal Communication, April 1999.

[17] S. Kent. *Protecting Externally Supplied Software in Small Computers.* PhD thesis, MIT, September 1988.

[18] B. W. Lampson. Hints for computer system design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 33–48, October 1983. Published as Operating Systems Review 17, 5 (1983).

[19] T. Liss and P. Tipton. The Discovery of the Top Quark. *Scientific American*, pages 54–59, September 1997.

[20] M. Loewenthal and A. Helwig. SecureDevice v1.4. `ftp://ftp.demon.co.uk/pub/ibmpc/dos/apps/secdev/`, 1994.

[21] J. Markoff. How a Computer Sleuth Traced a Digital Trail. *New York Times*, February 1995.

[22] D. Maziéres. Security and Decentralized Control in the SFS Global File System. Master's thesis, MIT, 1998.

[23] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[24] Sun Microsystems. RFC1094 - NFS: Network File System Protocol Specification, 1989. `ftp://ds.internic.net/rfc/rfc1094.txt`.

[25] R. H. Morris. UNIX Operating System Security. Technical Journal 8, AT&T Bell Laboratories, October 1984.

[26] ASIS Online. 1995/96 Trends in Intellectual Property Loss Special Report. `http://www.asisonline.com/`, March 1996.

[27] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. D. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Symposium on Operating Systems Principles*, pages 15–24, 1985.

[28] PGPdisk. `http://www.pgp.com/`, 1997.

[29] W. Price. CryptDisk. `http://www.primenet.com/~wprice/`, 1995.

[30] M. Ranum. Security Policies & Practices. In *11th Systems Administration Conference*, October 1997.

[31] R. Rivest. The RC5 Encryption Algorithm. In Bart Preneel, editor, *Proceedings of 1994 Fast Software Encryption: Second International Workshop, Leuven, Belgium*, volume 1008 of *Lecture Notes in Computer Science*, pages 86–96. Springer-Verlag, 1995.

[32] J. H. Saltzer. Hazards of File Encryption. MIT Laboratory for Computer Science Request for Comments 208, MIT, May 1981. `http://mit.edu/Saltzer/www/publications/csrrfc208.html`.

[33] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[34] M. Satyanarayanan. A Survey of Distributed File Systems. *Annual Review of Computer Science*, pages 73–104, 1990.

[35] J. Schmidt. Die Geheimniskrämer: Verschlüsselnde Dateisysteme für Linux. *c't magazin*, 19:228–230, 1998. `http://www.heise.de/ct/`.

[36] B. Schneier. *Applied Cryptography - Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, New York, second edition, 1996.

[37] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22(11), November 1979.

[38] M. Spasojewic and M. Satyanarayanan. A Usage Profile and Evaluation of a Wide-Area Distributed File System. In *Proceedings of the Winter 1994 USENIX Conference*, pages 307–323, San Francisco, CA, 1994.

[39] E. Swank. SecureDrive v1.4B. `http://www.stack.nl/~galactus/remailers/securedrive.html`, 1996.

[40] A. Sweeny, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the 1996 USENIX Annual Technical Conference*, 1996. `http://www.usenix.org/publications/library/proceedings/sd96/`.

[41] A. Tanenbaum. *Modern Operating Systems.* Prentice Hall, Upper Saddle River, New Jersey, 1992.

[42] A. Tormasov. TorDisk. `http://www.bpdconsulting.com/TorDisk/`, 1997.

[43] Y. Yerushalmi. Incremental Cryptography. Master's thesis, MIT, May 1997.

Biographical Note

Kevin Edward Fu was born on February 21, 1976 in South Charleston, West Virginia. He graduated from West Ottawa High school in Holland, Michigan in 1994, then earned a B.S. degree in Computer Science and Engineering from MIT in 1998. He belongs to the Sigma Xi and Eta Kappa Nu honor societies and has been active with ACM, the USENIX Association, and the National Senior Classical League. He serves as a general editor for the ACM *Crossroads* magazine. Kevin is married to Teresa K. Lai '98.