Cryptographic Access Control for a Network File System

Anthony Harrington

A dissertation submitted to the University of Dublin in partial fulfilment of the requirements for the degree of Master of Science in Computer Science

September 17, 2001

Declaration

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

Signed: ____________________

Anthony Harrington

Date: September 17, 2001

Acknowledgments

I would like to thank my supervisor Christian Jensen for his help and the interest he displayed in the project throughout the year. I would also like to thank my family and friends for their support and encouragement. Finally my classmates whom I will miss and from whom I have learned more than a little.

Abstract

The rapid growth of mobile computing and the deployment of large distributed systems which span multiple administrative domains has created new challenges for security systems. Changing usage patterns and the physical limitations of mobile devices require users to rely on services provided by remote elements in the system to facilitate data access and storage requirements.

Users need to minimize the trust placed in remote elements of the system such as the transmission mechanism and file server which may be beyond their control. They require secure mechanisms for accessing and sharing protected data. The access control mechanism deployed on the remote server should be flexible but capable of providing the levels of security required. Finally access control should not constitute a bottleneck in the system.

This thesis will investigate the feasibility of using cryptography as a form of access control in distributed systems. Cryptographic Access Control is a new paradigm in access control mechanisms which uses a combination of symmetric and asymmetric cryptography to provide user authentication, data confidentiality and integrity in distributed file systems. Its design reflects the challenges placed on systems by the changing operational environment. It is flexible enough to operate across multiple domains with varying security requirements and offers a high degree of resource protection in untrusted environments. It makes no assumptions about the security of remote elements in the system such as the transport mechanism or file server and only requires that the user place complete trust in his own machine.

Access control is essentially determined by the possession of cryptography keys. Possession of the public key allows a user to have read access to data and possession of the private key allows a user to have write access to data.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Computer science and related technologies are in a continual state of flux. New developments continually emerge in response to the varying needs of an ever-growing and increasingly diverse user community. The advent of the computer network was arguably the single greatest contributor to the pivotal role played by computer systems in our world today.
The most obvious example of this has been the use of the Internet as a vehicle for a wide range of personal and commercial applications in areas ranging from banking and finance to entertainment, education and communication. The widespread availability of networks such as the Internet as a medium for communication has prompted a proliferation of both stationary and mobile devices capable of sharing and accessing data across networks spanning multiple administrative domains.

Mobile devices in particular can when networked, access vast amounts of data, yet are constrained by their limited amount of local memory and disk resources. However through the use of technologies such as FreeNet [1] and iDrive [2] such devices can store large amounts of data remotely. Such collaboration increases user dependence on

[1]Peer-to-peer distributed file system.

[2]Delivers network storage for mobile devices.

remote elements of the system which are often outside their control and heightens their need for effective security systems.

1.2 Security in Open Systems

Security systems aim to ensure the integrity and confidentiality of protected resources. Various schemes exist for controlling access to resources. Traditional security mechanisms such as those employed on Unix and NT systems maintain a list of authorized users and the resources those users are allowed to access. When a request to access a resource is received the identity of the user is determined and the list of users allowed to access the resource is retrieved. If the user is among this list then he is allowed access to the data. If a malicious attacker can fool the access control mechanism into believing that he is a valid user then he can compromise the integrity of the system.

The shift toward the networking paradigm has resulted in a fundamental re-evaluation of computer security and its place in system design. Traditional access control mechanisms are unsuited to use in distributed systems. They are by definition centralized and provide a single point for attack and for system failure. They enforce a homogeneous security policy on a heterogeneous environment and require global knowledge on the part of the entity defining security policy. This is infeasible in a distributed environment in which the lack of global knowledge is a defining characteristic.

The overhead of user authentication and resource access validation constitutes a bottleneck in the system. This bottleneck may become unacceptable in a large system with many thousands of users. The trend toward networked computers and distributed computing only heightens this problem as the potential number of users in a system is vastly increased.

The behaviour patterns of users are also changing. A user may wish to temporarily store or access secure data on a remote server without establishing an account on

that server. It should be possible to accommodate these users in a fast and effective manner.
Todays distributed systems often span multiple administrative domains with varying security requirements that are incapable of expression in centralized monolithic access control mechanisms. Given the lack of centralized control or authority in these systems it is vital that the user can minimize the trust he places in remote elements of the system.
Security models need to be adapted to reflect this new reality.

1.3 Cryptographic Access Control

The CAC mechanism was developed by Declan O'Shanahan and formed the basis for his Msc Thesis[O'S00]. CAC is a new paradigm in access control mechanisms which uses cryptography to provide user authentication, data confidentiality and integrity in distributed file systems. Its design reflects the challenges placed on systems by the changing operational environment. It is flexible enough to operate across multiple domains with varying security requirements and offers a high degree of resource protection in untrusted environments. It makes no assumptions about the security of remote elements in the system such as the transport mechanism or file server and only requires that complete trust be placed in the client machine. It is also a design goal of the CAC mechanism that the greater burden of access control be moved from the server to the client.

Cryptography is employed at the granularity of the file level and associated with each file is a symmetric key and an asymmetric key pair. Encryption and decryption of file data is performed on the client machine and only encrypted data is stored on the server. Symmetric cryptography is used to provide data confidentiality and digital signatures are used to provide user authentication and data integrity.
Server response times for file access requests require no explicit validation. The data

is unintelligible to a user unless they possess the key necessary to decrypt it. Requests to update files will however require explicit validation which is accomplished through the use of digital signatures. This is necessary to prevent the overwriting of valid data.
Access control is essentially determined by key possession. Possession of the private asymmetric key enables the user to generate valid digital signatures and perform an update on file data stored on the server. Possession of the public key enables the user to validate the data returned by a read operation.

1.4 Objectives

This thesis will examine the feasibility of replacing traditional access control systems with a new model based on the use of cryptography at the level of the file system. We have outlined the changing demands placed upon security systems in modern distributed systems. Our primary objective will be to build a Network File System which uses the CAC mechanism and to determine the effectiveness of our solution in meeting these demands.
NFS is designed to be a stateless service. An NFS server has no concept of the file as a logical unit and processes files as a series of data block writes. An efficient implementation of our CAC mechanism requires that the server be able to perform operations on the file as a logical unit and determine which users are updating open files. Adding state information to our modified NFS server will present a significant challenge.

1.5 Dissertation Outline

This chapter has outlined the growing trend toward distributed computing and the changing demands placed on security systems. It also provides a brief overview of the Cryptographic Access Control Technique.

In chapter two existing access control schemes and cryptographic file systems are examined. We also present a detailed discussion on the various security mechanisms available for the Sun Network File System.

Chapter three contains the design and analysis of the security mechanisms applied in our system. In order to prove our concept we have implemented a prototype Network File System.[3]

Chapter four describes the technologies used in constructing the prototype. Given the limited time available to the project we implemented our prototype as a user level solution. This facilitated the rapid development of our code and the use of existing cryptography libraries.

Chapter five contains an analysis of the performance of our prototype. Chapter six presents the conclusions of our work and suggests some extensions to the prototype file system.

[3]We have used the Linux user level NFS code-base

Chapter 2

State of the Art

2.1 Access Control Systems

As the name indicates Access Control systems are concerned with mediating on all requests from system users or processes to make use of a resource on the system. For the purpose of this document such a resource may be considered to be a data file. Although there are a number of access control schemes in use today, the US computer security standard TCSEC [U.S83] identifies two principal types:

- Discretionary Access Control

  These systems allow individual users and file owners to manipulate the access permission data for resources. A DAC mechanism allows users to grant or revoke access to any of the objects under their control without the intercession of a system administrator[FK92]. Such schemes are often referred to as optimistic access control. As defined by TCSEC these schemes provide:

  > A means of restricting access based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission on to any other subject.

- Mandatory access control schemes

  MAC schemes control access to resources based on the security labeling of that resource. Users accessing protected resources are unable to grant access to users with a lower security clearance, and formal methods exist for reclassifying the security level associated with protected resources. As defined by TCSEC these schemes provide:

  > A means of restricting access to objects based on the sensitivity of the information contained in the objects and the formal authorization of subjects to access information of such sensitivity.

  MAC systems are typically employed in multi-level secure military systems and are also referred to as pessimistic access control schemes.

2.1.1 Access Control Lists

Traditionally Unix/Windows NT systems have employed a form of access control based on the concept of an ACL [1]. Such systems store a set of access permission bits with each resource or file. Requests of the form {resource id, operation id, principal id} are received from clients and the server verifies that the principal is included in the list of authorized principals stored in the relevant access control list. If the principal is listed, the requested operation is examined and if it is valid, ie. if it is included in the list of allowed operations in the resource's ACL entry, then the operation is performed. Such systems have the advantage of readily identifying which principals can access a given resource and which operations are valid for those principals to perform on that resource. However given the identity of a principal such systems do not readily identify the resources to which that user has access [Gol99]. These systems are said to be optimistic or discretionary in nature as the resource owner is trusted to set appropriate permissions in the ACL. This traditional form of access control was designed for use in a centralized and secure environment. If used

[1] Access Control List

in a distributed system, this mechanism forces a high degree of uniformity between the security architecture of the client and server systems as the same user and action identifier scheme must operate on both. This may be not always be desirable or even possible. The access list look-up operation also acts a bottleneck and is a single point of failure and attack in the system.

2.1.2 Role Based Access Control

RBAC systems are an example of a mandatory access control policy and have traditionally been used in sensitive military and commercial environments where data integrity is a paramount consideration. RBAC systems differentiate between user identity and the role or function to which a user is assigned. Once a user's role has been determined he may only perform suitable tasks for that role. The determination of membership and the allocation of functions to roles is performed by the system administrator in accordance with organization specific guidelines[FK92]. This alliance of user function and access control is one of the strengths of RBAC systems as it allows the security policy of the system to reflect the organizational structure of the operational environment. The fact that users cannot pass access permissions on to other users at their discretion is a defining characteristic of such systems. These systems attempt to bind the data access and data transformation procedure together and are more concerned with access to functions and information than strictly with access to information[FK92].
Essentially these systems conform to three rules:

- Role Assignment

  All users/subjects must have selected or been assigned a role prior to executing a transaction. [2]

- Role Authorization

  A user can only activate a role for which they are authorized.

[2]A transaction encapsulates the transformation procedure and the data storage access

- Transaction Authorization

  A user can only execute a transaction if that transaction is authorized for the subject's active role.

This access control scheme does not require any checks on the user's right to access a data object, or on the transformation procedure's right to access a data item, since the data accesses are built into the transaction[FK92]. The finer grained access control mechanism supported by RBAC systems facilitates the Principle of Least Privilege [3] and allows for a clearer separation of duties.
This approach although more scalable than the ACL method outlined earlier, requires that the organizational structure of all entities in the system be capable of encapsulation in a system wide set of roles and transactions. This is a reasonable expectation in a system contained within a single administrative domain, but may prove too great a constraint in a distributed environment.

2.1.3 Capability Based Access Control

A CBAC system is inherently more distributed than an ACL or an RBAC system. Capabilities are essentially binary values that act as access keys for resources[Gol99]. They are discretionary in nature as capabilities can be distributed to other users [4] Requests of the form {op, user id, capability} are received from clients and the server verifies that the capability references a valid resource and that the specified operation is in the set permitted by the capability[CDK01]. Any user who holds a capability can distribute it to another user.
While this is a highly flexible system it also introduces two new problems not present in the previous access control schemes, namely key theft and revocation. A capability obtained in a fraudulent manner is still valid and may have become widely dispersed

[3]That a user only have access to the minimum set of resources needed to accomplish their task
[4]With the exception of identity based capability schemes

throughout the system. This problem is aggravated by the fact that capabilities are difficult to revoke and although they may be used with timeouts, they must then be periodically redistributed to all valid users.

The Cryptographic Access Control mechanism is quite similar to a capability based scheme. The public and private asymmetric keys of a file can be viewed as read and write capabilities respectively. However our access control scheme has the advantage of minimizing the trust placed on the integrity of the server. Like all capability based schemes there are complex capability distribution and revocation issues to be solved.

2.2 The Sun Network File System

The Network File System was originally designed and implemented by Sun Microsystems for use on its Unix based workstations. It allows an arbitrary collection of networked clients and servers to share a common file system[Tan95] [CDK01]. Clients using NFS can transparently access files stored on remote file servers as if the files were actually stored locally.
The NFS architecture as illustrated in Figure 2.1 is based on the traditional client-server model. The NFS specification defines two server daemons which provide the functionality of the system. The mountd daemon is responsible for mounting a directory of the remote file system to a directory on the client machine. Clients invoke this daemon to obtain a filehandle for the remote directory. If the NFS client successfully mounts a remote directory a filehandle is returned by the server. The filehandle contains information which uniquely identifies the remote file system type, the disk and the i-node number of the directory. This filehandle is used by the NFS client to create an r-node [5] in its internal tables.The client Virtual File System layer then allocates a v-node[6] to point at the r-node. Any future operations on the directory

[5]remote-inode

[6]virtual-inode

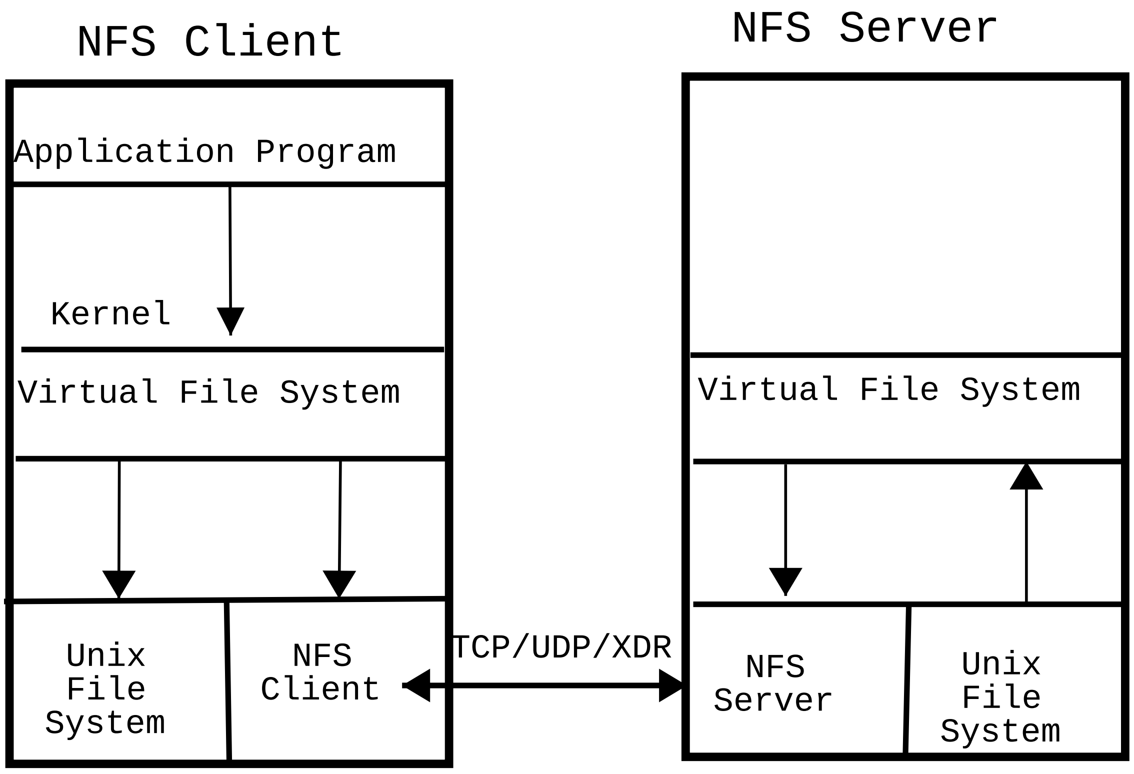

Figure 2.1: The Sun Network File System

associated with this v-node leads to a NFS client call using the r-node - filehandle mapping to the NFS server on the remote machine on which the directory is stored. The second server daemon known as the nfsd daemon, is responsible for handling these directory and file access requests from the client. This daemon implements most of the File System calls available on Unix systems.

The "/etc/exports" file serves as an access control list for those file systems which are available to be mounted by NFS clients. Each line contains an export point and a list of machine names allowed to mount the file system at that point. The access type read/write may also be specified in this file. One of the design goals was to allow file sharing to be transparent to the user and the application programmer. To this end, the Sun implementation of NFS is built on its RPC [7] technology.

[7]Remote Procedure Call

2.2.1 Remote Procedure Call

Through the use of RPC a client program on one host can invoke a procedure on another networked host acting as a server[Sri95a][Tan96] [Ste99]. The RPC architecture is illustrated in Figure 2.2. Among the more attractive features of this technology are the provision of location transparency to the user and the application programmer and the ability of the RPC mechanism to work among clients and servers of heterogeneous architectures. [8] Stated simply the first feature allows the invocation of remote procedures to appear as if they are executed locally. This is accomplished through the use of client and server stubs. The role of the stub is similar to that of a proxy[CDK01]. The client stub acts like a local procedure to the client but is actually responsible for packing the parameters to the function call and the function call identifier into a request message and transmitting them across the network. The server stub receives the packets, unpacks the function parameters and invokes the specified procedure with the parameters received from the client. The results are then returned to the server stub where they are packed into network packets before being re-transmitted to the client across the network. The client stub unpacks the results and returns them to the original caller.
If a user wished to invoke a read operation on a file which is being accessed via NFS then the parameters to the read call and the requested data are transmitted between client and server in the above manner.

There are seventeen services offered by the NFS server to clients[Mic89]. These services are roughly analogous to the standard Unix File System calls and are implemented using the RPC mechanism. Sun RPC uses at-least-once semantics which means that if the server fails while processing an RPC, the client kernel will wait indefinitely for a response in order to guarantee that the RPC will complete at least once.

[8]See [Tan95] for a detailed critique of RPC's transparency

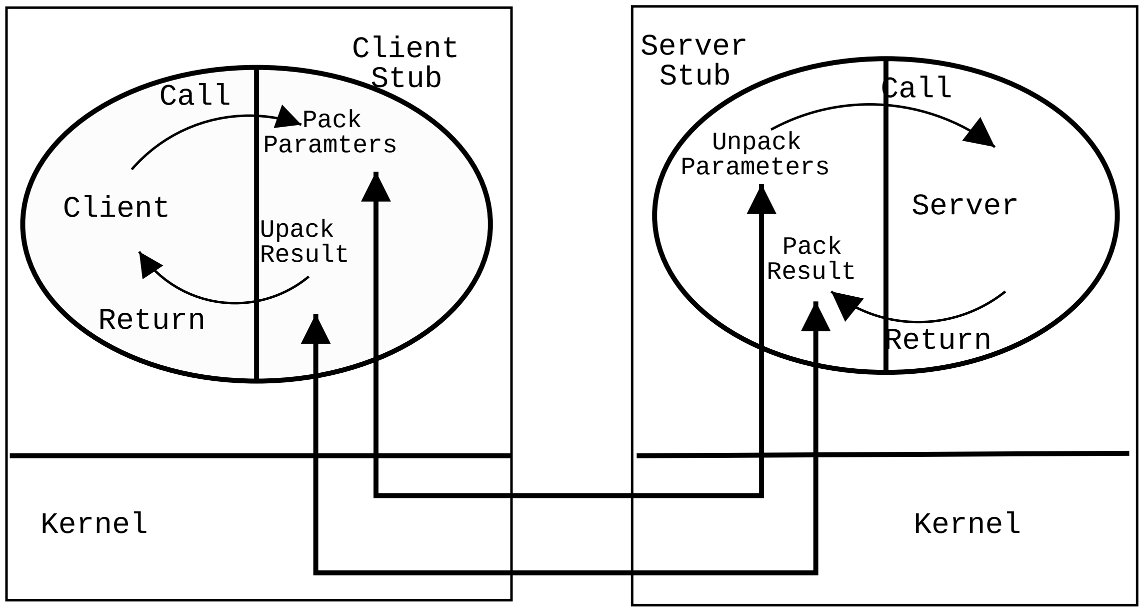

Figure 2.2: RPC Architecture [Tan95]

2.2.2 External Data Representation

Sun designed NFS to operate in a heterogeneous environment. Various clients with different architectures, data formats and types needed to be able to exchange data transparently to the user. The x86 family of processors are little endian whereas Sparc machines are big endian. Fundamental data types also vary in size from one platform to another. And yet a Sparc machine can act as an NFS server to an x86 client and vice-versa. This is accomplished through the use of the XDR [Sri95b] standard. XDR is both a language for describing the data and a set of rules for encoding the data[Ste99] and is similar in appearance to the C language.
The basic block size used in XDR is 4 bytes and all data types encoded in XDR require a multiple of 4 bytes of data[Sri95b]. The XDR byte order is big endian. Among the standard data types supported by XDR are:

- Signed and Unsigned Integers

  Signed and unsigned integers are 4 bytes in length. The signed integers are represented in two's complement notation.

- Floating Point Numbers

  Floating point numbers are represented by 4 bytes using the IEEE encoding standard for single-precision floating point numbers [Sri95b]. The sign is represented by 1 bit, the exponent of the number is represented by 8 bits and the mantissa by 23 bits.

- String

  A string of ASCII bytes is represented by an unsigned integer n followed by n bytes of data.

The XDR language also supports user defined data types through the use of a *struct* definition. This is analogous to the *struct* keyword in C.

2.2.3 NFS Security

NFS has never enjoyed a reputation for security. When NFS was originally deployed in the early 1980's its operational environment was very different from today. NFS clients were typically diskless workstations with few resources. Servers and clients were members of the same administrative domain and operated on private networks which were easily secured. That situation has changed radically with the advent of open networks, multiple administrative domains and clients capable of forging RPC requests.

As previously stated NFS is built on the RPC mechanism, which is itself insecure. Every time an RPC program starts up it must register with the portmapper process at which time it is bound to a particular port number. Subsequent bindings may result in a different port number being assigned to this program. This makes it difficult to protect RPC services using centralized firewalls. A secure portmapper exists [9]which makes it possible to restrict access to the portmapper services to a limited set of hosts but malicious attackers can bypass the portmapper and scan ports at will to discover RPC programs. The RPC specification does allow for the

[9]Wietse Venema has developed a secure portmapper for Linux

inclusion of authentication information with each procedure call[Eis99]. NFS can use the RPC authentication information to implement a variety of security methods. Among the more popular authentication schemes used are:

- AUTH_NONE

  Using this flavour of security no authentication information is passed between the client and server.

- AUTH_SYS or AUTH_UNIX

  This is the default method of authentication. Using this method the effective user and group identifier of the client are sent to the server. The server will then perform a lookup on the access control list associated with the specified file to determine if the client request should be allowed. The NFS server must map the clients UID space to that of the server. Various implementations provide different methods of accomplishing this. The Linux user level NFS server offers three methods for accomplishing this:

  - static mapping

    The nfsd daemon references a file which explicitly maps client UIDs to server UIDs.

  - dynamic mapping using NIS [10]

    The nfsd daemon queries the NIS server of the NFS client for the UID and GID corresponding to the user and group names of the client.

  - dynamic mapping using ugidd

    The ugidd daemon runs on the client machine and returns the user name associated with a UID.

  Compromising this security scheme is trivial. The RPC authentication information namely the UID and GID pair are sent across the network in the plain. These fields can be overwritten or duplicated by malicious agents who

[10]Network Information Service

can then access the data stored in the clear on the server. As well as providing no data integrity or confidentiality safeguards, this solution scales poorly and forces a tightly coupled relationship between client and server which may be undesirable in the modern operational environment of mobile computing.

- AUTH_DH or AUTH_DES

  This flavour of authentication uses the Diffie-Hellman protocol to exchange secret DES keys.[11] Users and hosts have their own secret session keys. These keys are used to encrypt the user identifier information included in the RPC request. When the client sends invokes its first RPC call on the server it generates a secret DES key and encrypts it using a public key scheme which relies on the Diffie-Hellman protocol for key exchange. Subsequent call to the NFS server include a verifier in the RPC request. This verifier is a client timestamp encrypted with the secret DES key that only the client and server share. The server decrypts this timestamp and if it is within a tolerance level of the servers time then the server can be sure that client possesses the valid key and is thus authenticated. Obviously some form of time synchronization is required. Again this model is fundamentally unsuited for use in today's open systems. Only the origin of the packets, namely the user and host are verified and not the actual content of the packets.

- AUTH_KERB4

  The Kerberos authentication system developed at MIT has become an industry standard for securing networked servers against unauthorized access and imposter attacks[CDK01]. A hybrid form of Kerberos has been applied to the NFS standard[CDK01]. When the client kernel executes a mount call it first contacts the local kerberos server and gets a ticket and session key. The ticket contains among other information the client timestamp. The client then issues an *nfsproc_getattr()* function call to the server using the newly allocated ticket.

[11] See [Sch94] for a description of this protocol

This function is not secured in most nfsd implementations.[12] The nfsd daemon then makes a call to the local Kerberos server to validate the client ticket. Upon validation of the ticket the Kerberos server returns the secret DES key to the NFS server and the UID associated with that key. Using this key the server decrypts the client ticket and checks that it is valid. If the UID and the timestamp match then this information is stored on the server along with the necessary mount information for that user. Further requests from that user are authenticated only if the UID and network address encrypted with the DES session key match those established at mount time. A drawback of this scheme is that only one user can be logged on to a client machine at any one time. This scheme also fails to address confidentiality or security of the data content of the RPC requests and replies. It also forces both the client and server to trust another external entity, the Kerberos server. Establishing a common trusted third party in open networks is difficult and widens the trusted computing base. It also mitigates against scalability.

- AUTH_KERB5

  The NFS protocol can use Kerberos V5 security over the RPCSEC_GSS security flavor[Eis99]. This security flavor sits on top of the GSS-API [13] for network transmissions and supports data integrity and privacy as well as the user authentication services offered by the earlier protocols. Before exchanging data between client and server, an application must establish a context for the exchange. The security mechanism, QOP [14], and type of service parameters are established at context creation.The QOP parameter details the encryption algorithms to be used with integrity or privacy mechanisms. Once a context is

[12]The *getattr()* procedure is implemented by the nfsd daemon and invoked by the mountd daemon at mount time. At mount time authentication information used in AUTH_DES and AUTH_KERB is not available. In order for automounter systems to work this operations are unsecured. The only information a malicious agent can gain from the *getattr()* procedure are file attributes.

[13]Generic Security Service API

[14]Quality of Protection

established, applications can reset the QOP and type of service for each data unit exchanged. DES is used to ensure data privacy and MD5 and DES-MAC are used to guarantee the integrity of the RPC requests and data. An implementation of AUTH_KERB5 is available from MIT. All data is unencrypted prior to being stored on the server or client. While AUTH_KERB5 ensures the security of the data while in transit, it does not address the security of the data while it is stored on the server. It also suffers from the weaker trust model and scalabilty issues identified with AUTH_KERB4. It should also be more computationally expensive than our system as it requires an extra round of decryption on the server prior to data storage.

2.3 Cryptographic File Systems

2.3.1 Cryptographic File System

CFS is the parent of cryptographic file systems. [15] It is based on the NFS file system and is implemented entirely at the user level[Bla93]. Each machine runs its own modified NFS server. When the machine is booted up the server daemon is started and a local directory is mounted. All requests to read and write files on this directory are intercepted by the CFS server. Encryption is implemented at the granularity of the directory with all files in the directory encrypted and decrypted with the same key. The encryption and decryption of data is transparent to the user.

CFS was designed to be portable and uses the underlying file system storage and access control procedures. It was implemented over NFS and used an underlying block size of 8K. Each 8K data block is independently encrypted in ECB [16] mode in order to preserve byte boundaries within the file. The encrypted data block is then xor'ed with a bit mask produced from the DES OFB mode. [17] A modified version of

[15]It was developed by Matt Blaze in the late 1980's

[16]Electronic Codebook Mode

[17]Output Feedback Mode

the *mkdir* command, *cmkdir* is used to create an encrypted directory. At this point the user is prompted for a passphrase which is then used to generate a DES key for that directory. The secret key is only stored on the client machine. In an effort to prevent malicious users from issuing RPC requests, the CNFS server will only accept calls originating on a secure port.

There are a number of similarities between the security model employed by CFS and that which we have designed. The storage of data in encrypted format on the server does limit the trust required in the server. We have also chosen to employ encryption at the file system level and to make its use transparent to the user. The actual architecture of our prototype is similar in that we have also modeled our solution on a user level NFS server. However CFS uses the access control mechanism provided by NFS. We have attempted to exploit the presence of cryptography in the file system to facilitate a new access access control paradigm and provide a model more suited to the evolving needs of mobile computing.

2.3.2 Transparent Cryptographic File System

The TCFS system was developed at the University of Salerno in Italy, and is currently only supported on the Linux platform. It can be thought of as a more sophisticated CFS in that the functionality has migrated away from user level space and into the client and server kernel. The core functionality of the system sits underneath the VFS[18] level and renders the encryption and decryption operations completely transparent to the user[CP97].

It has a similar trust model to the CFS system outlined above and decrytpion only takes place on the client machine. A special flag is used to mark encrypted files apart from normal files and TCFS can work transparently with either type. All files belonging to this user encrypted by a single DES key. As expected TCFS outperforms the user level solution, however it too failed to take advantage of the presence

[18]Virtual File System

of cryptography in the system when implementing its access control policy relying instead on the traditional ACL model.

2.3.3 Truffles

Truffles [19] recognizes that the operational environment is no longer homogeneous and that a set of distributed co-operating entities often exist in multiple administrative domains[RPC$^+$93]. It is based on the Fiscus file system which uses data replication to ensure high availability of user data[GHM$^+$90], but it also attempts to address security issues introduced by data sharing in an insecure environment.

Truffles uses PEM [20] service to pass messages between users with the purpose of requesting files, authenticating user requests and establishing which keys will be used to encrypt shared data [RPC$^+$93]. Once these administrative tasks have been taken care of, the Fiscus file system is used to handle the data replication and updating tasks. The PEM system relies on using public keys to authenticate messages. A certification authority is used to provide X.509 certificates to bind a user name to a public key. Truffles employs the traditional Unix access control mechanism and its designers needed to find a suitable method of mapping domain specific user identifiers across disparate domains. This is accomplished by associating a locally significant user identifier with a globally unique identifier. This unique identifier is stored along with the file ownership information. Initial Truffles implementations used X500 distinguished names for the unique identifiers.

The Truffles security model is quite different from that which we propose. Data is only encrypted while in transit and the use of traditional access control models inhibits the scalability of the system. The practice of using PEM to establish user connections may also hinder collaboration among users[Jen00].

[19]TRUsted Fiscus FiLE System

[20]Privicy Enhanced Email

2.3.4 CryptFS

CryptFS is a kernel level cryptographic file system which can be mounted on any directory and which can utilize any underlying file system such as UFS or NFS[ZBS]. CryptFS has a novel solution to the problem of key storage. When the user logs in, they are prompted for a passphrase which is hashed using the MD5 cipher to produce a key, this key is then stored in memory by CryptFS. The encryption and decryption operations are transparent to the user.

CryptFS has been implemented as a stackable v-node[21] interface[ZBS]. As CryptFS uses the v-node interface it can be used on top of any underlying file system without modification to that system. It uses an underlying block size of 4-8K

The Blowfish[Sch94] symmetric cipher is used in CBC mode to encrypt and decrypt data. The CBC routines are executed on each underlying block of data separately. This allows for decryption of individual blocks on request from the user. File names are also encrypted and normalized to prevent illegal characters from appearing in the ciphertext.

One of the premises in the design of CryptFS is that restricted files are rarely shared[ZBS]. As such the access control mechanism employed by CryptFS is based on a combination of the PID [22] and/or the UID. While it is relatively easy to obtain the UID of a user it is far more difficult to obtain the PID of the original process that prompted the user for the passphrase associated with the key.

Changing the key associated with a file requires the use of a dedicated program with creates a new key, reads the file with the old key and writes it back with the new key.

CryptFS provides a novel and effective file system for storing encrypted data. Its location in the kernel coupled with its use of the v-node interface make it fast, transparent and portable. However its access control policy is not designed to be

[21]A Virtual Node is a Unix data structure which provides a uniform interface to all file system objects such as files, directories, block devices etc

[22]Process Identifier

scalable and its use of PIDs in determining access to keys militates against the sharing of data among large user communities.

2.4 Log Structured File Systems

In the initial phase of the project we considered using a Log Structured file system on our file server. This would have allowed for a more flexible write validation routine on the server as write semantics would have been append only. The terminology used in the literature can sometimes be inconsistent. We regard the terms Log Structured or Versioning to refer to file systems in which both file data and meta data are preserved across multiple file system changes. It is a requirement of such a system that a user should always be able to access earlier versions of a file on demand.
Journaling file systems are taken to refer to file systems in which all file meta data updates are written to a system log file using transaction semantics. Such file systems are useful as they enable rapid recovery from server crashes. Traditionally a Unix file server on restart after an unclean shutdown must perform a *fsck()* operation on an entire partition to ensure the system is restored to a consistent state, however a Journaling file system need only replay the meta data updates following the last committed update in the log file.
However while Journaling file systems such as Reiserfs or IBM's JFS are available, log structured file systems are something of a rarity. Certainly a number of such projects were under development but no suitable system that could have been used as a codebase was discovered.

2.4.1 Elephant File System

One file system we examined in detail is the Elephant File System. The Elephant project is undergoing active development [23]. It is a Log Structured file system implemented on the FreeBSD operating system. However Elephant is a centralized file

[23]At the University of British Columbia in Vancouver

system. It does support access from NFS mounts but clients accessing the file system in this manner can only access previous file versions and not create new file versions. This is due essentially to the fact that NFS is stateless while the Elephant system requires knowledge of current and previous client operations on opened files. For the purposes of this project the file versioning properties of the Elephant system are most interesting. When a user executes the first write system call after an open system call the system creates a duplicate inode for that file. Once the user executes a close system call Elephant then appends this duplicate inode to the files inode log[SFH+99]. As NFS does not support these calls and maintains no server side state information Elephant cannot under its present form support the creation of file versions from remote clients.

The same research team which produced the Elephant FileSystem are also working on the Mammoth FileSystem. The will also be a Versioned system but will have built in support for NFS clients from a variety of platforms. These clients will be able to create and access various file versions and Mammoth could provide a suitable core system for a future implementation of our cryptographic access control policy.

2.4.2 LinLogFS

LinLogFs is an ongoing project to develop a full Log Structured File System for Linux [24]. Its design goals were to provide for faster recovery times by logging all data and meta data updates and to allow for consistent and rapid backups of the file system to be made. This is accomplished by duplicating the checkpoint entry of an existing file system and mounting the old version in read only mode while performing the backup. While LinLogFs does support versioning, it does so at the file system level rather than at the individual file level[CE00]. This is too coarse grained to be of use in our design.

[24]It was begun by Christian Czezatke and currently is under development at the Vienna University of Technology

Chapter 3

Analysis and Design

3.1 Operational Environment

The CAC mechanism has been designed to operate in an open network where establishing the identity of the client is irrelevant to the secure operation of the server. The security policies employed in the server's access control mechanism are ideally suited to deployment in an environment where clients require rapid access to data and storage space that is not available locally. Mobile devices would certainly fit into this category.

Certain applications require that data be shared among a number of entities. Storing such data remotely facilitates the sharing of common resources. It should also be possible in such a scenario to give each client the capability to access this data in an efficient and secure manner while at the same time ensure that no unauthorized access to the data can occur.

Mobile devices can cross multiple administrative domains and may use services provided by a variety of vendors. If one of these services necessitates the use of a remote data storage unit then it should be possible for the user to have a high degree of confidence in the security of his data while having minimal confidence in the integrity of the remote data storage service which may be beyond his control.

As network transmission rates increase the bottleneck in a systems performance is often the processing of remote requests at either end of the network connection[Tan96]. The CAC Network File Server performs no client authorization and access list lookup actions when servicing read requests from clients. The server validation routines for the less frequent write requests[1] do however require validation of the clients public RSA key.

3.2 Cryptographic Access Control Model

CAC closely resembles a capability based system. Essentially all data is encrypted prior to storage and possession of the decryption key is necessary before data can be accessed in a meaningful fashion. The access control is therefore implicit rather than explicit. Any user may request access to a resource and his request will be satisfied, however unless the user possess the necessary information to decode the resource then he can make no effective use of it. This scheme possess the strengths and weaknesses of Capability systems but reduces the server response time by placing the greater burden of authentication procedures on the client side.

3.2.1 Read Access

The fact that all data is encrypted using strong encryption[2] prior to storage on the CAC server enables all client read requests to be implicitly authorized on the server side without recourse to any traditional form of access control. Unless a client possesses the decryption key he cannot decrypt and use the data. The file server will serve data blocks immediately on request from any client. This should result in shorter response times and decrease server processing.

When the client issues a read request the data returned is encrypted with the symmetric key associated with that file. This key is only stored on the client machine.

[1]See the file system usage patterns in [Sat81]

[2]128 bit symmetric cipher

The client will possess this key and can thus decrypt the data however it must also ensure that the data has not been tampered with on the server side or in transit. This is accomplished through the use of message digests. When servicing the clients read request the server also returns a hash of the file which has been signed with the private key of the owner. The client creates his own hash of the data, decrypts the hash supplied by the server and compares the two digests. If they match then the authenticity and integrity of the data has been established[O'S00].

3.2.2 Write Access

When a user wishes to store a new file on the server he must first encrypt the data using symmetric key cryptography. Next he must produce a hash or message digest of the encrypted file contents. The message digest is then signed using the private asymmetric key known only to the client. This message digest and the public key used by the client are then sent to the server along with the data to be stored. Upon receipt of the data and digest the server will generate his own digest from the encrypted data using the same algorithm. He then decrypts the client-generated digest using the file's public asymmetric key and compares the two digests. If they match then the write request is considered to be validated. The server stores the public key associated with the file and uses this stored key for validating future write requests from the user. If the data originates from a malicious client then the message digest cannot have been signed using the valid private asymmetric key and the write operation is considered invalid[O'S00].

When a user wishes to update the contents of a file which already exists the server, he encrypts the data and signs it as before. When the server receives the data from the user, it retrieves the public key associated with the file at create time and validates the data as before.

Write Validation for Large Files

We extended the CAC validation routines to deal with files larger than the underlying NFS block size. When a user wishes to write a large file to the server the file is broken down into packets which are the size of the underlying block size and written to the server in a series of RPC write requests. The first block is signed using the private key. The remaining file data is then signed by the client. When the server receives the first write request it validates the first block. If the validation is successful then it will buffer the remaining data until the entire file has been received. At this point it validates the second client signature for the remainder of the file. If the second signature is also valid the buffer is flushed to disk. Otherwise the data is flushed from memory and the update fails.

3.3 Access Control for an Alternative Architecture

Originally we had considered implementing the CAC model on a file system with Versioning capability. This would reduce the complexity of the server access control mechanism when dealing with client write requests. The amended server side validation for write requests is detailed below. The procedures for validating client read requests remains unaffected.

3.3.1 Write Access

As before when a client wishes to store a new or updated file on the server he must first encrypt the data using symmetric key cryptography. He then produces a hash or message digest of the encrypted file contents. The message digest is then signed using the private asymmetric key known only to the client. This message digest is then sent to the server along with the data to be stored. Upon receipt of the write request, the server can immediately service the request without establishing the identity of the client. If the data originates from a malicious client then the message digest cannot have been signed using the valid secret asymmetric key and future read operations by clients will detect the invalid data. It is not necessary for the server to generate its own digest of the data and compare it to the digest sent by the client as it is always possible for clients to request a valid earlier copy of the data, thus foiling the attempt to corrupt the system.

The complete lack of explicit access control policy on this Versioning server would however leave the system vulnerable to Denial of Service attacks by filling the disks with bogus data. We have not yet identified a suitable strategy to deal with this attack.

3.4 Trusted Computing Base

The principal components of the system are the client module,the key distribution mechanism, the network connecting the file server and client and the file server itself. One of the design goals for the CAC project has been to minimize the number of trusted components in the system. The prototype system developed requires that the user need only completely trust the client machine and the key distribution mechanism, while only minimal trust is required in the file server.

3.4.1 Trusting the Client

It is a fundamental requirement of our system that the user have complete trust in the client machine. All data stored locally is unencrypted and if a malicious agent succeeds in compromising the client machine then data confidentiality is destroyed. The keys necessary to read and update the users files are also stored on the local machine. If an attacker obtains these keys then the system has been fully compromised.

3.4.2 Trusting the Network

The system is designed to operate over a public unsecured network. All data is encrypted by the client module at the application level before the remote storage procedures are invoked. This ensures that all data is encrypted prior to being transmitted across the network. All data sent from the server to the client is also encrypted. The use of digital signatures protects against deliberate or accidental changes to the payload of the network packets.

This effectively means that data integrity and confidentiality are guaranteed and the user need not place any trust in the network. However the effectiveness of the system can be reduced by certain man-in-the-middle attacks which are discussed later.

3.4.3 Trusting the Server

We have already stated in our analysis of the operational environment that the server may be located in an administrative domain outside the control of the client. With this in mind the extent of the trust that the user must place in the server is a critical factor in determining the strength of the system. We have three principal requirements to meet before the storage of data on the server can be considered secure. We must ensure the confidentiality of the data, the integrity of the data and the permanence of the data.

Data Confidentiality

Data is stored in encrypted format and the server does not possess the keys necessary to decrypt the data. An assailant who compromises the server but who does not possess the necessary keys cannot access the data in a meaningful manner. All data stored on the server can thus be considered confidential.

Data Integrity

When a client sends data to the server it also sends the file's public key and a message digest of the data signed with the file's private key. This digital signature is stored with the file data on the server. Future read requests from the client result in the file data and digital signature being sent to the client. If the data has been altered on the server the signed digest will be invalid and the user will be aware that the integrity of the data has been compromised. This scheme becomes problematic when the complexities introduced by file-sharing and updating are examined. [3]. When a file is created the public key associated with that file has to be sent from the client to the server. The data packet containing the file name and key is also signed using the relevant private key. This key is stored on the server. This is effect guarantees that once a file creation request is executed on the server, the public key sent to the server at create time is the only one that the server will use in its validation routines.

Data Permanence

The CAC mechanism has designed to operate in a distributed system and among the defining characteristics of these systems are the potential for partial failure, the lack of global knowledge and the lack of centralized control. Given these difficulties it is a non-trivial task to devise a protocol which can guarantee that the server has correctly stored the client's data. At this point we must draw a distinction between the ability of the client to detect the failure of the server to correctly store the data and the ability of the client to guarantee that the server has correctly stored the data. In the previous section we outlined the procedure by which the client can detect tampering with the data. That this tampering could have occurred while the data was on the server or while it was in transit across the network is irrelevant. The data integrity model outlined above is sufficient to detect all attacks data integrity in

[3]See the Lost Updates Attack

a single user environment, however in a multi-user environment this model becomes problematic.

3.5 Threat Model

The following scenarios describe a series of possible attacks against our security model. In a scenario where updates are frequently made to shared data by a pool of users, a malicious server can in effect partition the user space and compromise the data integrity model.

Lost Updates Attack

1. User A creates a file and stores the signed digest and data on the remote server.
2. He distributes the keys needed to read and update file data amongst a circle of trusted users.
3. After a number of update operations on the file user A performs a read operation and the file data and signed digest are returned to him.
4. Once the digest is valid and has been signed with the private key user A assumes that the data is valid, however a malicious server can respond to user A's read request with an older version of the file and the signed digest associated with this older version. In this case this data integrity check will validate the file contents and user A is unaware that the data is out of date.

Man in the Middle Attack

1. User A creates a file and sends the public key to the server.
2. A malicious agent intercepts the packet, creates a new asymmetric key pair, sends the new public key to the server and signs it with the new private key.

3. All future requests from user A are intercepted by the malicious agent who can alter the data and sign it with his private key.

Once user A issues a read request to the server he will invalidate the message digest issued by the server and will therefore detect that his data has been compromised. However the attack has already succeeded in limiting the effectiveness of the system.

First Denial of Service Attack

1. This attack is very similar to the Man in the Middle attack. Again the malicious agent intercepts the create message containing the file's public key and switches it with his own key.

2. Future write requests from the client are not intercepted by the malicious agent but they fail validation on the server due to a mismatching of the asymmetric key pair.

The client will of course detect that his writes are failing but again this attack has limited the effectiveness of the system.

Second Denial of Service Attack

1. User A writes a file whose size is a multiple of the underlying block size to the remote server.

2. The malicious agent allows the first RPC message through. This contains a valid signature and the server will therefore decide to buffer all remaining data until the entire file has been received.

3. The malicious agent can then intercept all the remaining data and modify it.

4. Once the server receives the entire contents of the file, it performs the validation check on the entire file minus the earlier validated 8K chunk of data.

5. This second validation check will fail and the server will not write the data to disk. However a successful denial of service attack has been mounted.

In this scenario the client will detect that his write has failed. We have extended the CAC mechanism to counter this attack by using a sliding scale to determine the optimum data size to sign. In the scenario outlined above, the client will detect the write failure, he might then decide to reissue the write operation and sign the first 8K and every subsequent 48K of data, thus reducing the effectiveness of the attack. This sliding scale can be implemented transparently to the user, who will simply notice a degradation of performance.

3.6 System Design

We have designed and implemented a prototype network file system which uses the CAC mechanism. We have named this prototype network file system as the CNFS system. The architecture of the CNFS System as illustrated in Figure 3.1 is based on the traditional client/server model. If the CNFS server is implemented in user level space then two additional context switches are required between user and kernel level. Figure 3.1 illustrates the extra context switches incurred by this design compared to the design of Figure 2.1. The flexibility of a user level solution offsets the overhead introduced by these additional context switches. A user level CNFS server does not require root privileges or kernel recompilation to install. The ease of debugging user level applications coupled with the widespread availability of cryptography libraries enables more rapid development for user level as compared to kernel level applications.

The File Server design should be as simple as possible and the system intelligence should in the main reside on the client side. Ideally the File Server would simply

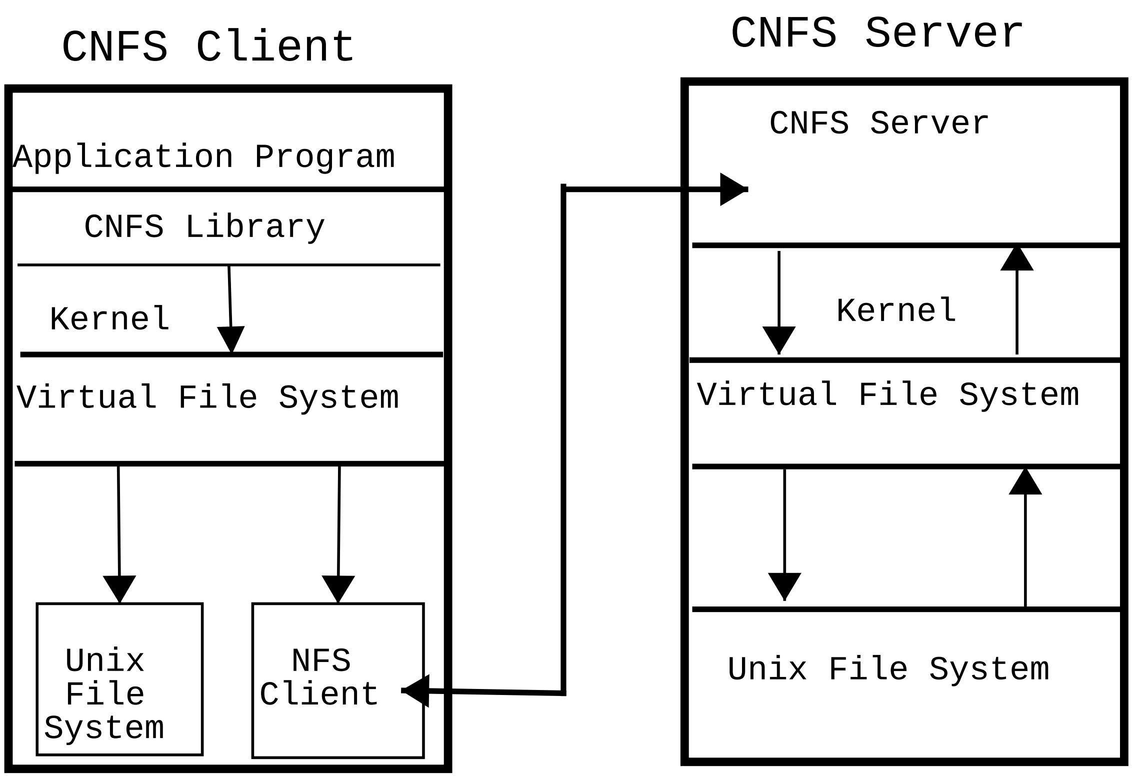

Figure 3.1: The Cryptographic Access Control Architecture

serve disk blocks on request from the client. The lack of centralized access control should decrease the response time. It is also a design goal that the minimum of trust be placed in the file server.

3.6.1 Server Architecture

Central to the design of the CNFS system is the observation that workstations have cycles to spare and that server cycles are precious [Tan95]. For this reason we place the greater burden of implicit access control for all read and write requests on the client side. The server will serve data blocks on request and unless the client possesses the required keys, the data will remain protected.

The NFS failure model detailed in the Sun specification requires that the server maintain no state information about client file operations and does not keep files open on behalf of its clients. All file operations are as far as is possible required to be idempotent. This allows for rapid recovery from server failures. The underlying block size for NFS version 2 is 8K. If a client wishes to write a file which is a multiple of the block size to a directory mounted on an NFS server, the client stub for the remote write procedure breaks the data down into 8K blocks and packs each one into an RPC write request message which is then sent to the remote server. Each execution of the write service routine on the server causes the file associated with the file handle to be opened, the data to be written to the appropriate offset in the file and the file to be closed again.[4] The NFS server is therefore never aware of the file as a logical unit itself. However the write validation routines implemented by the cryptographic access control mechanism requires the CNFS server to perform digital signature verification on the entire file before the data can be written to disk.

Adding State Information to NFS

Fundamental changes to the NFS write procedure were required to accommodate this functionality. When dealing with updates to a file which is a multiple of this block size we were forced to maintain state information on the CNFS server in relation to which clients were updating open files. It was also necessary to buffer data written from the client until the entire file was transferred across the network. The file name and client identity are established from information included in each RPC request header. This information is maintained in memory while a client has a server resident file opened in write mode on his machine.

When the client writes the first 8K block to the CNFS server, this block is validated by the CAC mechanism. If this operation is successful, state information in relation to file and client identity is maintained. Each subsequent 8K data block can then be buffered until all data is received from the client. The buffer is passed to the validation routines and ultimately written to disk or flushed from memory.

This also has the effect of delaying the write to disk operation and has important

[4]Use of file descriptor caching in some implementations my complicate this model

consequences for the NFS failure model. Buffering data increases the likelihood of cache inconsistency faults. The window of vulnerability associated with lost updates due to server crashes is also widened.

3.6.2 Client Architecture

The CNFS client should be able to mount a directory onto the server and store and access data remotely. All data to be stored on the server is encrypted on the client side prior to transmission across the network and all data to be read from the server is only decrypted on the client side. The full responsibility for ensuring the authenticity and integrity of file data rests with the client. Thus the client must also produce and validate hashes of all data stored on the server. In the CNFS system encryption is performed at the granularity of the file level. This raises some issues for key management and storage which will be examined later.

3.7 Use of Cryptography in CNFS

Cryptography is widely used in security systems today. Industrial strength cryptography is very computationally expensive, however the rapid increase in processing power has made the use of such technology feasible. Before a system designer chooses the particular ciphers most suited to the application or system being developed, he must first decide on the appropriate level in the system to place the cryptography functionality.

When deploying security care must be taken to ensure that transparency of utilization and the interoperability of system components are preserved. If we choose to employ encryption at the application level then we have to ensure that encrypted data output from one application can be accessed by another. The modification of all applications to deal with encrypted data is infeasible. It increases the potential for data to be compromised by reposing trust in an external agent such as the application developer, and would require the user to supply cipher keys to each process. A more sensible option would be to place cryptography in the operating system itself. Employing this functionality at the level of the file system itself enables applications and users to access protected data in a transparent fashion. Because all processes use a centralized set of encryption and decryption routines, it is easier to ensure that they are well designed and implemented. This is the approach we have taken in our system.

CNFS makes use of both symmetric and asymmetric cryptography. For performance reasons all file data is encrypted using symmetric or private key encryption. As stated earlier all encryption and decryption of file data is performed on the client side. Data transferred to or from the server is always sent in encrypted format. All file data stored on the server are encrypted and the decryption keys are never present on the server. This scheme has the advantage of requiring only the client machine to be secure. If the network or file server are compromised then the data is still safe. Public key cryptography is used to sign files and provides the means for ensuring the authentication and integrity of data operations.

3.7.1 Granularity of Encryption

We have chosen to employ encryption in the CNFS System at the granularity of the file level. This necessitates a high overhead for file generation operations on the client as a symmetric key and an asymmetric key-pair must be generated for each new file created. However the advantages offered by this approach, namely increased flexibility and security outweigh the disadvantages.

3.7.2 Key Management and Storage

The symmetric and private asymmetric keys for files are never stored on the file server. This is a fundamental requirement in ensuring that compromising the file server will not compromise the data stored there. These keys are to be stored on the client which owns or generated the files originally.

Key Distribution and Data Sharing

Key distribution is an important feature of any system using cryptography. In the case of CNFS sharing files is inextricably intertwined with key distribution. As files are meaningless without the appropriate keys the file owner must make the key available to whomever he wishes to share his files with. The Truffles file system solves this problem through the use of PEM as a key distribution mechanism[RPC+93]. Although this is a possible solution, the use of PEM to distribute keys mitigates against scalability in a large distributed system.

Studies of file system usage properties have noted that file sharing is unusual[Sat81]. It seems logical that sharing protected files among large numbers of users would be rarer still. Designing for the common case would argue that if file sharing is unusual, then we should not weaken system security by making file keys readily available. As the symmetric key and private asymmetric key are created and stored on the client machine, key distribution is at the file owners discretion. Our prototype does not currently support any mechanism to facilitate key sharing.

Chapter 4

Implementation

Although the concept of using cryptography as a form of access control was established in [O'S00], the actual design and implementation of the CAC mechanisms used in our CNFS prototype are entirely our work. As the implementation of the CNFS prototype proceeded it became necessary to extend the design of the CAC scheme. These extensions are also our own work. The principal components of the CNFS system are the File Server and the client library. Both these components have extensive cryptography modules.

4.1 CNFS Server Implementation

The original NFS server shipped with Red Hat Linux ran in user level space [1]. The obvious efficiencies gained from a kernel level server led to the migration of the core functionality of these daemons to the kernel from this point onwards. For ease of implementation we used the user level NFS server as the codebase for our prototype implementation. We obtained the source for the last version[2] of the user level Linux NFS server. The nfsd and mountd daemons provide the core functionality of NFS and CNFS. No modifications to the mountd daemon were necessary, however extensive

[1]The server daemons were principally the work of Rick Sladkey and Olaf Kirch

[2]Version 2.beta47

alterations to the nfsd daemon were required to implement our cryptographic access control mechanism.

4.1.1 Important Data Structures

The *fileInfo* structure is instantiated on the client and sent across to the server with the first 8K data block when performing a write operation. Because we are using a symmetric cipher and encrypting the file data in ECB [3] mode we need to pad the data to be a multiple of the cipher block size. For our prototype we have chosen a block size of 128 bits. The *padLen* variable stores the extra number of bytes required to pad the file to align on the cipher block boundary. The *initSignature* variable contains the message digest of the first 8K signed with the private asymmetric key. The *remSignature* variable contains the signed digest of the remainder of the file. To prevent tampering, the *fileInfo* structure is itself included in the data from which the first signed digest is generated.

```
typedef struct {
        int fileSize;
        int padLen;
        R_RSA_PUBLIC_KEY publicKey;
        BYTE initSignature[NFS_SIGNATURE_LEN];
        BYTE remSignature[NFS_SIGNATURE_LEN];
}fileInfo;
```

The *stateInformation* structure is instantiated on the server and is used to maintain state information about which files are currently being updated by clients. The *clntAddr* variable holds the IP [4] address of the client machine. The IP address is ob-

[3]Electronic CodeBook Mode

[4]Internet protocol

tained from the *svc_getcaller()* RPC library function which takes as a parameter the RPC service transport handle passed with the *NFSPROC_WRITE* call. The *fileBuf* variable is a pointer to a block of memory which is dynamically assigned when the first data block of a file is received from the client. This buffer is allocated enough memory to store the entire file and the *numBytesWritten* variable is updated once a data block associated with the file is processed. Only when the *numBytesWritten* and *fileSize* variable are equal, is the *remSignature* validated by the server. These structures are maintained in an unsorted linked list. A lookup operation on this list is an $O(\mathrm{n})$ operation, however this performance was deemed to be sufficient because the number of files opened at any one time on our prototype server is small. Once the file data has been written to disk or flushed from memory the list node is removed.

```
struct stateInformation {
        char fileName[MAX_FILENAME_LEN];
        struct in_addr clntAddr;
        char signature[NFS_SIGNATURE_LEN];
        int fileSize;
        int numBytesWritten;
        BYTE * fileBuf;
        struct stateInformation * nextPtr;
};
```

The *fileKeyNode* structure is used to hold the public keys of all files stored on the server. The *fNameHash* variable holds the normalized hash value of the absolute file path. When the server wishes to find a file's public key it reconstructs the file path from the filehandle and searches each *fileKeyNode* for a match. These structures are maintained in a binary tree which is sorted by the hash value. A lookup operation on this tree is an $O(\log \mathrm{n})$ operation. The extra efficiency here is necessary as the tree contains a node for each file on the CNFS server and can potentially contain a very large number of elements. As each new key is registered with the server an extra node is added to the tree, this information is also written to a file stored on the server so that they keys can be restored if the server fails. When the server is started it must first load all *fileKeyNode* structures from this file into memory.

```
struct fileKeyNode {
        unsigned long fNameHash;
        R_RSA_PUBLIC_KEY publicKey;
        int status;
        struct fileKeyNode * leftPtr;
        struct fileKeyNode * rightPtr;
};
```

4.1.2 NFS Server Modifications

The nfsd daemon provides the service routines for the seventeen RPC procedures listed in the NFS specification[Mic89]. Three of these procedures required substantial modification:

- *NFSPROC_CREATE*

  Additional functionality has been added to this procedure. When the nfsd daemon executes this call it creates a *fileKeyNode* entry in the binary tree used to map file names to their associated public key. The absolute path of the file is hashed and normalized to produce a value which is then used to determine that node's order in the tree.

- *NFSPROC_WRITE*

  This procedure has been completely rewritten. The underlying block size used by the CNFS server is 8k. As we outlined in the previous chapter this creates a problem when trying to validate the digital signature of a file whose size is a multiple of the block size. As NFS servers are supposed to be stateless and write operations idempotent, the unmodified NFS server had no concept of the file in its entirety. Originally a write to a file involved opening the file referenced by the file handle, writing the data block received from the client to the disk and closing the file again.

The modified write procedure establishes client identity, validates the first 8K of data and buffers the remaining file data until all data has been received. If at this point the server validates the remainder of the file, it issues a local open system call and writes the data to disk.

- *NFSPROC_READ*

  The only changes to this procedure has been to remove as much of the authentication checks as was possible. The server caching and authentication modules were closely intertwined in the Linux NFS server codebase that we used. Because we rewrote the *NFSPROC_WRITE* procedure we were unable to use the original server caching mechanism. In the *NFSPROC_READ* procedure we tried to retain the caching mechanism and just remove as many of the authentication hooks as were possible.

CAC Validation Routines

We have implemented the core functionality used by the CAC write validation mechanism in the *doSecureCNFSWrite()* procedure. This procedure is invoked from within the *NFSPROC_WRITE* procedure and can operate in three distinct modes. The functionality of the procedure varies according to the mode it is invoked with. These modes are:

- WRITETHROUGH

  This mode is invoked when the size of the file being written is less than the underlying CNFS block size of 8K. All files smaller than this limit only require one digital signature validation, and therefore there is no need to buffer such files prior to writing to disk. If a signature is verified with the server public key associated with he file then this validation procedure returns a *VALID_SIGNATURE* flag and the data is written to disk.

- BUFFER

  This mode is invoked when the data block being written to disk is the first block of a file which is a multiple of the underlying CNFS block size. The initial signature is validated with the server public key associated with the file. If the signature is valid then the procedure returns an *AUTH_CACHE* flag. A buffer large enough to hold the file contents is then allocated and a *stateInformation* entry is added to the in memory list of open files and clients. Future data blocks received up to and including the penultimate data block are then simply appended to the file buffer by the *NFSPROC_WRITE* procedure. When then last data block is received the *doSecureCNFSWrite* procedure is invoked with the FLUSHBUFFER flag set.

- FLUSHBUFFER

  This mode is invoked when the last RRC write request is received for a file whose size was greater than the CNFS block size. When this mode is invoked the file contents are loaded from the file buffer. The last data block is appended and the digital signature of the file contents minus the initial data block is validated using the server public key associated with the file. If the signature is validated a *VALID_SIGNATURE* flag is returned and the data is written to disk.

4.2 Client Implementation

The current NFS[5] client is implemented in the Linux kernel. The cryptography libraries we wished to use ran at user level so for ease of implementation we decided to encapsulate the functionality of our CNFS client in a user level library which other applications can be built against. We provide procedures which are similar to the open, read, write and close system calls in Linux. The use of cryptography to provide access control, data integrity and confidentiality is transparent to the user.

4.2.1 Client Library

The client library provides the interface to the CNFS system. This interface consists of four procedures:

- sopen()

  This procedure takes as arguments the name of the file to be opened and the mode in which it is to be opened. If the mode is CREATE, routines are invoked to produce a symmetric key and an asymmetric key pair. These keys are stored in a file in the users *HOME* directory. Only the public asymmetric key is sent to the server. An open system call is executed in the directory mounted onto the remote server. This culminates in an *NFSPROC_CREATE* procedure being executed on the remote CNFS server. The sopen procedure returns the file descriptor associated with the remote file.

- sread()

  This procedure takes as arguments a file descriptor, a file name and a pointer to a buffer to store the data. The file name is necessary as the CNFS client must perform a lookup operation for the cryptography keys which are associated with that file. The procedure loads the keys and issues an *NFSPROC_READ*

[5] Version 3

call to the CNFS server. The encrypted file data and signatures are returned. If they are valid, the data is decrypted and passed up to the user.

- swrite()

  This procedure takes as arguments the file name, a pointer to the data to be written and the size of the data to be written. This procedure buffers all data on the local cache until the sclose procedure is invoked.

- sclose()

  This procedure takes as arguments the file name and the file descriptor. It loads the cryptography keys for the file and encrypts the plaintext data stored locally with the symmetric cipher. The plaintext data is padded if necessary to ensure that the plaintext length is a multiple of the cipher block size . If the file size is smaller than the underlying CNFS block size it creates one signature and sends the data and signature to the server. If the file size is a multiple of the underlying block size then it signs the first block and sends it to the server. The remaining data is then signed and sent to the server. Upon completion the local plaintext version is removed.

  This client library provides the user interface. The client library also contains procedures for key and local cache management as well as sizable cryptography modules which will be described later.

4.2.2 Gnome ToolKit

In order to effectively demonstrate the prototype a sample application was built against the client library. We obtained the source for a text editor application which was built using glade and GTK [6]. References to the Linux file system calls were replaced with references to the interface provided by the CNFS client. This sample application provided a clear demonstration of the functionality of the CNFS system

[6]Gnome ToolKit

and the ease with which applications can be built against the CNFS client library.

4.3 Cryptography

The CNFS system employs both symmetric and asymmetric ciphers to ensure data confidentiality. Hashing functions are also used to guard against attacks on data integrity. Symmetric cryptography is used to encrypt the file data. The authentication mechanism relies on the asymmetric digital signature verification routines.

4.3.1 Symmetric Cryptography

In 1997 the NIST [7] announced the AES[8] program. They were searching for a block cipher to replace the DES [9] standard. One of the finalists in the selection process was the Twofish cipher developed by Bruce Schneier and colleagues at Counterpane systems. This is the cipher we have chosen for use in implementation of our prototype.

Twofish Cipher

Twofish is a 128 bit symmetric block cipher which supports key lengths up to 256 bits[SKW+98]. It is designed to be flexible and efficient on a variety of platforms varying from 64 bit Sparc processors to 32 bit Intel machines. It is essentially a 16 round Feistel Network. The fundamental building block of a Feistel Network is an F function which performs a key-dependent mapping of an input string onto an output string[SKW+98]. Feistel Networks basically transform any function into a permutation, and are the basis of most block ciphers including DES.
Twofish has been shown to be an efficient cipher[SKW+98] and to date no significant success have been reported in cryptanalysing the cipher [10]. We obtained the reference

[7] National Institute of Science and Technology

[8] Advanced Encryption Standard

[9] Data Encryption Standard

[10] See [MM99] for details of key analysis attacks

implementation which was submitted to the AES panel. The API is not very clear and the key generation routines are rather obscure but it proved to be adequate for our purposes. The Twofish API only supported ECB mode, however this was the most suitable mode for our prototype as we wished to preserve byte boundaries within the encrypted files. The key size we are using for our prototype is 128 bits. This was deemed sufficient for security in the envisaged operational domain. Our tests indicated that the key length can be increased to 256 bits and the CNFS system will still operate in an efficient manner.

4.3.2 Asymmetric Cryptography

In the CNFS system asymmetric cryptography is only used to encrypt the message digests generated from the contents of the encrypted files. The CNFS write validation routines decrypt the digest using the public key and compare it to the file contents received across the network.

RSAEuro Library

The RSAEuro library we used provides a mature and extensive API with support for RSA encryption, decryption, key generation, and the generation and verification of Message Digests using MD2, MD4 and MD5. We use the MD5 hashing algorithm to generate 128 byte digests of the file data which are signed with the private asymmetric key. The asymmetric key length used in the prototype implementation is 1024 bits. Tests showed that the 2048 bit key size introduced prohibitive increases in file processing times.

Chapter 5

Evaluation

This chapter presents an evaluation of the performance and functionality of the CNFS prototype. The overhead introduced by the cryptography routines is measured and the performance of the file operations in our prototype is also examined. The measurements of the unmodified NFS server which provided the codebase for our prototype are included for comparative reasons. These results are discussed in the concluding section of the chapter.

5.1 Test Setup

The following tests were all carried out on a Pentium Pro 233 MHz machine with 128 MB SDRAM. For tests involving the prototype CNFS system, the user level CNFS server and client were run on the same machine. A local directory was mounted and requests for file operations in this directory were handled by the CNFS server. The same setup was used for tests on the unmodified NFS server. The machine was lightly loaded during the tests. Each test was carried out twenty times and the mean time in milliseconds and standard deviation for each test are shown.
Caching strategies play a vital role in determining the performance of NFS. For the purposes of our tests caching was disabled on the client and server side. It is also worth noting that Linux provides asynchronous write operations, this explains

why the write operation times displayed in the results are much less than the read operation times.

5.2 Cryptography Routines

5.2.1 Symmetric Key Creation

A key length of 128 bits is used in the prototype implementation. The Twofish cipher supports key lengths up to 256 bits in length. Table 5.1 shows the results of the key generation operations. The C library *rand()* function is used to generate the key material. The random number generator is seeded with the system time. The overhead of key creation is borne by the CNFS client. Table 5.6 lists the average time required for the unmodified NFS client to execute an *NFSPROC_CREATE* procedure on the test system as 28.40 ms. The extra overhead averaging at 0.58 ms introduced by symmetric key creation only represents a 2% increase in the time taken to complete an *NFSPROC_CREATE* procedure.

5.2.2 Asymmetric Key Creation

A key length of 1024 bits is used in the prototype implementation. The RSAEuro libraries require only minimal changes to use a key length of 2048 bits. However the performance penalty incurred by generating 2048 bit keys in our prototype was too great. As is the case with the symmetric key generation, this operation is only performed on the client. Table 5.1 displays the results of the key generation operations. Table 5.6 lists the average time required for the unmodified NFS client to execute an *NFSPROC_CREATE* procedure on the test system as 28.40 ms. The extra overhead averaging at 3228.44 ms introduced by asymmetric key creation represents a significant increase in the *NFSPROC_CREATE* procedure.

Key Creation	Symmetric	Asymmetric
Mean Time in Millisecs	0.58	3228.44
Standard Deviation	0.01	663.95

Table 5.1: Key Generation

5.2.3 Twofish Symmetric Cipher

The Twofish cipher routines used in the prototype implementation were run in ECB mode. The measurements given in this section are for Twofish cipher operations on memory resident data only. The overhead of loading the data into memory is not factored into the figures. The Twofish routines are never executed on the server. The overhead introduced by these routines is borne solely by the client. Two file sizes were chosen for the tests. The first size is 8K, the underlying NFS V2 and CNFS data block size. The second file size of 1M was chosen as an arbitrarily large file size. The overhead of the symmetric routines is again borne only be the CNFS client. A sample of the encryption times for the 8K and 1MB file sizes are shown in Table 5.2. These times represent a significant overhead on the unmodified NFS client write operation times listed in Table 5.6. However as the size of the data to be written and encrypted increases the weighted cost of the encryption operation decreases.
A sample of the decryption times for the 8K and 1MB file sizes are also shown in Table 5.2. The overhead introduced by the decryption routines is of the order of 21% when compared to the unmodified NFS client read operation times listed in Table 5.6.

5.2.4 RSAEuro Ciphers

The RSAEuro library provides an extensive API for users. Our prototype uses the key creation, encryption, decryption and hashing routines. Again the file sizes tested were 8K and 1MB. As in the previous section the measurements given in this section

Twofish Cipher	Encryption of 8K Block	Encryption of 1MB Block	Decryption of 8K Block	Decryption of 1MB Block
Mean Time in Millisecs	6.62	835.15	6.55	833.35
Standard Deviation	0.01	7.74	0.03	4.72

Table 5.2: Twofish Cipher Performance

are for operations on memory resident data only. The RSAEuro library contains optimized routines for this scenario. As in the previous section the overhead of loading the data into memory is not factored into the figures. The RSAEuro routines are executed on both the client and server.

A sample of the digital signature generation times for the 8K and 1MB file sizes are shown in Table 5.3. This operation is performed on the CNFS client prior to writing data to the remote CNFS server and represents a significant overhead for the client when compared to the figures in Table 5.6 which represent the average time for a write operation on an unmodified NFS client. As the size of the data to be encrypted increases toward 1MB the overhead of this operation decreases to about 19%.

A sample of the digital signature validation times for the 8K and 1MB file sizes are also shown in Table 5.3. This procedure is performed on both the CNFS server and CNFS client during the *NFSPROC_WRITE* and read operations respectively. It represents an overhead of 72% on the unmodified NFS client read operation for a file of size 8 K. However this overhead decreases to 3% for a file of size 1 MB.

Digital Signature	Generation 8K	Generation 1MB	Validation 8K	Validation 1MB
Mean Time in Millisecs	414.55	499.37	22.30	111.98
Standard Deviation	3.78	1.92	0.72	1.01

Table 5.3: Digital Signature Cipher Performance

5.3 Performance of CNFS Server

In this section the performance of the CNFS server is compared with the performance of the unmodified user level NFS server. The codebase of the unmodified server was the one used to develop the CNFS server. The tests were also carried out on files of size 8K and 1 MB. The underlying data block size for both the NFS and CNFS servers is 8K. The 1 MB size was chosen as an arbitrarily large file size. The measurements below measure the execution time of the *NFSPROC_CREATE*, *NFSPROC_OPEN* and *NFSPROC_WRITE* procedures on the server only. TCP stack and XDR operation times are not included.

5.3.1 Read Performance

The user level Linux NFS server codebase which we used had closely integrated the access control and request caching functionality. The modifications required to the *NFSPROC_READ* operation on the CNFS server involved removing the ACL lookup operation. Some weakening of the caching strategy was unavoidable. This is borne out in the results displayed in Table 5.5 and Table 5.4 which display the times required for read operations for files of size 8 K and 1 MB on the CNFS and unmodified NFS server respectively. The CNFS read operation on the 8 K file is 20% faster than the read operation on the unmodified NFS server, however the read operation on the larger file was 10% slower on the CNFS server.

NFS Server	Read 8K	Read 1MB	Write 8K	Write 1MB
Mean Time in Millsecs	0.79	792.84	0.89	1187.89
Standard Deviation	0.09	83.11	0.11	156.30

Table 5.4: NFS Server Routines Performance

CNFS Server	Read 8K	Read 1MB	Write 8K	Write 1MB
Mean Time in Millsecs	0.63	874.168	23.062	1480.16
Standard Deviation	0.02	150.85	0.25	222.16

Table 5.5: CNFS Server Routines Performance

5.3.2 Write performance

The *NFSPROC_WRITE* procedure on the CNFS server was completely rewritten. All existing validation and caching were removed and digital signature validation routines were added. The unmodified server writes each data block to disk as it is received. The CNFS server buffers individual data blocks until the entire file has been received. At this stage a digital signature is validated and the contents are written to disk or flushed from the buffer. This effectively renders all comparisons between write operations requiring buffering on the CNFS server and standard write operations meaningless. However one case is of interest. In the case of a write operation for a file of size 8 K on the CNFS server, the data is validated and written to disk or discarded without buffering. As expected the overhead of validating the digital signature introduces a significant overhead on the CNFS server.

NFS Client	Open	Read 8K	Read 1MB	Write 8K	Write 1MB
Mean Time in Millisecs	28.40	307.57	3527.51	0.35	2700.10

Table 5.6: NFS Client Performance

5.4 Discussion

The cryptographic access control mechanism requires that each request to write data be accompanied by the relevant digital signatures and public key. This information is sent to the server and must be stored on the server for the purposes of validation. We have encapsulated this data into a structure of size 780 bytes. This represents a

considerable overhead when small temporary files are being written to the server.
One of the goals of this project was to decrease the load on the server and decentralize the access control mechanism. This has been achieved for the CNFS read operation. The performance of this operation is 20% faster than the unmodified NFS server read operation for small files. These results are logical given that no access control mechanism is required on the server.
It is much more difficult to compare the performance of the write operation on the CNFS and NFS servers. The overhead of digital signature validation is greater than the overhead of an ACL lookup when processing small files. However when processing files greater than the underlying data block size of 8K the use of data buffering on the CNFS server renders comparisons with the unmodified NFS server meaningless.
In order to conduct a more realistic evaluation of the CNFS server it would be necessary to modify an NFS server so that it would also perform encryption and decryption of data. As discussed in Chapter 2, the AUTH_KERB5 security flavor encrypts and decrypts data prior to and after transmission of data from client to server. The CNFS Server should be faster than an NFS server using AUTH_KERB5 as it does not decrypt data prior to writing it to disk. It also offers its users a stronger and more flexible security model.

Chapter 6

Conclusion

6.1 Objectives Fulfilled

In the first chapter we stated that our primary objective was to build a prototype network file system which would use the CAC mechanism. This prototype would be evaluated to determine if the CAC mechanism was suited to deployment in a modern distributed computing environment. The challenges posed by this operational environment for security systems were identified as follows:

1. Access control mechanisms are required to protect resources in systems spanning multiple administrative domains. Changing user behaviour patterns and the growth of mobile computing require client machines to use the services of remote elements in the system. As these remote elements are often beyond the control of the user, the trust reposed in these elements must be minimized.

2. Roaming users may not wish to create accounts on remote servers or pay the penalty of heavy authentication before they can access or store data on these servers. Security policies should accommodate these users.

3. Centralized access control mechanisms can become a bottleneck in large distributed systems with many thousands of users. Ideally access control functionality should migrate from the server toward the client machine.

The first challenge has been met by the integration of industrial strength cryptography with file system operations. We have identified that user authentication, data confidentiality and data integrity are central requirements in any security mechanism. Before data is stored on a remote server it is encrypted on the client machine. This allows the user to place no trust in the transmission medium or remote server to guarantee the confidentiality of his data. The CAC mechanism uses public key cryptography and digital signatures to guarantee data integrity. Cryptography keys are used as capabilities by the CAC mechanism to provide support for user authentication.

The second challenge has been met by using cryptography keys are capabilities. If a data packet arrives at the CNFS server and is signed with a valid private key then the server will write the data and digest to disk. The server never establishes the identity of the user, it merely verifies that the correct key was used to sign the data. When the server receives a read request from a user it immediately responds with the requested data and digest. No authorization is required as the data is unintelligible to a user not possessing the appropriate keys.

The third challenge has been partially met by decentralizing the access control policy for all read requests. The CAC mechanism places the onus of verifying data integrity on the client side. The client must validate the digest with the public key to ensure its validity. However write requests still require validation on the CNFS server to prevent valid data being overwritten. We have suggested that a Log Structured or Versioning file system could be used to prevent the overwriting of valid data and allow for the complete decentralization of the access control mechanism.

We have succeeded in building our prototype CNFS server from our Linux NFS codebase and have successfully deployed the CAC mechanism. The primary challenge faced during the prototype implementation has been to integrate state information into the CNFS server. This was necessary to allow the CAC mechanism handle larger files efficiently. This has been accomplished through a combination of RPC header information, NFS filehandle and CNFS client information.

6.2 Related Work

This project has built on the work carried out by Declan O'Shannahan in his Msc Thesis [O'S00]. He designed the CAC mechanism and implemented the CAC mechanism in a kernel level network file system. Kernel level file systems require less context switches than user level file systems however they lack portability and require root access to install and configure. The lack of cryptography libraries and the difficulty of code debugging, complicated the kernel level implementation and made it difficult to perform extensive testing of the system.

The experience gained from this previous work led to a decision to implement the CAC mechanism in a user level network file server. This meant that we had access to extensive cryptography libraries which reduced the complexity of developing the security modules.

We also wished to identify the changing requirements placed upon security systems in a changing operational environment and perform a rigorous examination of the suitability of the CAC mechanism for use in a large distributed system.

One of the challenges we faced in building from the previous work was modifying the CAC mechanism to efficiently handle files larger than the underlying network file system size. Our solution required fundamental alterations to the operation of the NFS protocol and extending the functionality of the CAC mechanism.

Finally the CNFS prototype developed allowed us to analyse the performance of the CNFS Server routines.

6.3 Future Work

Our prototype is fully functional and has provided a platform to test the effectiveness of the CAC mechanism. However there are several areas in which further work could be carried out.

- As we state earlier we had initially intended to implement the CAC mechanism on a Versioning file system. This would have allowed complete decentralization of the access control mechanism. The Mammoth distributed file system project[1] may provide a suitable core system for a full implementation of the CAC mechanism.

- The CNFS prototype currently lacks a secure key distribution mechanism which is necessary to facilitate file sharing. This issue has not been examined in detail in this thesis and could prove a fruitful area for future research.
- The system is vulnerable to the *Lost Updates* and *Man in the Middle* attacks identified in Chapter 3. Future work could be undertaken to address these issues.
- The CNFS client is currently implemented as a user level library against which other applications can be built. This has the advantage of being very portable and easy to install. However these advantages are obtained at the cost of transparency to the user. Investigation could be undertaken to determine if the advantages of migrating this functionality to the client kernel are warranted by the difficulties entailed by such action.

6.4 Conclusion

The rapid growth of mobile computing and the deployment of large distributed systems which span multiple administrative domains has created a series of new challenges for security systems. Existing access control mechanisms need to be adapted to reflect the changing operational environment. We have described these challenges and have offered a critique of the suitability of the CAC mechanism for use in modern distributed systems.
We have found the CAC mechanism to be flexible and scalable yet powerful enough to effectively protect resources in unsecure environments.

[1]Currently under development at the University of British Columbia in Vancouver

Bibliography

[Bla93] Matt Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

[CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design*, chapter Distributed File Systems, pages 323–335. Addison-Wesley, Harlow, 2001.

[CE00] Christian Czezatke and M. Anton Ertl. LinLogFS — a log-structured filesystem for Linux. In *Freenix Track of Usenix Annual Technical Conference*, pages 77–88, 2000.

[CP97] G. Cattaneo and G. Persiano. Design and implementation of a transparent cryptographic filesystem for unix, 1997.

[Eis99] M. Eisler. Nfs version 2 and version 3 security issues and the nfs protocol's use of rpcsec gss and kerberos v, 1999.

[FK92] David Ferraiola and Richard Kuhn. Role-based access control. In *15th National Computer Security Conference*, pages 1–11, 1992.

[GHM$^+$90] R. Guy, J. Heidermann, W. Mak, T. Jr, G. Popek, and D. Rothmeier. Implementation of the fiscus replicated file system, 1990.

[Gol99] D. Gollmann. *Computer security.* John Wiley and Sons, 1999.

[Jen00] Christian Jensen. Cryptocache: A secure shareable file cache for roaming users, 2000.

[Mic89] Sun Microsystems, Inc. NFS: Network file system protocol specification. *Internet Request for Comments*, (1094), 1989.

[MM99] F. Mirza and S. Murphy. An observation on the key schedule of twofish, 1999.

[O'S00] Declan O'Shanahan. Cryptosfs: Fast cryptographic secure nfs, 2000.

[RPC$^+$93] P. Reiher, T. Page, S. Crocker, J. Cook, and G. Popek. Truffles—a secure service for widespread file sharing, 1993.

[Sat81] M. Satyanarayanan. A study of file sizes and functional lifetimes, 1981.

[Sch94] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source-code in C.* John Wiley and Sons, New York, 1994.

[SFH$^+$99] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *Symposium on Operating Systems Principles*, pages 110–123, 1999.

[SKW$^+$98] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, and C. Hall. Twofish: a 128-bit block cipher, 1998.

[Sri95a] R. Srinivasan. Rpc: Remote procedure call protocol specification version, 1995.

[Sri95b] Raj Srinivasan. XDR: External Data Representation Standard. Technical Report 1832, 1995.

[Ste99] W. Richard Stevens. *Unix Network Programming*, chapter Sun RPC, pages 399–452. Prentice Hall Inc, New York, 1999.

[Tan95] Andrew S. Tannenbaum. *Distributed Operating Systems*, chapter Distributed File Systems, pages 272–279. Prentice Hall Inc, New York, 1995.

[Tan96] Andrew S. Tannenbaum. *Computer Networks Third Edition.* Prentice Hall, 1996.

[U.S83] U.S. Department of Defense. Trusted computer systems evaluation criteria, 1983.

[ZBS] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system.