CryptosFS: Fast Cryptographic Secure NFS

Declan Patrick O'Shanahan

A dissertation submitted to the University of Dublin,

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

2000

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: _____

Declan Patrick O'Shanahan

15/9/2000

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _____

Declan Patrick O'Shanahan

15/9/2000

Acknowledgements

I would like to thank Christian Jensen for all his help and support throughout the year. I would also like to thank my parents Liam and Peggy O'Shanahan for their continued support. Thanks also to Paul for his patience and assistance over the year, especially for his editing skills. Special thanks to Edyta for putting up with the absence enforced by my decision to do the Masters.

Summary

The issue of security in file-systems is as relevant today as when the first file system was developed. Current file system implementations rely heavily on centralised security mechanisms such as access control lists. The problem of security in file systems was made more complicated by the introduction of remote access to files. Storing information on a remote server has the potential to introduce additional security weaknesses into the file system model. The client, the communication links and the server make up the file system model.

The Network File System (NFS) is a widely used and oft maligned file system. Developed by Sun Microsystems in the 1980s it introduced a means to access files remotely. It is by no means the only distributed file systems but it is one of the most widely used. Serious security limitations were identified in the NFS protocol, as the original design did not include a security aspect. Security was added to the NFS protocol by the introduction of secure RPC. The security added was in the form of authentication of users. The distributed file system model that NFS uses is susceptible to attack in the following ways.

1. An attacker who can gain control of the NFS client has the ability to read data and can compromise the confidentiality of the data. If the NFS client has write access, an attacker can also compromise the integrity of the data stored on the server.

2. An attacker who can gain access to the NFS server can compromise the confidentiality of the data stored on the server. The attacker can also compromise the integrity of the data by modifying the data stored on the server.

3. An attacker who can gain access to the network can compromise the confidentiality of data passing over the network. If a client is performing a write operation, the attacker

can modify the data associated with the write operation and affect the integrity of the operation. The authenticity of information passing between a client and a server is not guaranteed as an attacker who can compromise the integrity of the information can also compromise the authenticity of the information by modifying the data on the fly.

CryptosFS is a distributed file system prototype that uses a combination of cryptographic techniques to provide confidentiality, integrity and authenticity of information. Blowfish symmetric-key cryptography is used to encrypt file system data and meta-data. The symmetric-key cryptography provides information confidentiality. Asymmetric-key cryptography and MD5 message digests are used to create digital signatures. Validation of the digital signatures provides authentication and integrity.

Authenticity and integrity are ensured by the validation of digital signatures by the NFS server. The NFS server possesses the public-key for each file allowing it to verify read and write requests received from clients. Integrity of the information on the remote server is preserved by not storing the symmetric-keys to encrypt the file data on the server.

Table of Contents

Table of Figures

Table of Tables

Abbreviations

ACL	Access Control List
ACE	Access Control Entity
CFS	Cryptographic File System
DES	Data Encryption Standard
GNU MP	GNU Multi Precision Arithmetic Library
IP	Internet Protocol
LIP	Large Integer Package
NFS	Network File System
PDA	Personal Digital Assistant
PEM	Privacy Enhanced Mail
RBAC	Role Based Access Control
RPC	Remote Procedure Call
SMS	Short Messaging Service
TCB	Trusted Computing Base
TCP	Transmission Control Protocol
TCFS	Transparent Cryptographic File System
UFS	UNIX File System
UMTS	Universal Mobile Telecommunications System
VFS	Virtual File System
WAP	Wireless Access Protocol
XDR	External Data Representation

1 Introduction

The Internet has seen tremendous growth since its commercial inception early in the 1990's. New techniques are being continuously developed to access information over the Internet. The last couple of years have seen an explosion in the number of people using hand held devices such as personal digital assistants (PDA's) and mobile phones. Convergence of voice and data services is starting to happen with the increased use of short messaging service (SMS) and the wireless access protocol (WAP). Connection by users to information and services stored on remote computers is on the increase.

Current GSM technology allows users to roam in a foreign network and to use the services on that network. The development of the third generation of mobile services Universal Mobile Telecommunications System (UMTS) and broadband services can only increase the numbers of users utilising mobile communication services. As data-services become increasingly important in the future, users will require access to information on remote computers safe in the knowledge that the information is secure from compromise.

The ability of an individual to securely access information on a remote machine is increasingly necessary with the increased usage of the Internet in every day life. The ability to share the information with other users and provide them with differing levels of access to the information is even more desirable. Current schemes for accessing remote information require registration and significant amounts of infrastructure to allow the information to be shared among users. When information is stored on a remote machine, the need to store passwords or keys on the remote machine makes it vulnerable to attack. An ideal remote information repository is one where the

information is stored in encrypted format. This means that the information is safe from all but sophisticated cryptanalysis. The use of encryption ensures security of the information as it makes it way from the local to the remote host avoiding problems of insecure communication links.

CryptosFS investigates the use of public-key cryptography to replace the access control mechanisms of NFS [1]. Traditionally NFS uses secure RPC [2] to provide authentication of users. This does not provide integrity or security of information stored on a server. CryptosFS replaces the existing access control mechanisms of NFS with asymmetric-key cryptography to allow users access information. Possession of the private-key allows a user to create, modify and remove files and directories. Correspondingly, possession of the public-key allows a user to read the files and directories.

Encrypting the file data and file meta-data with symmetric-key cryptography provides information confidentiality. The MD5 [3] algorithm is used in combination with asymmetric-key cryptography to produce digital signatures. The digital signatures are used as a form of access control that allows the server to validate clients requests for operations. The public-key of the asymmetric-key pair is used to create a digital signature for read operations, while the private-key is used to create a digital signature for write operations. This provides the integrity and authentication of the information on the server.

Only clients who possess the correct keys can validate themselves to the server to perform the read and write operations. The ability to distribute the public or private keys allows a user determine the level of access to grant to other users. Using asymmetric-key cryptography for access control allows arbitrary sharing of files

among trusted users. This form of access control provides strong security and is a distributed form of access control as the asymmetric-keys can be shared with other users. A distributed access control mechanism scales better than a centralised form and is not a single point of failure.

1.1 Granularity of File Encryption

Encryption of data in the file system is one of the most important aspects of the research. The level at which the encryption is to be applied to the file system data has important implications in terms of performance and security. By choosing to encrypt file system data at the individual file level, it provides a sufficiently low level of access control but more importantly, it also provides a higher level of security. Without individual keys for each file, it is not possible to replace the access control mechanism of the file system. Creating keys for each file increases the security of the file system by reducing the impact of a key being compromised to a single file.

It is easier to generate keys to encrypt the contents of a directory rather than each individual file. Several of the existing cryptographic file systems encrypt all of the file data for a single user with one key. Typically, a user is prompted for a pass-phrase that generates a key to encrypt the data. Relying on a user tool to generate the password leaves the file system vulnerable to attack. An attacker can replace the key generation program with a Trojan horse that stores a copy of the keys generated for each user. Having the file system handle the keys transparently for the user eliminates this threat.

1.2 An Overview of File System Development Options

The idea of writing a file system from scratch is not practical given the amount of time and resources available for this thesis. Research of the available options to perform the development was important. It allowed us to identify where our proposal

fitted in with relation to current and past file system development. Our experience with file system development is limited so the research served to identify the different techniques that can be used. The variety of file systems encountered show what an active area of research file system development is. The file systems researched fell in to one of the following three categories:

1. The file system is developed in user space and runs as a user process.

2. The file system is developed in the kernel and runs as a privileged process. The file system development implements all of the file system functionality.

3. The file system is developed in the kernel and runs as a privileged process. The new functionality that the file system provides is stacked on top of existing file system functionality using stackable layers.

Each of the three techniques outlined have peculiarities and advantages that are described in the following sections.

1.2.1 User Process File System

The idea of developing a file system as a user process is appealing for a variety of reasons not least of which being that it is simpler than other techniques. By developing the file system as a user level process, the complexity of kernel level programming can be avoided. This simplifies the development process enormously, as developing in the kernel is more restrictive than user level development. The standard development, debugging tools and programming libraries can be used. This helps to reduce the time required to implement the file system.

One of the most advantages of developing a file system as a user level process is that the file system can be installed by a user without the assistance of a system administrator. This provides the user with greater flexibility in how they use files. Figure 1 illustrates how a file system developed to run in user space interacts with the local and remote operating systems.

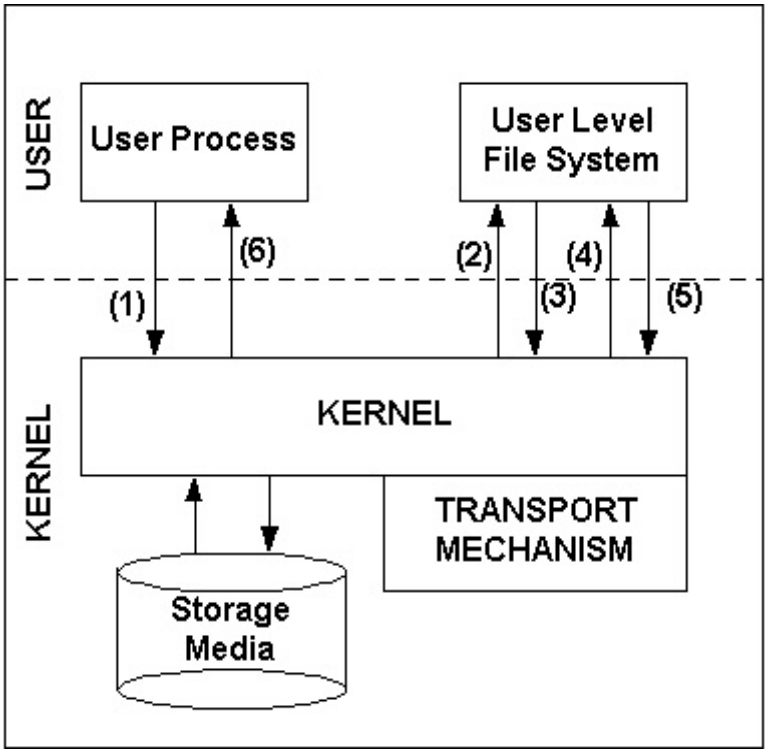

Figure 1: File System Implemented as a User Level Process.

A user process requests access to a file from a user-space file system. The request is routed through the kernel. The steps in the communication show how a request by a user process results in a context switch in to and out of the kernel. Starting at (1) the user makes a request to read a file. This results in a call to the kernel that forwards the call on to the user level file system (2). The user level file system makes another call to the kernel (3) to retrieve the data required by the read from the storage media.

The kernel passes the data back to the user level file system at (4). The user level file system now calls the kernel again to pass the data back to the user process (5). The

kernel completes the read command by delivering the data to the user process at (6). This results in two additional context switches to the kernel than a normal read.

There are many examples in the literature of file systems developed as user processes; these include CFS [4] and TCFS [5]. Please refer to section 2 for more a more detailed evaluation of these file systems. All of the file systems that are implemented as a user level process are susceptible to a major performance problem. The use of a user level process requires additional context switches that increase the overhead of every system call and thus reduce performance.

1.2.2 Kernel Level Process File System

To develop a file system in the operating system kernel means forgoing the simplicity of development that a user process provides. This increases development complexity because kernel level programming requires specialist knowledge of the specific operating system being used. When the file system resides in the kernel, a tight coupling exists between the file system and the kernel level services that it uses. This coupling reduces the ability of the file system developer to port the file system to another operating system.

Developing the file system from scratch inside the kernel allows the file system developer greater freedom in the implementation process. Gaining experience with the internals of the kernel requires time and considerable knowledge of the underlying operating system structure. Developing a file system from first principles does not utilise any of the development previously done. Redeveloping all of the file system functionality in this way does not make any sense.

Examples of file systems developed in the kernel include the Echo Distributed File System [6] and NFS. Figure 2 illustrates how a typical user level process utilises the file system operations in the kernel. It requires two calls, (1) for the request and (2) for the response.

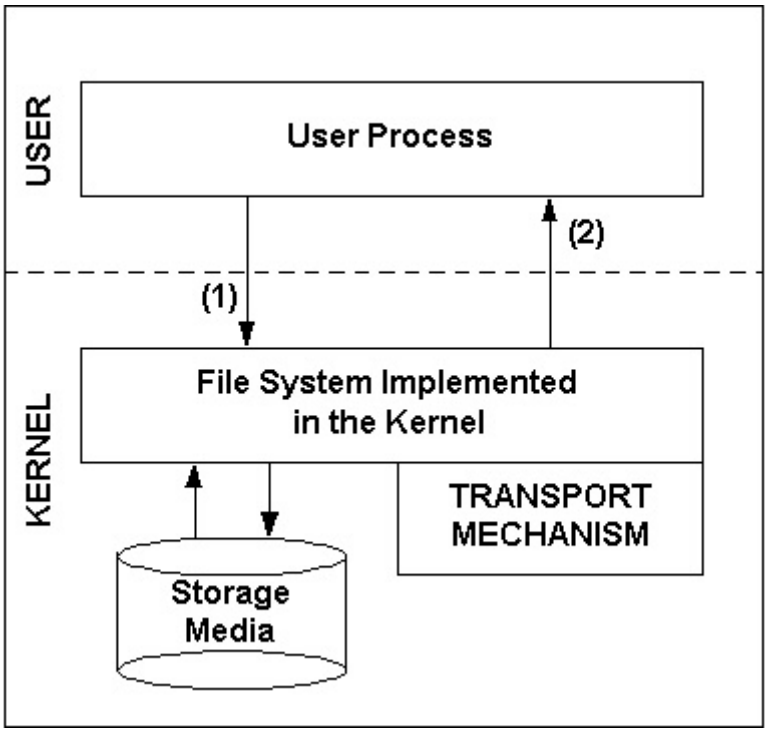

Figure 2: File System implemented in the kernel.

For the majority of file systems developed in the kernel (in UNIX based operating systems) the system calls that are made to the kernel are routed through the Virtual File System layer (VFS). The VFS layer allows the kernel to provide access to different file systems through a common interface. The kernel provides a data structure called a vnode to represent an open file or socket without revealing the underlying file system implementation. All operations performed on vnodes are the same regardless of the underlying file system implementation. Figure 3 shows how file system operations are routed through the VFS vnode to the underlying file system ext2 [7]. The NFS file system is also present to illustrate how a user could access

remote files. This serves to illustrate how multiple file systems are catered for in the same kernel.

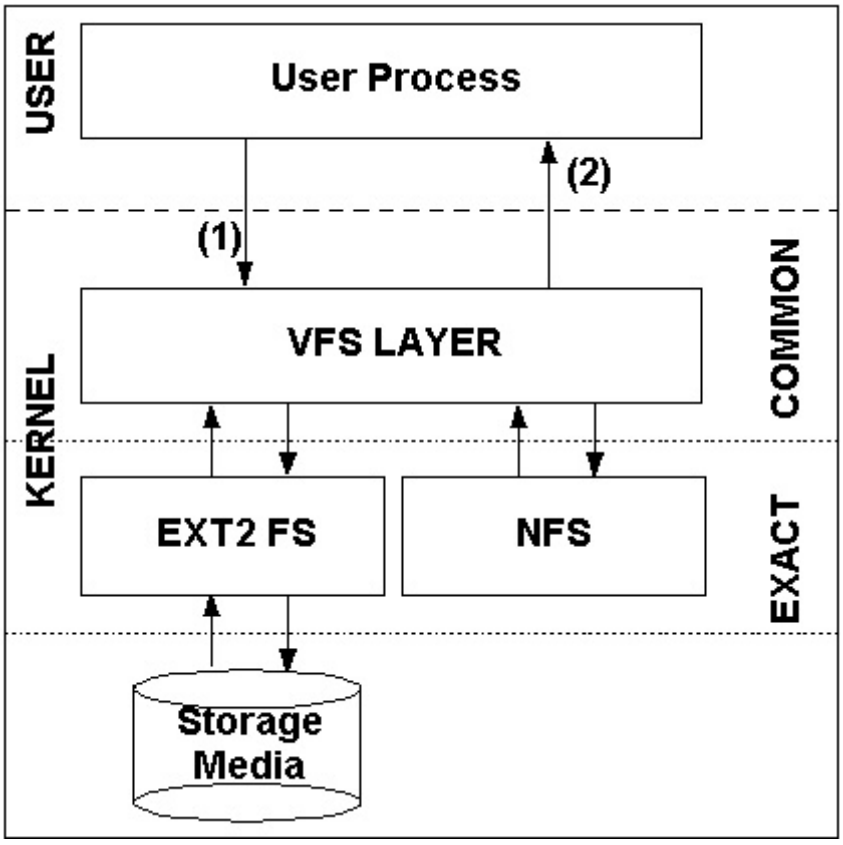

Figure 3: VFS Supports Multiple File Systems in the kernel[1].

1.2.3 Stackable Layer Kernel Level Process File System

Significant work has been done using stackable layers to leverage existing functionality provided by file systems implemented in the kernel. The extensible file systems in Spring [9], Lofs, Rot13fs and Usenetfs are examples of file systems that use stackable layers and are discussed in "A Stackable File System Interface For Linux" [8]. The Fiscus Replicated File System [10] describes how stackable layers provide replication of files. An implementation of a cryptographic file system in Linux, Cryptfs [11] demonstrates how stackable layers can be used to create a useful file system by leveraging the existing file system functionality. Stackable layers use

[1] Adapted from A Stackable File System Interface for Linux [8] .

the VFS interface and vnodes to layer functional operations one on top of the other. Stackable layers are described in more detail in "Vnodes: An architecture for Multiple File System Types in Sun UNIX" [12].

The process of developing file system functionality in the kernel is difficult due to the constraints that the kernel imposes. It is preferable to reuse existing code whenever possible as it has usually been thoroughly tested and is generally stable. The main idea behind stackable layers is to reuse existing functionality by layering new functionality on top of it. Developers can reap the benefits of previous work and concentrate on the problems associated with their required functionality.

The following figure shows how a stackable layer is used inside the kernel to utilise existing functionality.

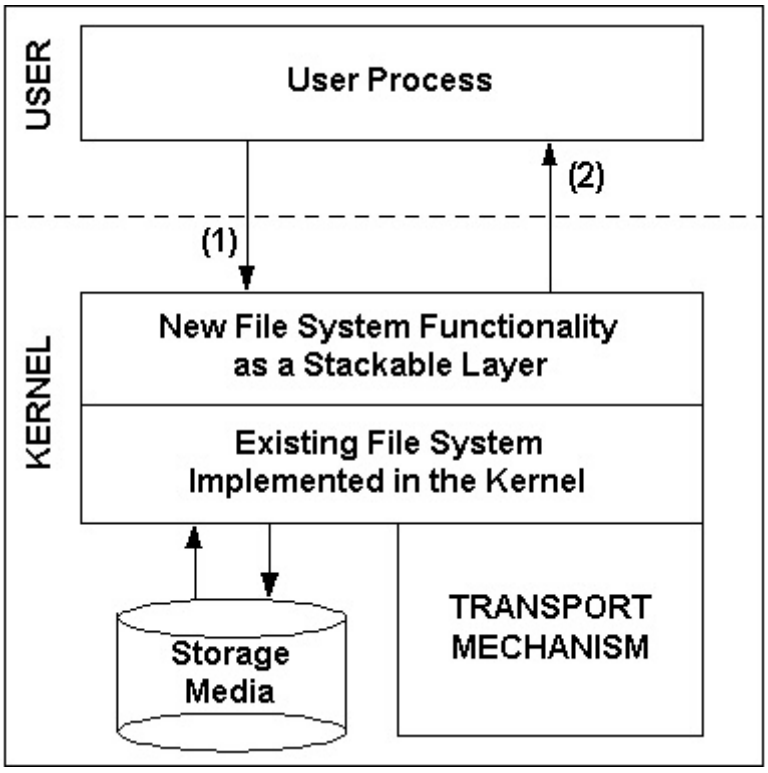

Figure 4: New File System functionality added to the Kernel as a Stackable Layer.

1.2.4 Stackable layers for the CryptosFS Architecture

Having reviewed the different development techniques available, stackable layers was selected to develop the prototype of CryptosFS. The reasons for choosing stackable layers are as follows.

1. A significant amount of work has been done on developing file systems such as ext2 or NFS. Stackable layers allow this existing functionality to be reused. This frees the developer to concentrate on the specific implementation problems.
2. By placing the prototype of CryptosFS as a stackable layer within the kernel, it is possible to avoid the performance impact inherent in a user space implementation.
3. There was a limited amount of time available for the development of the CryptosFS prototype. By reusing existing functionality in Cryptfs and NFS, this allowed us to concentrate on implementing the specific functionality of CryptosFS.

As the CryptosFS file system is supposed to be fast it is more appropriate to place the file system in the kernel. Developing the file system, as a user process can't provide the fast performance that is required. This is because of the additional context switches that a user process requires. Placing the file system functionality in the kernel complicates the development process because it requires specialist knowledge of the kernel. Creating the file system from scratch is not a realistic option as sufficient time is not available to gain the necessary experience.

The VFS layer of the operating system allows multiple file systems to be supported by the kernel. Stackable layers use VFS layer vnodes to enable a function of one file system to use the functionality provided by another file system. Our cryptographic file system layer (CryptosFS) added to the kernel provides an exact encryption/decryption

service to the user. CryptosFS uses the common services provided by ext2 and NFS, the underlying file systems as shown in Figure 5.

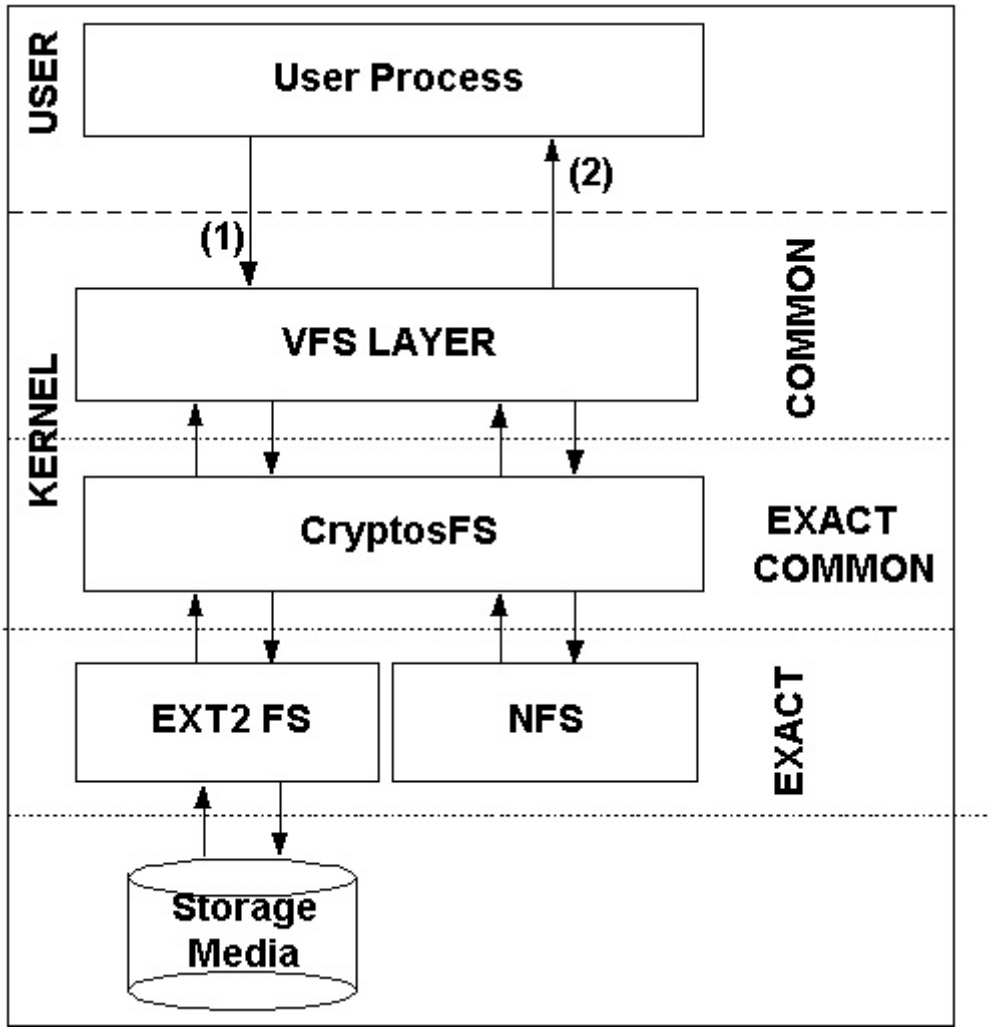

Figure 5: CryptosFS uses ext2 and offers Services to the VFS Layer[2].

[2] Adapted from A Stackable File System Interface for Linux [8]

2 State of the Art in Cryptographic File Systems

The idea of applying encryption to data stored in the file system is not a new one. There are many examples in the literature and the commercial world of file systems that utilise encryption. These include:

1. CFS uses DES [13] cryptography to provide confidentiality of file data and file meta-data.
2. Truffles [14] uses DES to encrypt file data and asymmetric-key cryptography to exchange the DES keys.
3. TCFS uses DES encryption to provide confidentiality and authentication.
4. Cryptfs uses Blowfish [15] encryption and stackable layers technology to encrypt the file data and file meta-data.

Reviewing the literature for the reference file systems helped to identify the Cryptfs implementation. Cryptfs uses stackable layers to allow the fast implementation of file system functionality. Stackable layers exploit the ability to use vnodes to allow different file systems to use each other's functionality. By selecting the Cryptfs implementation as a starting point to use for development, it provides a large amount of stable kernel code. Using the Cryptfs file system allows the development effort to be concentrated on the required cryptographic functionality without having to worry about the low-level details of the file system.

2.1 Cryptographic File System – (CFS)

CFS is a portable user-level cryptographic file system that is based on NFS. It uses the NFS loop back device to intercept system calls and redirect them to the kernel. CFS applies encryption at the granularity of the directory both on the local and remote file

system. Files stored in the directory are stored under a different mount point and a user-attached directory.

Files in CFS are encrypted using a series of user level programs. A specialised form of the “mkdir” command “cmkdir” is used to create encrypted directories. During the creation of a directory, the user is prompted to enter a key. To use an encrypted directory the user must attach the encrypted directory to a normal directory. This requires the key, the name of the encrypted directory and a directory name that is used as a mount point to access the encrypted directory during the attach process.

For example a user creates an encrypted directory “/home/oshanahd/myeyesonly” with a key “declan”. To access the encrypted directory the user enters the key “declan”, the directory name “/home/oshanahd/myeyesonly” and the mount directory /mnt/oshanahd that is used to attach the encrypted directory. CFS determines whether a user has the right to access the attached directory based on the user id of the process trying to access the directory.

Different cryptographic algorithms can be used to encrypt the data in CFS, including DES as discussed in the Data Encryption Standard. The implementation discussed in the literature describes how DES with a 56-bit key is used in different modes to provide security. 56-bit DES no longer provides adequate security because its 56-bit key-size is vulnerable to brute-force attack as explained in “Efficient DES Key Search” [16]. Advances in differential cryptanalysis as discussed in “Differential Cryptanalysis of the Data Encryption Standard” [17] and linear cryptanalysis in “Linear Cryptanalysis Method for DES Cipher” [18] indicate that DES is vulnerable to other attacks also.

CFS encrypts file data and the meta-data. Encryption of the file meta-data results in path names and file names that are on average fifty percent bigger than the normal unencrypted equivalent. This reduces the size of valid file names that the user can use. CFS supports multiple directories allowing different keys and different cryptographic algorithms per directories. The additional context switches that CFS performs to service file requests limit the performance of the file system.

2.2 Truffles

The Truffles file system is a distributed file system that uses the Fiscus replicated file system and TIS/PEM [19]. The Truffles file system provides replication and sharing of file data. Privacy enhanced mail (PEM) provides security in the form of authentication and encryption. Email is used to exchange the information required to share a file volume replica. Information in the email is encrypted using DES. To allow different users share information securely requires the exchange of the DES key. The key exchange uses the public-key contained in an X.509 digital certificate. Using DES to encrypt the information in email leaves the data open to brute force attack as discussed in “Efficient DES Key Search”.

Each Truffles volume has a different DES key so that access to the each volume is restricted to only those users who possess a copy of the relevant DES key. The use of email to exchange DES keys allows users to share access to file volumes without the assistance of system administrators. System administrators can use policy to limit the abilities of certain users to share file volumes. This provides additional flexibility and security.

Fiscus is a stackable file system that resides in the kernel and utilises the underlying file system that the operating system provides such as the UNIX File System (UFS) or

NFS. UFS provides local file system storage while NFS provides remote file system storage. As Truffles is built on top of Fiscus it is as readily available as NFS. Despite the fact that the file system capability of Truffles is in the kernel, the performance of Truffles is limited by the use of email to exchange information. Email can be lost in the Internet or can be delayed due to congestion, this seriously degrades the performance of the file system.

2.3 Transparent Cryptographic File System - (TCFS)

TCFS is a modified client side NFS server that communicates with a remote NFS server and with a specialised RPC based attribute server. TCFS is only available for Linux and requires both its client and server run the Linux operating system. TCFS can use several different block ciphers including DES and IDEA [20].

TCFS provides transparent management of user keys. Instead of the user being prompted to enter a key by a specialised program, random keys are generated by the file-system. The random keys are encrypted with the password of the user and stored in a file called /etc/tcfspasswd. The user id is used as an index in to the key file to retrieve the key for the user process trying to access the file. This has the advantage of transparently handing the encryption of the file data. At the same time, it reduces the security provided by the system to the difficulty of decrypting the user password to retrieve the keys. Dictionary password programs can be used crack a user password with little difficulty, these programs are freely available on the Internet.

TCFS provides a finer granularity of encryption than CFS. An extended file attribute called “secure” is tested upon the creation of the file. If the “secure” attribute is set, subsequent read and write operations are directed through the cryptography layer. If the “secure” attribute is not set, the read and write operations are treated as normal.

2.4 Cryptfs

Cryptfs is implemented as a kernel resident file system. Cryptfs can be mounted on any local or remote directory and can utilise any underlying file system such as UFS, ext2 or NFS. Cryptfs does not require any specialised daemon program to run as it layers itself on top of the existing underlying file systems. Cryptfs is implemented as a stackable code interface. Similar to CFS, users of Cryptfs are prompted to enter a pass phrase to generate a key for authentication. A message digest of the pass phrase is generated using MD5 and is stored in the memory used by Cryptfs. The keys to encrypt the file data are not stored in a file, this makes it more secure than TCFS. As the user must enter the pass phrase at the start of each new session that is started, this results in reduced flexibility but provides greater security.

Cryptfs uses Blowfish symmetric-key cryptography for encryption of file data and meta-data. A 128-bit key provides a balance between performance and encryption strength. Keys can be used in two different ways. The user id can be used to identify the key to use. Alternatively, a combination of the user id and the session id of the accessing process can be used to identify the key. The second method provides additional security, as a malicious user who can use the root user capability to forge the user id of the user can't attain the same session id as the user.

Blowfish has the property of preserving the size of the encrypted data. This is desirable as many programs access files using an offset. If the encrypted data is a different size to the unencrypted data, it renders the offset used by programs invalid. This requires a conversion scheme to translate the offset values of the unencrypted data to the offset of the encrypted data. This in turn has implications for performance.

Encryption of the file name and path name attributes adds additional complexity. UNIX file names and path names have certain restrictions regarding valid names. Using certain characters such as "/" or null produces invalid file names. This is a problem in Cryptfs as encryption of filenames produces invalid filenames. Cryptfs solves this problem by uu-encoding the filename to convert them to valid values. It does not encrypt the "." and ".." directories because it could provide malicious users with examples of encrypted strings to try a known plain text attack. Uu-encoding filenames results in the encrypted filenames being 25% larger on average than the corresponding unencrypted names.

The performance of Cryptfs is improved by the location of the encryption/decryption layer in the kernel. This results in the same number of context switches as file access in a regular file system.

2.5 Summary of properties of Cryptographic File Systems

The table below summarises the properties of the different cryptographic file systems researched. The CryptosFS file system is supposed to be fast and secure and the size of the encrypted file data is required to be the same as the clear text. From the review of the available file systems, Cryptfs was selected for detailed analysis. Cryptfs is a kernel resident file system. This means that the performance of the file system is better than user level file systems. As Cryptfs uses stacking vnodes, the implementation is less complex than the complete implementation of a file system such as ext2. Even though it requires system administrator intervention to install, it is not considered a major problem.

Properties	CFS	TCFS	Cryptfs	Truffles
Authentication using public keys.	No	No	No	Yes
Remote file access capability.	Yes	Yes	Yes	Yes
Cipher text preserves data size of clear text.	No	No	Yes	No
Files encrypted with individual keys	No	Yes	No	No
X.509 Certificates used.	No	No	No	Yes
Kernel resident file system.	No	No	Yes	Yes
Requires System Administrator Intervention.	Yes	Yes	Yes	No

Figure 6: Summary of the properties of different cryptographic file systems.

3 Design of CryptosFS

3.1 Trusted Computer Base and CryptosFS

Cryptography can be used in a distributed file system to provide confidentiality, integrity and authentication. A distributed system is by its very nature made up of many different components. Each of the components in a distributed system potentially provides an avenue for an attacker to exploit, so as to gain unauthorised access to data and information. Depending on the application domain, the requirements for information security are very different. The military typically places the highest priority on non-disclosure of information. The banking industry is more concerned with integrity of information and preventing attackers from modifying data. Utilities and service providers place a high priority on ensuring availability. An attacker who can successfully interfere with the provision of a service can be extremely damaging to a service provider.

The use of components to produce a distributed system requires that the components of the system have to be evaluated with regard to security. The combination of components to produce a distributed system forms what is called the trusted computing base (TCB). The trusted computing base is defined by Butler Lampson in "Requirements and Technology for Computer Security" [21] as

> "The set of trusted hardware and software components is called the trusted computing base or TCB. If a component is in the TCB, so is every component that it depends on, because if they don't work, it's not guaranteed to work either."

As part of the design process for CryptosFS it is important to evaluate how exactly CryptosFS fits in to the TCB. The different parts of the file system were evaluated to determine how they could be secured to provide the necessary level of protection.

3.1.1 Lack of Trust in the components of CryptosFS

CryptosFS is a fast secure version of NFS. Like other NFS implementations, it has the following components.

- Client
- Server
- Communication mechanism

The design of CryptosFS requires determining the level of trust required among the different components. The server should distrust the client and require that all of the operations it performs be validated. Validation of operations requests from a client requires the server to verify that the client has the authority to perform the requested operation. CryptosFS does not authenticate the clients identity but it does ensure that a valid client is performing the operation.

In CryptosFS a new access control mechanism is used to control access to file stored on the server. The new access control mechanism uses RSA asymmetric-key encryption [22] to create digital signatures. This allows the server to verify that only valid clients are allowed to perform operations. It also allows the client to give other users the capability to perform operations. Trust is established between clients and servers by the ability of the server to validate the operations that the client requests.

Just as the server does not trust the client without verifying that it has the correct authority to perform an operation, the client does not trust the server. The client does

not trust the server because someone unknown to the client administers the server. The server has the ability to disclose information without the authorisation of the client.

The client can prevent unauthorised access to data by using encryption. Even if the server discloses the data belonging to the client, the data is in encrypted format and so is useless without the corresponding key to decrypt it. The client encrypts all of the data with symmetric-key encryption before it sends it to the server. The data is never decrypted by the server. The server does not possess the key to perform the decryption of the data. This enforces trust between the client and the server even though there is no explicit trust.

The communication mechanism allows the client and server to pass requests and responses to each. The client and the server do not trust the communication mechanism, as it is possible for an attacker to compromise it. CryptosFS is designed to try to overcome the problem of not trusting the communication mechanism. The problem with the communication mechanism is that an attacker can read and change the data being sent over it. The ability to change the data sent to a server negates the ability of a server to validate a client request for an operation. CryptosFS encrypts the data before it is sent over the network, this ensures confidentiality of the data. The server uses the digital signature created by the client to validate the client requests. This ensures the integrity of the data.

3.2 Access Control Mechanisms

Providing secure access to objects stored in a distributed environment is difficult. As the number of objects and users trying to access those objects increases, traditional access control mechanisms become a bottleneck. Centralised management of access

control works on the LAN of an organisation but it is impractical for sharing objects in the global Internet. Centralised management of access control complicates the creation of new users, it prevents arbitrary once off access to files from non-trusted users and is a single point of failure.

3.2.1 Access Control Lists

Access Control Lists (ACL) as discussed in Pasc P1003.1e [23] provide a mechanism for secure access to objects in a distributed system. ACL's are a form of discretionary access control as the end user determines who can access the object. Access control is defined in terms of objects and subjects. Objects are defined as what is controlled by the access control; a subject is defined as the entity that commences the access to the object.

Each subject is described by a collection of attributes. The attributes are used to identify the subject to the object and to control the access that the subject has to the object. Validating the subject involves authentication and authorisation. Authentication involves acquiring and verifying the attributes of the subject. Authorisation validates the permissions of the subject by the object. Objects are governed by a set of rules called the access control entity (ACE). There are two types of ACE: authentication ACE and authorisation ACE.

The authentication ACE provides a way to identify a subject from its attributes i.e. machine address, username or the location of the authentication information. The authorisation ACE defines the permissions that a particular subject has. Subjects can be organised into groups so that authentication and authorisation are based on the membership of the group.

ACE's are used to form ACL's that can be stored in a centralised structure such as a file or a database. The ACL mechanism obtains the subjects ACE from the ACL to authenticate and authorise the subject's access to the required object. As the ACL is stored on a single host, it is a potential single point of failure. Management and administration of ACL's is problematic when there are large numbers of changes to be made to the subject's attributes.

3.2.2 Role Based Access Control

Role Based Access Control (RBAC) is an example of mandatory access control. The subject who has access to an object does not have the automatic right to distribute access to the object. RBAC seeks to differentiate between the subject and the task to be performed. A subject can perform a task only if it has the correct permissions to access the objects required to complete the task. By placing the permissions to access an object under a specific role, any subject can be assigned that role.

RBAC differs from ACL's because a subject does not possess the ability to pass permissions on to another user. RBAC is pessimistic due to its restrictive nature. It assumes that subjects will attempt to pass on permissions and so by mechanism prevents it from happening. RBAC is group oriented because it relies on the assignment of subjects to roles. Each role has the ability to perform a certain subset of transactions; RBAC mechanisms adhere to the Principle of Least Privilege as discussed in "Integrity in Automated Information Systems" [24]. A role that a subject has is only granted the minimum privilege required to perform a task. RBAC is centralised in nature as it requires a system administrator to manage the assignment and revocation of roles to and from subjects.

3.2.3 Capability Access Control

Capabilities provide access control to an object in a system by restricting access to the object to only those users who possess a token. The token provides the token holder with permission to perform operations on the object. Capabilities provide a single mechanism for accessing, naming and securing all objects within a system. Dennis and Van Horn discuss capabilities in “Programming Semantics for Multiprogrammed Computations” [25]. Capabilities are used in the Amoeba operating system [26], in SFS [27] and in CapaFS [28].

Using capabilities provides improved flexibility as they can be shared out among subjects. Sharing the capability to read data from an object is possible by giving another subject a copy of the read capability. As copying capabilities provides a mechanism for sharing access to object, it is important that the capabilities are difficult to forge. If it is not difficult to forge capabilities, malicious subjects can gain access to objects. Strong encryption provides protection against the arbitrary forgery of capabilities.

Capabilities come in several different forms, these include tagged, partitioned and sparse.

- Tagging is a computer-architecture-oriented technique; a number of bits are added to each memory area, so a distinction can be made between the data and capabilities functionality.
- Partitioned capabilities are stored in a special area separately from any data and can only be accessed by the system. This separation is used to preserve the integrity of the capabilities. Partitioned capabilities are also computer-architecture-oriented.

- Sparse capabilities do not need to be distinguished from data by tagging or partitioning; a simple bit string is used to represent them.

3.2.4 Lack of Formal Access Control in CryptosFS

CryptosFS is designed as a distributed cryptographic file system that does not require the server to authenticate the identity of the user. CryptosFS does not use traditional access control for validating users. CryptosFS relies on creating digital signatures and using the digital signatures to verify requests for operations from clients. The server does not attempt to verify the identity of the user. Validations of read and write requests are performed by decrypting a digital signature received from the client making the request. To successfully read the data in the server the public-key for the file is required, conversely the private-key is required to perform a write operation.

3.3 CryptosFS Security Model and the use of Cryptography

The use of cryptography reflects the lack of trust between the components of CryptosFS. The security model used for the CryptosFS prototype uses three different types of cryptography. These include Blowfish symmetric-key cryptography, RSA asymmetric-key cryptography and MD5 message digests. This combination of cryptographic algorithms is used to ensure confidentiality, integrity and authentication of file data and file meta-data.

3.3.1 Confidentiality in CryptosFS

Confidentiality ensures that only the valid users of data can perform the operation to read the file data. CryptosFS ensures confidentiality of file data and file meta-data by encrypting it with symmetric-key (Blowfish) encryption. Only those users who possess the correct symmetric-key key can decrypt the encrypted file data and meta-data. The operation to encrypt the file data and file meta-data is only performed on the client of CryptosFS and the symmetric-keys are never transmitted across the network.

The data that is transmitted across the network is always in encrypted format, this guarantees confidentiality of the data even if it is successfully retrieved from the network. Data stored on the server is stored in encrypted format. If an attacker can compromise the server, the confidentiality of the file data is maintained, as the data is stored in encrypted format.

3.3.2 Authentication in CryptosFS

Authentication in CryptosFS is achieved by the use of digital signatures. A client that tries to perform a remote read operation must prove to the server that it has the authority to perform the operation. When a file is created on the server the public-key for the file is stored. The authentication process for a read operation involves the creation of a digital signature by the client. The client creates a digital signature by first producing a message digest of the encrypted data. The message digest is then encrypted with the public-key of the file. The server possesses the same public-key as the client and creates its own digital signature. The server authenticates the client's read request by comparing the digital signature received from the client against the digital signature that it creates. If the digital signatures match then the client has successfully authenticated itself to the server for a read operation.

To authenticate itself to the server for a remote write operation the client creates a digital signature by encrypting a message digest with the private-key of the file. The server decrypts the digital signature received from the client and retrieves the message digest contained in it. The server creates its own message digest and compares it against the message digest recovered from the digital signature of the client. If the message digests match then the client has successfully authenticated itself to the

server for a write operation. Only the client who possesses the private key can create the correct digital signature.

3.3.3 Integrity in CryptosFS

The integrity of files created on the server is maintained by the use of digital signatures. Write operations to modify the data stored on the server require the private-key. The integrity of the file data is guaranteed because even if the public key is changed on the server, the comparison of the message digests stored in the digital signatures will fail.

3.4 Location of Cryptography in the System

The choice of the cryptographic system is important but it is also critical to implement the cryptography in the correct place in the system. Locating the cryptography in an inappropriate location in the system can negate the security offered by it. Refer to “A Cryptographic File System for Unix” by Matt Blaze for a thorough description of the options available for encrypting data.

3.4.1 Manual Encryption by the User

The user can manually encrypt the data using an encryption tool such as PGP [29] or the UNIX crypt program. If encryption is required on a small scale by a single user then manual encryption is practical. Manual encryption requires the user to keep a copy of the key used for encryption. This leaves the data open to compromise from the key being stolen. Apart from the security risks of manual encryption, it is inflexible for large numbers of files. Manual encryption of files is not scaleable, because as the number of files increases the number of keys that the user has to manage increases.

If a user encrypts the data manually, it is not possible to share the data with another user unless they possess a copy of the cryptographic key and the cryptography program used to encrypt/decrypt the data. Users are human and can make mistakes such as entering the wrong encryption key or worse losing the key. Many programs that are used in the UNIX environment create copies of data that they work with. An encryption program may make a temporary copy of the clear text being encrypted and may not delete it immediately once the encryption is finished.

3.4.2 Encryption at the Application Level

An alternative to applying encryption manually by the user is to embed the encryption in to the application that creates the data. An example of this is an editor that encrypts the data in the files that it creates. The user is prompted by the application program to enter an encryption key when the file is written to or to enter a decryption key when the file is read. This is a little more flexible than the manual approach because the encryption process is seamlessly integrated into the application.

Sharing of data between different applications requires that each application which accesses the data, must implement the same cryptographic algorithm. Multiple implementations of a cryptographic algorithm can introduce problems due to an error in an implementation. Bruce Schneier in “Data Guardians” [30] describes some of the problems encountered with different implementations of the DES algorithms. Failure on the part of the user to encrypt the file or to delete the clear text version of the file data can render useless the security provided by application encryption useless.

3.4.3 Encryption at the File System Level

Applying encryption to the data at the user or application level is not practical because it requires the active participation of the user. Users are human and are prone to error. It is better to take the responsibility of encrypting the data away from the user. If users

do not have to worry about keys then the potential for problems from incorrect keys being used or the keys being mislaid is eliminated.

Entrusting the file system to perform encryption reduces the risks of security breaches due to multiple implementations of cryptographic algorithms. Cryptographic algorithms are complex and mistakes in implementation are possible. Experiences with the DES algorithm provide evidence of problems encountered due to the complexity of coding cryptographic algorithms.

The end-to-end argument as specified by Saltzer et al in “End-to-end arguments in system design” [31] discusses the use of encryption to provide for secure transmission of file data. Saltzer et al argue that it is better to let the end application apply encryption rather than let it be performed by the communication subsystem. It is not appropriate to trust the communication subsystem to manage the keys required for encryption. A further complication of applying encryption to the communication subsystem is that once data has cleared the communication subsystem on the target machine it will be in unencrypted format.

3.4.4 Encryption at the System Level

The data can be encrypted at the system level using a hardware device. The hardware device can encrypt the data before sending it over the communication link. The clipper chip [32] is an electronic device that can be embedded in to network cards. This provides a mechanism for encrypting all communications between different principles. It never established itself because the US government possessed a backdoor that allowed it to override the encryption mechanism.

When file data is stored on a remote file system, the security of the communication link is important. Malicious users who can gain access to the link can read data sent in the clear to a remote site. Programs such as packet filters make this easy to do. A hardware device can be used to encrypt and decrypt data before it is stored on the physical media. Using hardware in this manner complicates back up and retrieval of information stored on the media. Adding an additional component to the system further increases the risk of failure of a component. Failure of the hardware can render the data inaccessible. The need to store keys at some location in the network whether at the local or remote site complicates matters and increases the risk of security breaches.

3.4.5 CryptosFS & Encryption at the file system level

The location in the operating system where cryptography is applied is important. Security of the generation, management and storage of keys can impact the overall effectiveness of encryption when it is applied to an operating system. During the design of CryptosFS the use of cryptography at the user and application level were rejected because of the problem with security of keys and scaling problems. It is preferable to let the file system take responsibility for the application of encryption. CryptosFS is designed to encrypt file data at the file system level. The keys for encryption are generated and stored in the file system. Key management is completely transparent to the user. By applying the encryption in the file system, the file data is encrypted and decrypted at the client only. This ensures confidentiality of the file data from the client end to the server end and back. Blowfish symmetric-key encryption is used in CryptosFS to encrypt the file data and meta-data. This provides complete security to the data stored on the server.

The CryptosFS file system is responsible for managing the encryption keys. No other part of the system is required to manage keys. The file data is encrypted and decrypted only by the CryptosFS file system application on the client. As no other part of the system needs to decrypt the file data, the need for complicated key-management is removed. This simplifies the key-management model and improves the security of the file system.

Efficient distributed system design dictates that as far as possible processing is transferred from the highly burdened server to the lightly burdened client. By passing the responsibility for encryption of file data to the client, a computationally expensive operation is removed from the server. This allows the server to concentrate on performing other work and allows it to scale more effectively to handle additional client requests.

3.5 CryptosFS - Design Goals

The design of CryptosFS reflects a desire to add stronger security to NFS through the application of different cryptographic techniques. The use of RSA asymmetric-key encryption was considered to encrypt the file data. After consideration, this was rejected because the performance of file operations would have been very poor.

Blowfish symmetric key encryption is used in CryptosFS to perform the file data encryption. There are no references in the literature to show that the Blowfish algorithm has been successfully cryptanalysed. Blowfish offers a balance of strong encryption and high performance; this makes it ideal for encrypting data. The choice of encryption selected influenced the design of CryptosFS.

The design goals for CryptosFS include:

- Keys produced in the file system are stored in the file system. User assistance is not required to produce the keys. Keys are generated from random information from within the operating system.

- When a file is created, the file system automatically produces the required symmetric and asymmetric-keys. The keys generated by CryptosFS are saved somewhere in the file system.

- Encryption of file data and file meta-data is performed by the client. The encrypted file data and meta-data are stored on the server in encrypted format. No decryption of file data is performed by the server. The server only uses encryption to authenticate operations by manipulating the digital signatures produced by the client.

- The design of CryptosFS follows efficient distributed system design principles as far as possible. By passing the encryption and decryption of data to the client, the computationally expensive work of encryption and decryption is transferred from busy server to the less lightly loaded client. This allows the server to scale more efficiently and handle an increase in the number of file operations.

- CryptosFS must modify the kernel RPC mechanism to allow the transfer of digital signatures from the client to the server and from the server to the client.

4 Implementation

4.1 CryptosFS - Implementation goals

The following are the implementation goals were set for the implementation of CryptosFS.

- A combination of Blowfish symmetric-key encryption, RSA asymmetric-key Encryption and MD5 message digests algorithms are to be implemented. The different cryptographic algorithms are used to provide a mechanism for validating operation requests from clients on the server, as the server does not trust the client.

- Cryptography is expensive in terms of the additional time it adds to a file operation. To offset the cost of the encryption and decryption the implementation of these three algorithms have to be implemented in the kernel to ensure that they are as fast as possible.

- The following bit sizes are used for the three different cryptographic algorithms.
  - Blowfish key size of 448-bits. A key size of 448-bits provides a good balance of strong encryption and high performance.
  - RSA key size of 1024-bits. A key size of 1024-bits provides strong encryption to protect the digital signatures against brute force attack.
  - MD5 message digests of 128-bits. The 128-bit message digest size is a standard size used for the creation of digital signatures.

- The Blowfish symmetric-key encryption algorithm is used to encrypt and decrypt the file data and the file meta-data. All file data and file meta-data is stored in

encrypted format on the server. No symmetric-key encryption is performed by the server to improve security.

- The MD5 algorithm is used to generate the message digests. The message digests are used to produce a digital signature by signing them with the RSA public or private-key. The RSA public-key is used by the NFS server to validate the digital signatures and to authenticate the results returned to the NFS client.

- To implement the RSA asymmetric-key encryption in the kernel, a method for producing large integers in the kernel had to be found.

- The kernel NFS client and server implementation in Linux use kernel RPC as its transport mechanism. This RPC is hand coded, as there is no rpcgen program for the Linux kernel. The RPC code had to be analysed to determine how best to modify it. The modifications allow it to transport the digital signatures and associated asymmetric-keys from the client to the server and vice versa.

- CryptosFS is to be developed in Linux and has to use Linux kernel modules to allow the dynamic loading of the file system into the running kernel.

4.2 Architecture of CryptosFS

The general architecture of the CryptosFS file system is illustrated in the following diagram.

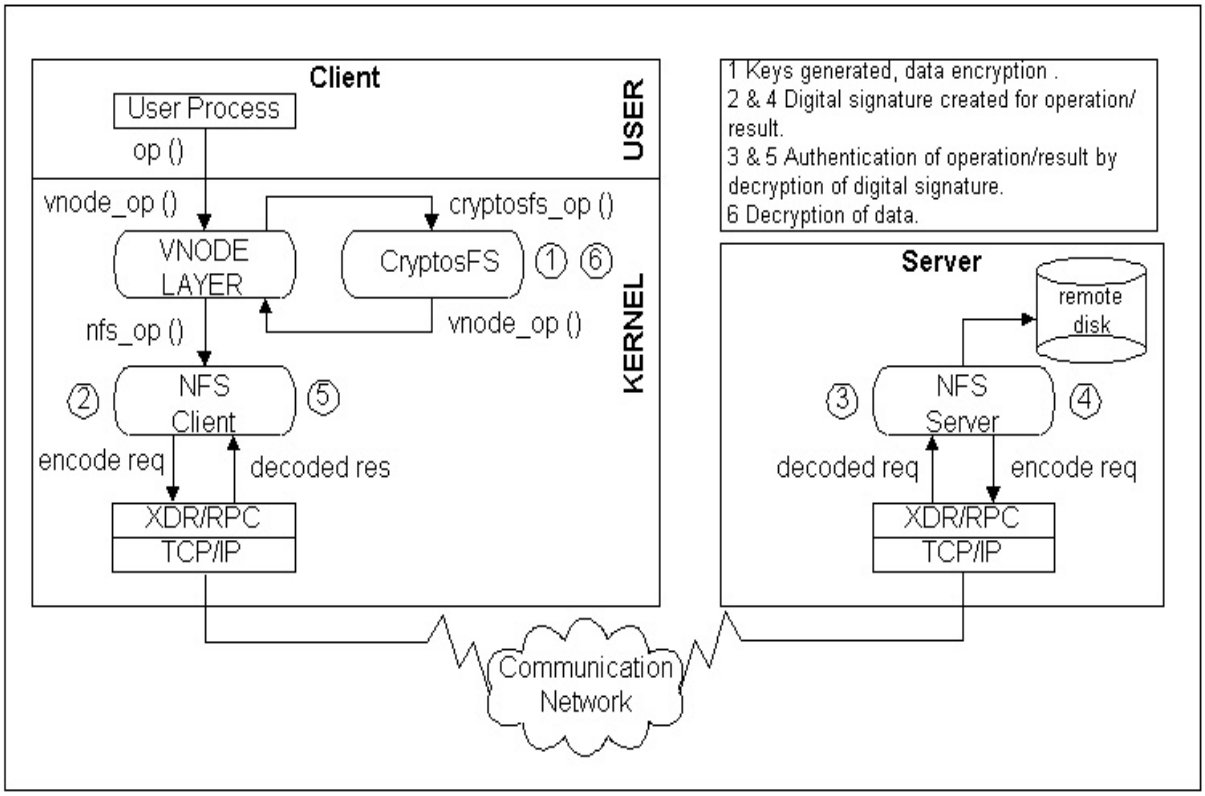

Figure 7: General Architecture of the CryptosFS implementation.

The steps numbered one to six describe how and where the different cryptographic algorithms in CryptosFS are implemented.

1 The symmetric and asymmetric-keys for each file are created when the file is created. The keys are stored in the vnode for the file. Data received from the user is encrypted before it is passed down to the underlying NFS client. The encryption is performed using the symmetric-key stored in the file vnode.

2 The NFS client receives the encrypted data and the request for an operation from the CryptosFS layer. The client creates a digital signature for the operation it is requesting from the server. The client produces the digital signature by creating a message digest of the encrypted data received from the CryptosFS layer. The message digest is signed with the public or private-key of the file. The public-key is used for read operations and the private-key is used for write operations. The

digital signature, the public key and the encrypted data are encoded using XDR before they are sent to the server.

3 The NFS server decodes the request from the client. The decoded request contains the operation requested, the digital signature and the public-key for the file.

- If the client requests a write operation the server uses the public-key to decrypt the digital signature to retrieve the message digest created by the client. The server creates its message digest from the encrypted data and compares it against the message digest retrieved from the client's digital signature. If the message digest produced by the client matches that produced by the server, the server knows the client who created the digital signature is authorised to perform the write file operation.
- If the client requests a read operation, the server creates a message digest of the encrypted data. The server encrypts the message digest with the public-key to create a second digital signature. The server compares the two digital signatures and if they match the server knows that the client is authorised to perform the read operation.

4 The NFS server performs the operation if the request from the client is validated correctly. The results of the operation are used to create another digital signature by creating a message digest of the result data. The server uses the public-key of the file to sign the message digest. The result data and the digital signature are encoded using XDR and sent back to the client.

5 The NFS client decodes the results received from the server. The decoded results contain the results of the operation requested by the client and the digital signature created by the server. The client creates a message digest of the results

data and signs the message digest with the public-key of the file to create a digital signature. The client compares the digital signature it created against the one received from the server. If the digital signatures match, the client knows that the result data is correct and has not been modified as it passed over the network.

6 The NFS client passes the results data back up to the CryptosFS layer. The result data is decrypted with the symmetric-key from the vnode. The decrypted data is then passed to the user process.

4.2.1 Functionality in CryptosFS

Figure 8 illustrates how the different components of the CryptosFS implementation fit together. The key generation and storage is performed in the CryptosFS layer. Both the NFS client and server generate digital signatures for the validation of the different file operations.

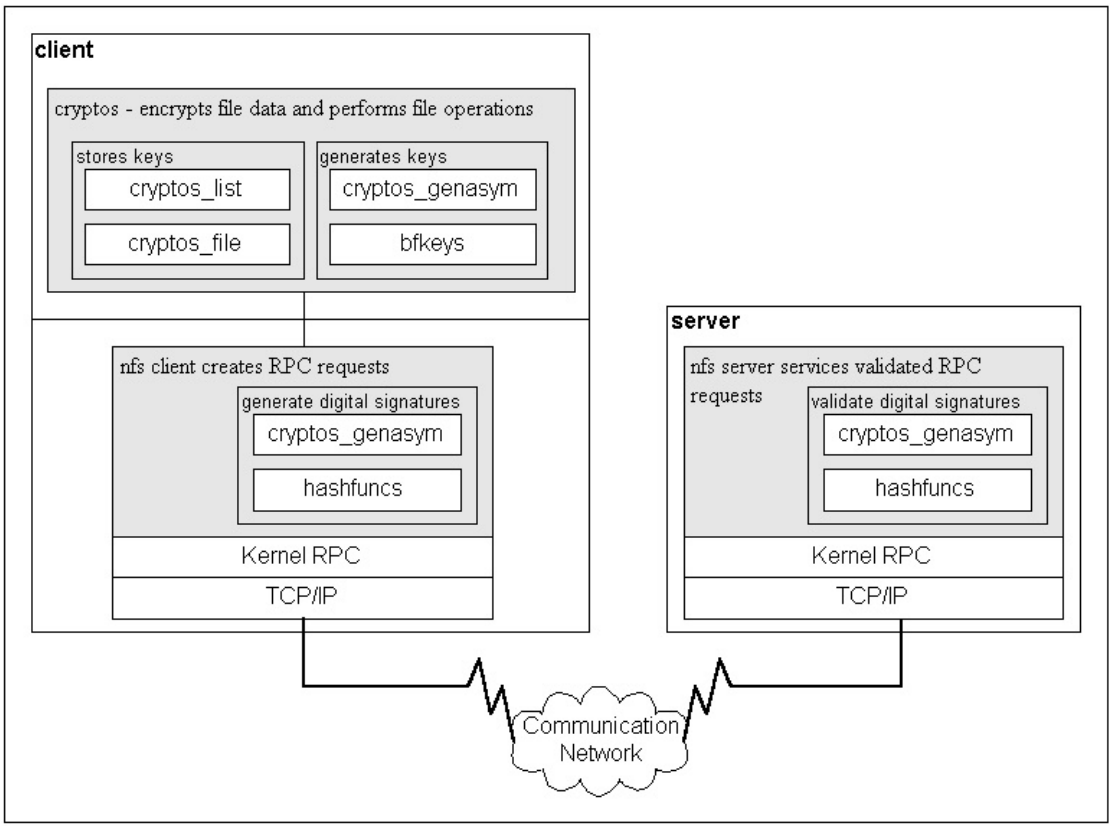

Figure 8: Structure of components in the CryptosFS File System.

There are four main components in CryptosFS that perform the key generation and management. The different components are discussed below.

1. cryptos_genasym generates the asymmetric-keys for each vnode; it also produces the digital signatures from the message digests on both the client and the server.
2. bfkeys generates the symmetric-key for each vnode.
3. cryptos_list and cryptos_file store the generated keys in main memory and in flat files respectively.
4. hashfuncs generates the message digest of the encrypted data. This message digest is used in the generation of the digital signatures.

Refer to Appendix A for a complete list of the software components of CryptosFS.

4.3 Key Structure for CryptosFS

CryptosFS uses symmetric and asymmetric-key encryption to provide confidentiality. To allow each file to possess its own keys, the vnode structure of the VFS layer is modified to store the keys to perform the encryption and decryption. The vnode is implemented in the Linux kernel as a C structure. The CryptosFS keys are defined as a layer of structures that are contained in the VFS vnode. The structure asym_keys_t contains the character representation of the public and private keys for the RSA asymmetric-key encryption. The symkey contains the character representation of the Blowfish key. By storing the keys in the vnode, the information accessible from each vnode can be individually encrypted and decrypted simply by retrieving the keys from the vnode structure.

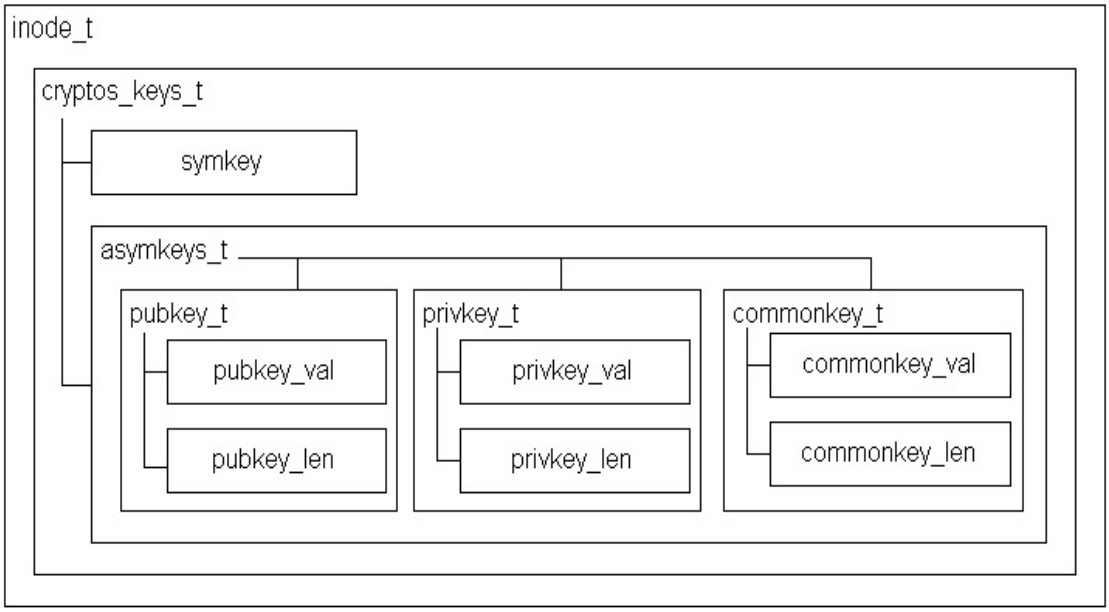

Figure 9: Structure for CryptosFS keys as stored in VFS vnode.

4.3.1 Vnode stacking and Encryption

The CryptosFS prototype encrypts and decrypts the file data and meta-data before passing it on to the underlying ext2 and NFS file systems. This is possible by using stackable layers and stacking vnodes. The vnodes of the VFS layer in the Linux operating system possess the capability to store arbitrary information. Stackable layers exploit this capability to store a pointer to another vnode (called a hidden vnode) in the higher-level vnode.

The encryption and decryption of data is performed on the highest-level vnode. The hidden vnode is used by the underlying file system to perform operations on the encrypted data. This allows the client to encrypt the data and pass it on to the ext2 or NFS file systems safe in the knowledge that the data will never be seen in clear text format. This mechanism allows the cryptography to be isolated into a separate layer; the file operations are performed by the underlying file system.

4.3.2 Blowfish symmetric-key Generation for CryptosFS

CryptosFS uses 448-bit Blowfish keys to encrypt file data and file meta-data. The Blowfish keys are created automatically when a file or directory is created. For each symmetric-key to provide sufficient entropy, it is produced using a seed value obtained from the system time in the Linux operating system. The system time in Linux is represented by jiffies. Linux Kernel Internals [33] describes jiffies as

> "Jiffies represents the system time since the system was started up in, they are measured in ticks. Each tick represents 10 milliseconds".

The seed to the Blowfish key generator is generated from the following formula.

- seed = (jiffies % HZ) * (1000000000/HZ)

jiffies = system time in milliseconds since the operating system was booted.

HZ = 100

The number of seconds from the time of day is retrieved from the operating system and is combined with the seed value to produce the Blowfish key according to the following formula. The key is 448-bits in length (56 bytes) and so is produced in a loop.

```
for (i=0; i < SYMKEY_SIZE; i++)
    symkey [i] = (unsigned char) ((seconds) * (seed * i)) % CHAR_RANGE);
```

The CHAR_RANGE is 255. Using the modulus function ensures that each of the characters produced is in the range 0 - 255.

4.3.3 Generating Large Integers in the Linux kernel

The implementation of CryptosFS requires a mechanism to generate large integers in the Linux kernel. The RSA asymmetric-key encryption relies on the difficult of factoring large numbers for security. This is discussed in detail in “A method for

obtaining Digital Signatures and Public-Key Cryptosystems". Without the ability to produce large numbers in the Linux kernel, it is not possible to implement the RSA asymmetric-key encryption algorithm in the kernel.

The kernel of the operating system is limited in the functionality that it provides. Once the kernel is loaded in to the memory of the machine it runs continuously without ever being swapped out to disk. This places a limit on the size of the kernel. The aim of the kernel is to provide low level services quickly and efficiently. Adding large integer capability to the kernel to generate large integers does seem to contravene the end-to-end argument. This is because the kernel does not have access to the functionality that a higher-level layer has. One of the implementation goals for CryptosFS is to use RSA asymmetric-key encryption in the kernel. This is not possible without a kernel implementation of a large integer package.

Two large integer packages were evaluated to determine whether it is possible to use them in the Linux kernel. These included the Large Integer Package (LIP) [34] and the GNU Multi-Precision Arithmetic Package (GNU MP) [35]. A considerable amount of effort was concentrated on trying to port the LIP to the kernel. Efforts to port the LIP to the kernel failed as it relied too heavily on the functionality of the glibc library.

The attempt to port the GNU MP package to the kernel was more successful. Porting the GNU MP package to the kernel required modification of its memory allocation routines. The GNU MP is a user space library that uses the glibc functions malloc, realloc and free for memory allocation and de-allocation. These functions are not available in the Linux kernel and so had to be replaced with kernel equivalents. The realloc function is the only function that caused significant problems as the malloc

and free functions mapped to the kernel functions kmalloc and kfree respectively. A function implemented using kmalloc and kfree allows the correct simulation of the operation of realloc.

4.3.4 RSA Asymmetric-key Encryption

The security provided by RSA asymmetric-key encryption is due to the difficulty of factoring large prime numbers. The algorithm for RSA asymmetric-key encryption is discussed in “A method for obtaining Digital Signatures and Public-Key Cryptosystems”. The details of the RSA algorithm implemented are as follows:

1. Select two large prime numbers p and q.
2. Compute n = p*q.
3. Select a value e that is relatively prime and is less than p*q. The public key is e, n.
4. The Extended Euclidean Algorithm is used to find a suitable d value, see the next section for more detailed information of how this is implemented.
5. To perform encryption, raise a value V to the power of e and obtain the modulus n of the result to produce cipher text C. This looks as follows:

   C = ((V^e) mod n)
6. To perform decryption, raise the cipher text C to the power of d and obtain the modulus n of the result to produce the original value V. This looks as follows:

   V = ((C^d) mod n)

The public-key pair is e and n and the private-key is d. The public-key pair is used by the NFS server to validate file operations and is made freely available. The private-key d is stored internally by CryptosFS. The private-key is given to other users to allow them perform write operations on files.

4.3.4.1 RSA asymmetric-key Generation in CryptosFS

The GNU MP library provides almost all of the functionality necessary to produce the RSA asymmetric-keys with the exception of the Extended Euclidean Algorithm. This necessitated the design and implementation of a version of the Extended Euclidean Algorithm. The theoretical basis for the algorithm is obtainable from a description written by Bruce Ikenaga as detailed in “An Example Using the Extended Euclidean Algorithm” [36].

The RSA asymmetric-keys are generated using the algorithm as detailed in the previous section. The two prime numbers p and q are produced using the GNU MP prime number generator function. The prime number generator function uses a base number as a starting point. It retrieves the next prime number greater than the base number. The first base number is generated using the GNU MP random number generator seeded with the system time (jiffies). The second call to the prime number generator uses the first prime number generated as the base number. This allows two different prime numbers to be generated.

4.4 Storage of Keys Generated in CryptosFS

The keys generated in CryptosFS are stored in a structure called cryptos_keys_t that is located in the vnode in the VFS layer. The vnode information in the VFS layer is only active in the memory of the computer. Some means of storing the keys to provide persistence is needed.

4.4.1 Link-list Implementation in CryptosFS

A link-list implementation stores the keys in the computers main memory as they are generated by the CryptosFS file system. Each of the keys generated is stored in a link and the links are connected together to produce a link-list. By structuring the link-list

in this way, a clean separation of the implementation of the link-list and the information that it stores is possible. Modification of the link-list implementation is possible without affecting the structure that stores the key details. The following diagram shows the structure of the key-object that is stored in each link of the link-list.

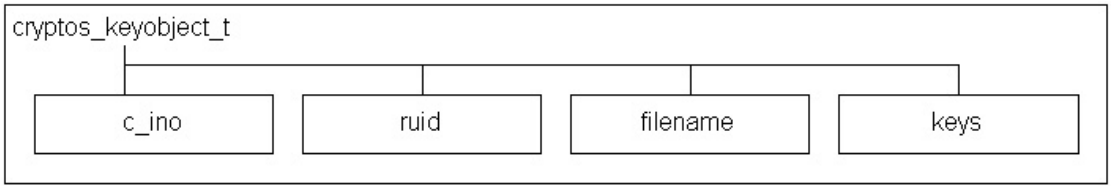

Figure 10: Structure of the key object used in CryptosFS.

The fields in the key object include:

- c_ino - the vnode number of the file that the key is created for.
- ruid - the user id of the user who created the file.
- filename - the name of the file that the keys are created for.
- keys - a cryptos_keys_t structure, this contains the symmetric and asymmetric-keys (see section 4.3 for more details).

The links in the link-list used in CryptosFS are arranged to form a doubly linked-list. The pointer “previous_link” at the start of the list always points to NULL, as does the pointer “next_link” at the end of the list. Functions were written to search the list, add links, remove links and destroy the list. The keys generated for each user are written to a user file when the CryptosFS kernel module is unloaded. The following figure shows the layout of the link-list.

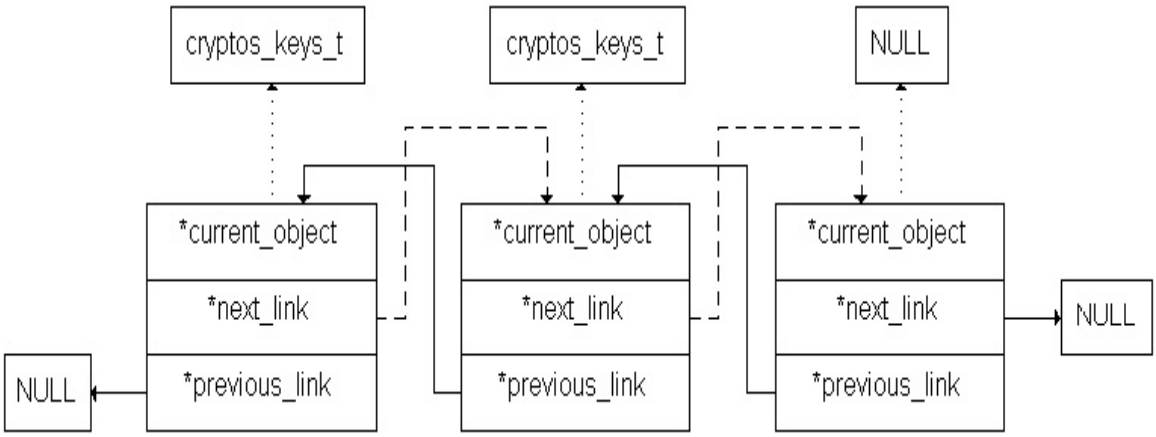

Figure 11: Doubly linked-List implemented in CryptosFS to store the generated keys.

By placing the generated keys in a link-list, it is possible to dump the contents of the link-list to a file and so store and retrieve the keys generated. The unloading of the link-list is performed when the kernel module for CryptosFS is removed from the system. Conversely, when the kernel module for CryptosFS is inserted in to the system a file is read to retrieve the keys for each vnode. Sufficient information is stored in each file to differentiate between each vnode and the user associated with it. The mounting of the CryptosFS file system is only possible by the root user and it is in the root users account that the file containing the mount keys is stored.

4.4.2 Key Files in CryptosFS

The doubly linked-list implemented in CryptosFS allows sufficient information about the keys generated for each vnode to be stored to ensure the persistence of the keys. Persistence of the keys is guaranteed by writing the contents of the link-list out to a file for each user id.

The contents of each cryptos_keyobject_t structure contained in the link-list in CryptosFS is written out to a file by user id and read in by user id. Concatenating the following pieces of information produces the key-file names.

- homedir - home directory of the user.

- . + passwd_dir - Set to ".gubu/"
- . + hostname: - Hostname of the client set by the root user when installing the CryptosFS file system kernel module.
- userid: - Operating system id of the user.
- username - Operating system username of the user.

An example key filename for the root user on the host bonny is “/root/.gubu/.bonny:0:root".

The key information stored in the key-files is the same information that is stored in the key-objects of the link-list. The separator character "|" is used to delimit each field in the key-file. The following shows the field layout for the key-file.

Figure 12: Field description of CryptosFS key file.

The necessity to store the keys generated by CryptosFS could have been facilitated by modifying the underlying ext2 file system used by Linux. Modification of the ext2 file system code was considered but rejected because of the additional time to perform the necessary analysis.

4.4.3 File Operations Implemented by CryptosFS in the Linux Kernel

Normally file operations are called using the system call interface that routes the calls from a user process through the VFS layer to the kernel. As CryptosFS stores the keys generated for each user in a flat file in the users home directory, a mechanism is needed to manipulate the information stored in these files from inside the kernel. The system call interface is only available to user space processes.

It is possible to perform system calls to execute the required file operation by creating a user process to perform the file operation. This results in an additional context switch from the kernel to the user process and then back in to the kernel. This approach is not practical because of the performance penalty it imposes. As the code for the VFS file is available, it is possible to copy sections of this code to implement the necessary file operations. The following file operations are implemented in CryptosFS.

open	close
read	write
mkdir	rm

Figure 13: File operations implemented for key-files in CryptosFS.

4.5 NFS Client

The Linux NFS client is implemented in Linux as a part of the kernel. The CryptosFS implementation modifies the NFS protocol used by the NFS client to pass digital signatures. The digital signatures are used by the server to validate client requests for file operations. The client uses the digital signatures to validate the results returned from the NFS server.

4.5.1 NFS Client Read Operations in CryptosFS

The NFS read operations on the client are modified to generate a digital signature as follows.

1. Pass an encrypted string of information to the MD5 message digest function. The string is in encrypted format because the CryptosFS layer encrypts the data before passing it to the NFS client.
2. The MD5 message digest function creates a 128-bit message digest of the information.

3. Pass the message digest to the RSA encryption function.
4. The RSA encryption function uses the public-key of the file to produce a digital signature.

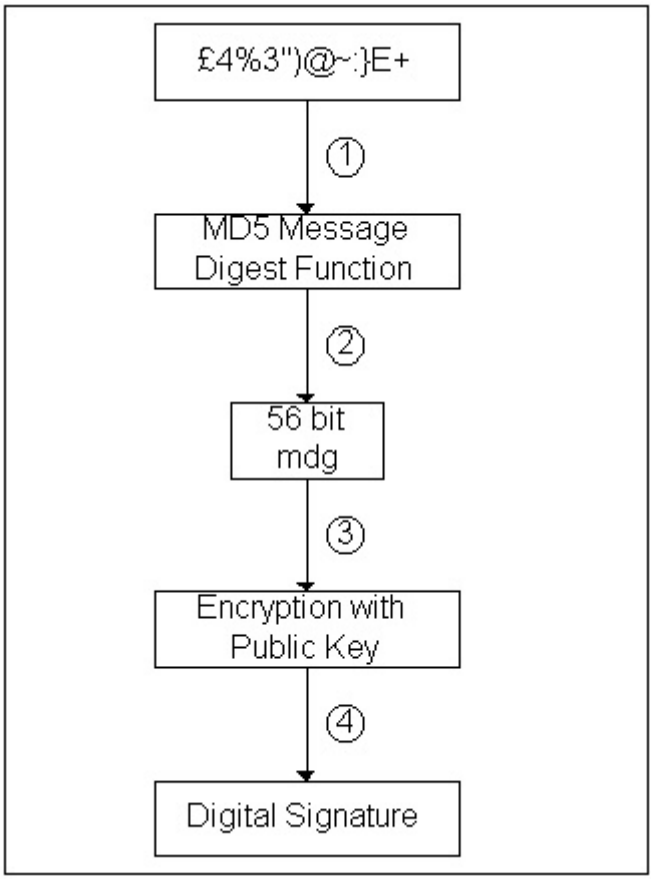

Figure 14: NFS Client creates a digital signature for a read operation.

The client uses the public-key to create a digital signature which is used to authenticate read operations as shown in the previous figure. Read operations include file reads and directory lookups. The NFS server validates the read operations by decrypting the digital signature[3].

4.5.2 NFS Client Write Operations in CryptosFS

The only difference between the NFS clients read and write operations in CryptosFS is that the write operation uses the private-key of the RSA asymmetric-key pair. Figure 14 looks the same for a write operation except that the private-key is used to

[3] See section 4.6 for more information on how the NFS server validates the NFS read operations.

perform the encryption. The write operations include file writes, file creation, file removal, file copy, file move, directory creation and directory removal. The NFS server validates the write operations by decrypting the digital signature[4].

4.5.3 NFS Client Result Validation in CryptosFS

When the NFS server validates and performs an operation requested by the NFS client, it creates a digital signature from the results of the operation. The server returns the results data and the digital signature to the client. The NFS client validates the results data received from the NFS server as follows.

1. The results data received from the server is passed to the MD5 message digest function on the client.
2. The MD5 message digest function creates a 128-bit message digest from the results data.
3. The 128-bit message digest is passed to the RSA encryption function.
4. The RSA encryption function uses the public-key of the file to create a digital signature. This digital signature is the client's digital signature as the client created it.
5. The client's digital signature and the server's digital signature are passed to a comparison function to check whether they match.
6. The digital signatures do not match so the results data is not valid.
7. The digital signatures match so the results data is valid.

[4] See section 4.6 for more information on how the NFS server validates the NFS write operations.

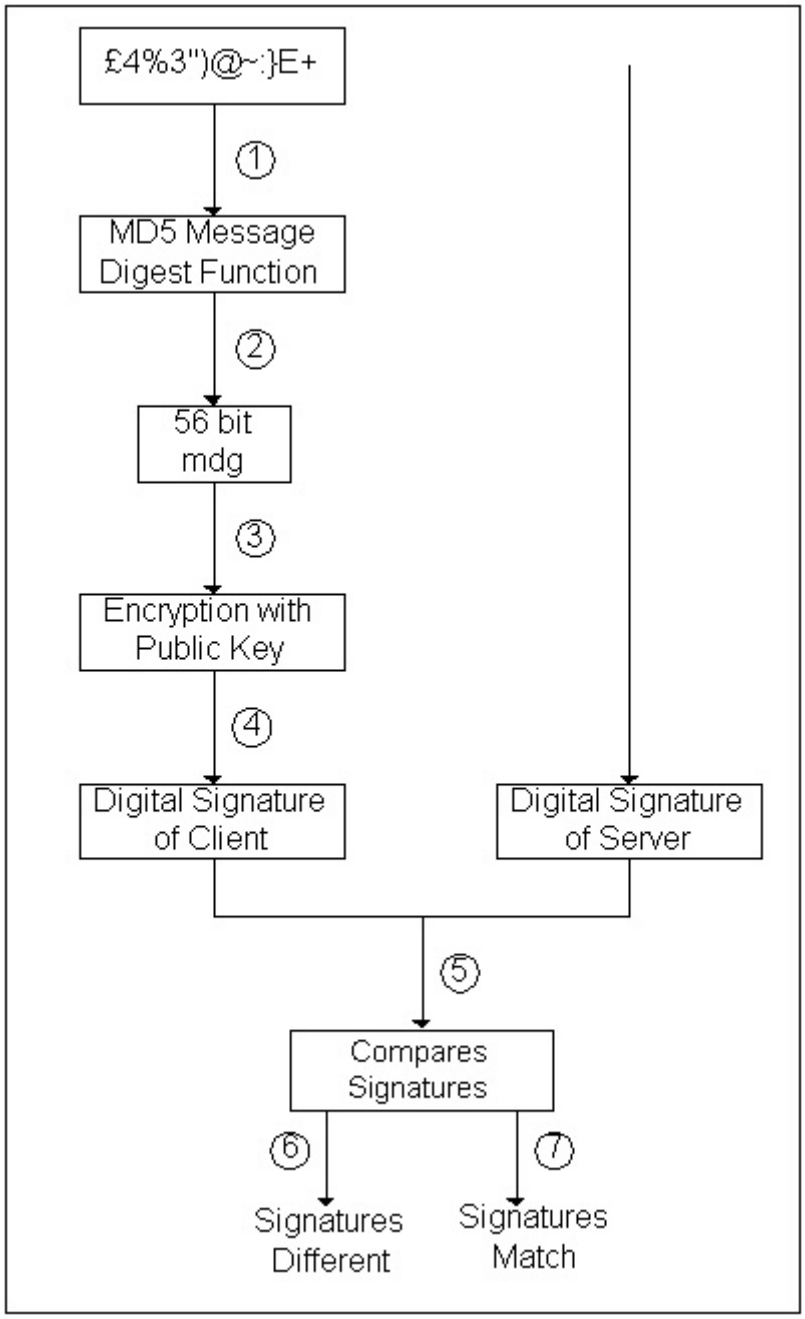

Figure 15: NFS Client validating result data from NFS Server.

4.6 NFS Server

The Linux NFS server is implemented in Linux as a part of the kernel. The CryptosFS implementation modifies the NFS protocol to allow digital signatures to be exchanged between a client and server. The digital signatures allow a server to validate clients requests for file operations. They also allow a client to validate the results returned by a server.

4.6.1 NFS Server Validation of Read Operations in CryptosFS

The validation of a read operation request on the NFS server uses virtually the same method as that used by the NFS client when validating the results of an operation received from the server. Figure 15 shows the process. The only difference is that in the validation of a read request on the server, it is the server that generates the digital signature. The server can validate the read operation because only a client with the correct public-key can create the correct digital signature.

4.6.2 NFS Servers Validation of Write Operations in CryptosFS

The NFS write operations on the NFS server are modified to validate the digital signature received from the NFS client. The write operation differs from the others described in that it decrypts the digital signatures created to retrieve the message digest contained in it.

1. The same encrypted data used to create the digital signature on the client is passed to the MD5 message digest function on the server.
2. The MD5 message digest function creates a 128-bit message digest from the encrypted data received from the client. This is called the server's message digest.
3. The digital signature received from the client is passed to the RSA decryption function.
4. The RSA decryption function uses the public-key of the file to decrypt the client's digital signature and retrieve the message digest contained in it.
5. The client's message digest and the server's message digest are passed to a comparison function to check whether they match.
6. The message digests do not match and the write operation fails because the client does not possess the correct private-key.

7. The message digests match so the write operation can proceed. Only a client that possesses the private-key can generate the correct digital signature.

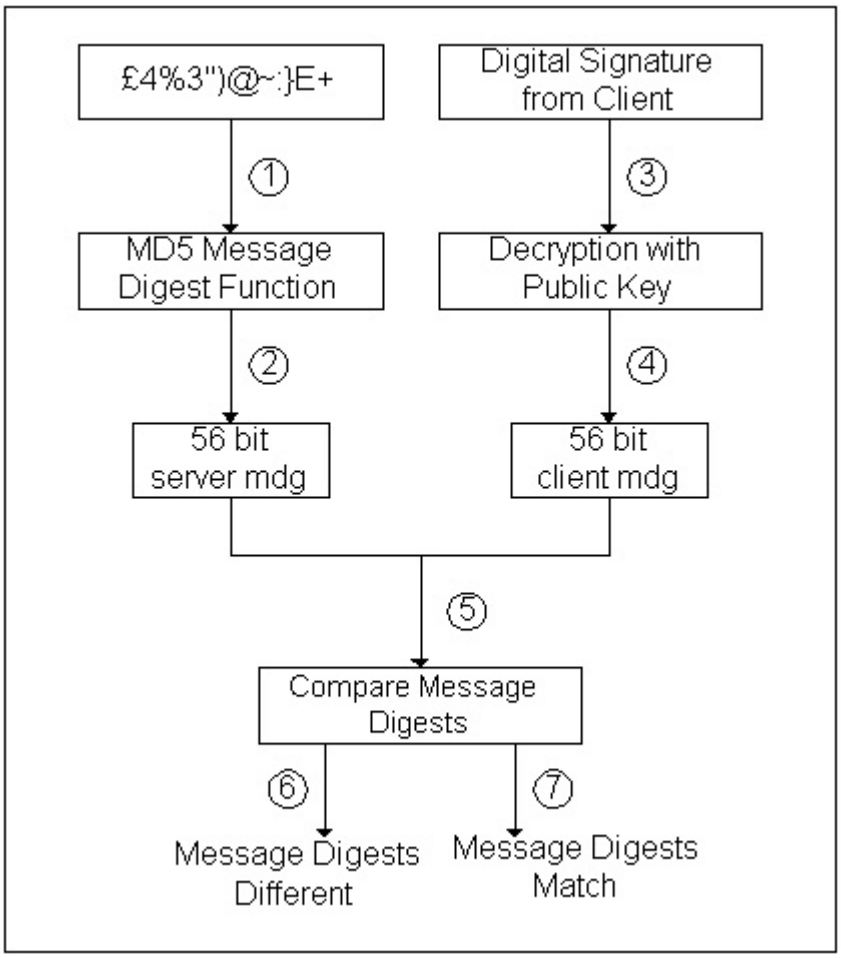

Figure 16: NFS Server validates a digital signature for a write operation.

4.6.3 NFS Servers Authentication of Results in CryptosFS

The results data returned by NFS server operations are modified to include a digital signature. The creation of the digital signature by the server follows the same procedure as that performed by the NFS client when performing a read operation. Refer to section 4.5.1 for more information.

4.7 XDR and RPC in NFS

4.7.1 Overview of Secure RPC in NFS

The original implementation of NFS has significant problems with security, as it does not authenticate the identity of users making requests. This makes it relatively easy to

forge requests being sent to server. Sun modified the NFS protocol to use both asymmetric-key (Diffie Hellman) and symmetric-key (DES) encryption to authenticate users instead of the machines that they are logging in from. This process is described in detail in “Secure Networking in the Sun environment”.

The application of symmetric-key cryptography (DES) to a time stamp allows the secure transmission of information in the network. The DES algorithm is no longer secure as it is open to attack from different techniques. Distributed computing has the ability to provide supercomputing levels of processing power to groups such as the Digital Frontier, allowing DES and other cryptographic algorithms to be cryptanalysed. The weaknesses of DES have been discussed in “Differential Cryptanalysis of the Data Encryption Standard”.

NFS uses secure RPC for Network Services. Secure RPC uses DES in Secure NFS. The DES authentication is designed around the ability of a sender to encrypt the current time in a message and transmit it to the receiver. The receiver decrypts the message, removes the time and compares it against her own clock. The authentication process requires that the client and the server have access to the same time. If they don't have a synchronised view of the time then the client requests a copy of the server’s time and calculates the difference between its local time and the server’s time. The client uses the difference between the two times to offset its clock value when computing timestamp values.

In our discussion of how the secure RPC mechanism works, the term encryption-key refers to the public-key of the asymmetric-keys. Similarly, the term decryption-key refers to the private-key of the asymmetric-keys. Figure 17 illustrates the

authentication process in detail. Two parties are involved in the communication, client X and server Y.

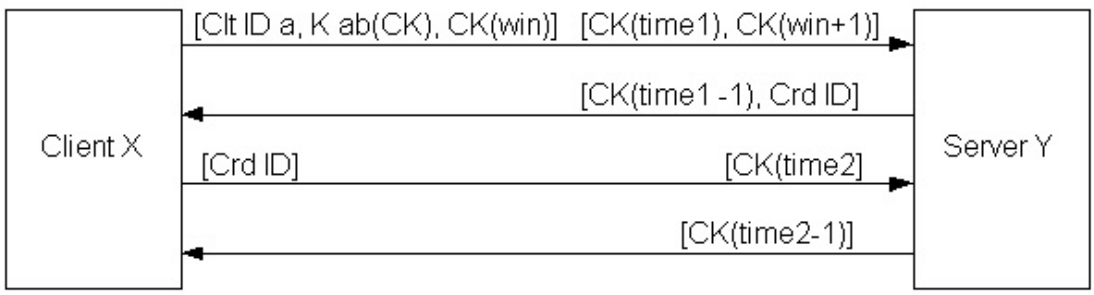

Figure 17: Secure RPC validation in Secure NFS.

The authentication process requires the client and the server to agree on a secret key. The client, trying to communicate with the server, produces a random key to encrypt the timestamps. The random key is called the *"Conversation Key"*. The client encrypts the Conversation Key using the encryption-key of the server. The client knows that the server is the only one who possesses the corresponding decryption-key. This ensures that only the client can decrypt the Conversation Key. The client then sends to the server a message containing the client ID, the encrypted conversation-key and a window value encrypted with the conversation-key. The server, upon receipt of the message, identifies the client from the client ID of the message.

The server uses its decryption-key to decrypt the encrypted conversation-key. Once the server has decrypted the conversation-key, it uses the conversation-key to decrypt the window value. The window value allows the server to verify that the messages it receives are valid. The server authenticates the message by checking that its current time is between the timestamp and the timestamp plus the window. If this is not the case then the server rejects the message. The server now stores the following information in its credential table.

1. Client ID
2. Conversation Key for the client

3. The window.
4. The timestamp of the last message. (This is used to prevent replays of messages.)

If the server authenticates the client successfully, it returns to the client an ID into the credential table and the clients timestamp encrypted with the conversation key CK. The client can verify that the information is from the server as the server is the only one has the client's timestamp. In subsequent communications between the client and the server, the client uses the client ID to identify itself to the server.

4.7.2 XDR/RPC in CryptosFS

The RPC mechanism used by the NFS client and server specifies the NFS protocol using XDR (External Data Representation Standard) as specified in RFC 1832 [37]. XDR is described in RFC 1832 as:

> "A standard for the description and encoding of data. It is useful for transferring data between different computer architectures, and has been used to communicate between such diverse machines as the SUN WORKSTATION*, VAX*, IBM-PC* and Cray*.

The implementation of the NFS client and server in Linux uses XDR/RPC to ensure architecture independence. The CryptosFS prototype uses the Linux NFS implementation. The RPC structures in CryptosFS are modified to accommodate the digital signatures and public-keys that are transferred during an NFS file operation. This requires increasing the amount of space allocated for the relevant argument structures.

Each of the NFS commands specified in the NFS protocol has its arguments and results encoded and decoded using XDR. The following diagram shows how the RPC

mechanism encodes and decodes the arguments and results transferred between the between the NFS client and NFS server.

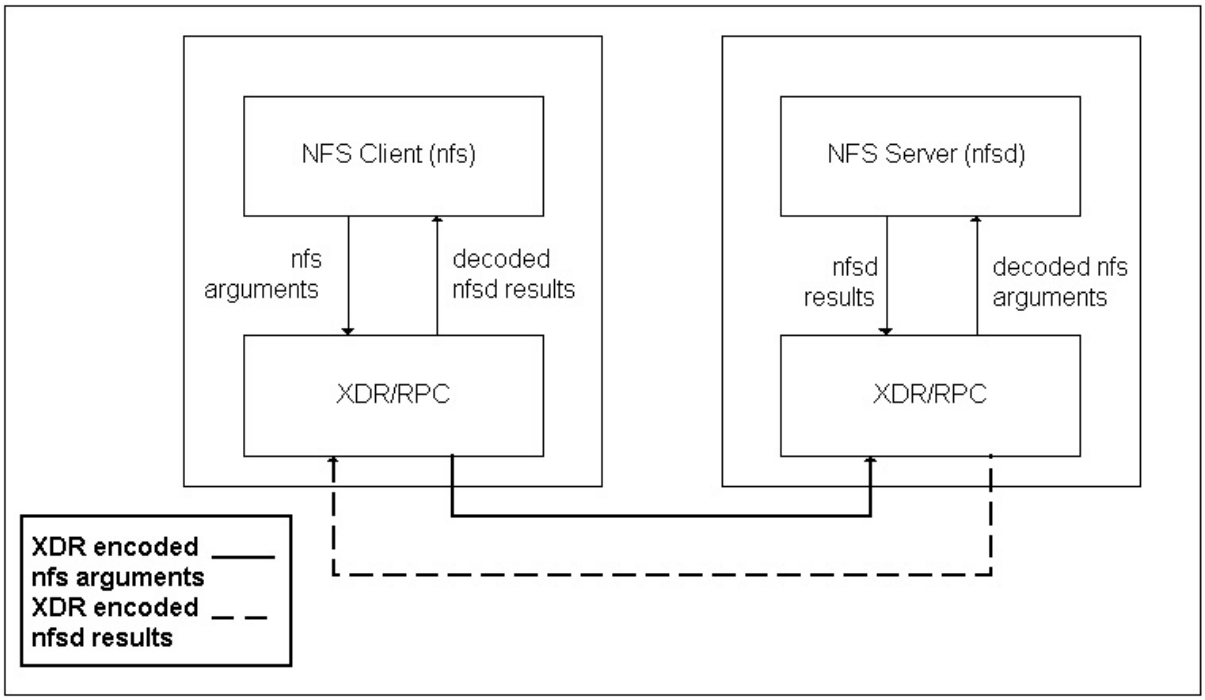

Figure 18: XDR/RPC Mechanisms for NFS Client and Server Communication.

5 Performance Evaluation of CryptosFS

5.1 Analysis of CryptosFS

The evaluation of the performance of the CryptosFS file system is one of the goals of this thesis. The CryptosFS file system is implemented in the kernel because of the requirement for speed. A user process file system does not deliver adequate performance. Development of the CryptosFS file system in the kernel complicates the development process in two fundamental ways.

1. The kernel libraries are limited in the functionality that they provide. The limited kernel functionality means functionality that is normally available from user space C libraries has to be implemented in the kernel. This slows the development process down.
2. Developing in the kernel is sensitive to coding and logic errors. If a section of code has a memory error and it is loaded in to kernel, it has the potential to halt the machine with a kernel segmentation fault when it is executed.

Finding and correcting errors in kernel code is made more difficult by the limited debugging facilities that the kernel provides. The CryptosFS prototype uses a large amount of C code that relies heavily on pointer manipulation. Any errors in the kernel code results in the kernel stopping, requiring the machine to be rebooted. This makes the development process extremely slow and laborious. Rebooting the machine after the operating system has crashed runs the risk of damaging the hard-drive of the machine. This is due to the unstable state that the file system is left in when an error occurs in the kernel.

While developing the CryptosFS prototype, two hard-drive failures occurred. The first failure occurred a month in to the implementation while the second occurred a month

before finishing the implementation. It was possible to procure a replacement for the initial failure but it was not possible to replace the second failure. This resulted in reduced capability to perform a rigorous performance evaluation of the CryptosFS prototype.

Even-though the hardware to perform a rigorous evaluation of the CryptosFS implementation was not available; a series of micro benchmarks of the different cryptographic algorithms implemented in CryptosFS was produced. The micro benchmarks were performed on the following processes.

1. RSA asymmetric-key generation.
2. Blowfish symmetric-key generation.
3. Generation of a 128-bit message digest.
4. Encryption of a 128-bit message digest using 1024-bit asymmetric-keys.
5. Decryption of a 128-bit message digest using 1024-bit asymmetric-keys.
6. Digital signature creation using 1024-bit asymmetric-keys.
7. Validation of digital signatures using 1024-bit asymmetric-keys.

5.2 Micro Benchmark Process

The micro benchmarks of the different cryptographic algorithms in CryptosFS were produced on a Pentium machine running RedHat Linux 6.1 with 64MB of RAM. The NFS client and the server were compiled in to the kernel and were set up on the same machine. The loopback device was used to simulate the communication mechanism. The CryptosFS kernel module was loaded into the running kernel. The machine was run in multi user mode with a light loading.

The micro benchmark calculations were produced by looping through a number of executions of the cryptographic process in a single run. Ten different runs were

performed and an average figure for each of the ten runs was calculated. The timing measurements were produced by obtaining the number of jiffies before each run started and the number of jiffies after the run had finished. This provides us with a mechanism of measuring the time that does not interfere with the kernel operation.

The cryptographic algorithms implemented in the kernel lock the kernel before they start and unlock the kernel once finished. This allows complete control of the kernel during execution. This also prevents them being interrupted and results in faster execution than a corresponding user space process.

5.3 Results of Micro Benchmarks

5.3.1 RSA Asymmetric-key Generation

The following table shows the average time in milliseconds to create an asymmetric-key pair in the kernel. The generation of RSA asymmetric-keys does introduce an overhead as the time required to generate the 1024-bit keys is on average 2.04 seconds. The keys are generated when a vnode is allocated for a file or for a data block.

Execution Number	1	2	3	4	5	6	7	8	9	10
Total Running Time per 100 Executions in Milliseconds	180900	198495	157905	203360	218400	209053	242600	219240	185640	221886
Average Execution Time in Seconds	1.81	1.98	1.58	2.03	2.18	2.09	2.43	2.19	1.86	2.22
Average Execution time in Seconds for 10 runs	2.04									

Table 1: Time to create a 1024-bit key RSA key.

5.3.2 Blowfish Symmetric-key Generation

Initially the number of executions performed was 100 but this always produced an average execution time of zero. The number of executions was increased to 10000 to

increase the probability of producing meaningful figures. The average time to produce a symmetric-key is 0.00321 milliseconds. This shows the overhead for the generation of each of the symmetric-keys is negligible and does not introduce a significant overhead. Each symmetric-key is generated when a vnode is allocated for a new file or for a data block.

Execution Number	1	2	3	4	5	6	7	8	9	10
Total Running Time per 10000 Executions in Milliseconds	33	32	32	32	32	32	32	32	32	32
Average Execution Time in Milliseconds	0.0033	0.0032	0.0032	0.0032	0.0032	0.0032	0.0032	0.0032	0.0032	0.0032
Average Execution time in Milliseconds for 10 runs	0.00321									

Table 2: Time to create a 448-bit Blowfish key.

5.3.3 Generation of 128-bit Message Digests

Creation of a message digest is the first step in the creation of a digital signature. The cost of generating the message digests using 16-bytes and 1024-bytes of input data was measured. These represent a 16-byte filename and a 1024-byte data block respectively.

5.3.3.1 Generation of Message Digest from 16 bytes of Input Data

The message digest is created from a filename of 16-bytes in length. The timing figures illustrated in table 3 shows the overhead of producing a message digest for a 16-byte input is 0.00107 milliseconds, this does not represent a significant overhead.

Total Running Time per 10000 Executions in Milliseconds	11	11	11	11	11	11	10	10	10	11
Average Execution Time in Milliseconds	0.0011	0.0011	0.0011	0.0011	0.0011	0.0011	0.0010	0.0010	0.0010	0.0011
Average Execution time in Milliseconds for 10 runs	0.00107									

Table 3: Time to create a 128-bit message digest from 16-bytes input.

5.3.3.2 Generation of Message Digest from 1024 bytes of Input Data

To simulate the cost of generating a message digest for a data block of 1024 bytes a series of tests using a 1024-bytes of data as input to the message digest function was executed. The figures in table 4 show the cost of generating a message digest of 1024 bytes is 0.9505 milliseconds.

Execution Number	1	2	3	4	5	6	7	8	9	10
Total Running Time per 10000 Executions in Milliseconds	9350	9204	8901	9754	9823	9234	9564	9678	9780	9765
Average Execution Time in Milliseconds	0.9350	0.9204	0.8901	0.9754	0.9823	0.9234	0.9564	0.9678	0.9780	0.9765
Average Execution time in Milliseconds for 10 runs	0.9505									

Table 4: Time to create a 128-bit message digest from 1024 bytes input.

5.3.4 Encryption of message digest using 1024-bit Asymmetric-keys

The second part of the process of creating a digital signature involves encrypting a 128-bit message digest with the public or private-key of the RSA asymmetric-key pair. The public-key is used for creating digital signatures for read operations and the private-key is used for creating digital signatures for write operations.

5.3.4.1 Encryption of 128-bit message digest using 1024-bit public key

Table 5 shows the time in seconds required to encrypt a 128-bit message digest with a 1024-bit public-key. The overhead to encrypt the message digest is on average 7 seconds.

Execution Number	1	2	3	4	5	6	7	8	9	10
Total Running Time per 100 Executions in Milliseconds	613800	628200	631200	693600	701400	765600	763200	738000	704400	783600
Average Execution Time in Seconds	6.1	6.3	6.3	6.9	7.0	7.7	7.6	7.4	7.0	7.8
Average Execution time in Seconds for 10 runs	7.0									

Table 5: Time to encrypt a 128-bit message digest using a 1024-bit public-key.

5.3.4.2 Encryption of 128-bit Message Digest Using a 1024-bit private key

Table 6 shows the time in seconds required to encrypt a 128-bit message digest with the 1024-bit private key. The overhead to encrypt the message digest is on average 7.26 seconds.

Execution Number	1	2	3	4	5	6	7	8	9	10
Total Running Time per 100 Executions in Milliseconds	634025	750345	745001	689412	702367	749147	736902	749824	801346	704781
Average Execution Time in Seconds	6.34	7.50	7.45	6.89	7.02	7.49	7.37	7.50	8.01	7.05
Average Execution time in Seconds for 10 runs	7.26									

Table 6: Time to encrypt a 128-bit message digest using a 1024-bit private-key.

5.3.5 Decryption of Digital Signature using 1024-bit public-key

The validation of the digital signatures for a write operation involves decrypting the digital signature using the 1024-bit public-key. The following table shows the time in seconds required to decrypt a digital signature using a 1024-bit public-key. The average time to perform this operation is 9.26 seconds.

Execution Number	1	2	3	4	5	6	7	8	9	10
Total Running Time per 100 Executions in Milliseconds	855000	997500	912000	816000	810000	1104000	918000	1069500	874800	901800
Average Execution Time in Seconds	8.55	9.975	9.12	8.16	8.1	11.04	9.18	10.695	8.748	9.018
Average Execution time in Seconds for 10 runs	9.26									

Table 7: Time to decrypt a digital signature using 1024-bit public-key.

5.3.6 Digital Signature Creation

The process of creating a digital signature involves the creation of a message digest and the encryption of the message digest. A 1024-bit public-key is used to create a digital signature for a read operation and a 1024-bit private-key is to create a digital signature for a write operation. These figures show the total cost of creating a digital signature.

5.3.6.1 Creation Of a Digital Signature For a Read Operation

Table 8 shows the overhead for the creation of a digital signature for a read operation using a 1024-bit public-key. An average time of 11.289 seconds represents a significant overhead.

Execution Number	1	2	3	4	5	6	7	8	9	10
Total Running Time per 100 Executions in Milliseconds	1050000	1045000	1170000	1293750	1184900	1221800	911360	1224720	1060800	1126400
Average Execution Time in Seconds	10.50	10.45	11.70	12.94	11.85	12.22	9.11	12.25	10.61	11.26
Average Execution time in Seconds for 10 runs	11.29									

Table 8: Time to create a digital signature for a Read Operation.

5.3.6.2 Creation Of a Digital Signature For a Write Operation

Table 9 shows the overhead for the creation of a digital signature for a write operation using a 1024-bit private-key. An average time of 13.32 seconds represents a significant overhead.

Execution Number	1	2	3	4	5	6	7	8	9	10
Total Running Time per 100 Executions in Milliseconds	1402500	1494656	1380575	1031080	1352501	1388141	1378658	1134083	1418362	1336608
Average Execution Time in Seconds	14.03	14.95	13.81	10.31	13.53	13.88	13.79	11.34	14.18	13.37
Average Execution time in Seconds for 10 runs	13.32									

Table 9: Time to create a digital signature for a Write Operation.

5.3.7 Digital Signature Validation

As there are two types of digital signatures for the read and write operations, there are two validation processes.

5.3.7.1 Validation of a Digital Signature For a Read Operation

The validation of a digital signature created for a read operation using the public-key requires the creation of a message digest. The message digest is encrypted with the 1024-bit public-key to create a second digital signature. The two digital signatures are compared to see if they match. An average run time of 13.317 seconds was recorded for this process.

Execution Number	1	2	3	4	5	6	7	8	9	10
Total Running Time per 100 Executions in Milliseconds	1432945	1395723	1572348	1237948	1452629	1489256	1473578	1345123	1528362	1295734
Average Execution Time in Seconds	14.33	13.96	15.72	12.38	14.53	14.89	14.74	13.45	15.28	12.96
Average Execution time in Seconds for 10 runs	14.22									

Table 10: Time to validate a digital signature for a Read operation.

5.3.7.2 Validation Of a Digital Signature For a Write Operation

The validation of a digital signature created for a write operation requires the creation of another message digest. The digital signature to be validated is decrypted using the 1024-bit public-key and the two message digests are compared to see if they match. An average run time of 12.730 seconds was recorded for this process.

Execution Number	1	2	3	4	5	6	7	8	9	10
Total Running Time per 100 Executions in Milliseconds	1249500	1327031	1493275	1271837	1352501	1107425	1230362	1226435	1275478	1196208
Average Execution Time in Seconds	12.50	13.27	14.93	12.72	13.53	11.07	12.30	12.26	12.75	11.96
Average Execution Time in Seconds for 10 runs	12.73									

Table 11: Time to validate a digital signature for a Write operation.

6 Conclusion

The CryptosFS prototype implementation is a distributed file system that provides users with the ability to share files securely. CryptosFS provides end-to-end encryption of file data and file meta-data using symmetric-key cryptography. It uses asymmetric-key cryptography for validating reads and writes operations. The intervention of the system administrator is required to mount the CryptosFS layer and to recompile the operating system kernel to install the modified NFS client and server.

The server in CryptosFS does not validate the users identity to authenticate access to files as in other distributed file systems. CryptosFS uses asymmetric-key cryptography as a form of capability to control file access. Each file created on the server has a corresponding public-key. When a client makes a request to the server, it passes a digital signature to the server. The server uses the public-key to validate the request made by the client. Possession of the correct public-key allows a client perform a read operation; while possessing the private-key allows a client perform a write operation.

The client in CryptosFS does not need to establish trust with the server as it stores the information on the server in encrypted format. The use of encryption negates the need for the client to trust the server because even if the server allows unauthorised access to the data it is useless without the symmetric-key to decrypt it. The CryptosFS prototype offers users the ability to securely store files on a remote file system with the knowledge that they are safe from an attacker who can compromise the server or the communication link between the client and the server.

6.1 Related work

CryptosFS builds upon a large body of work that use stackable layers and stacking vnodes to allow the rapid development of file system functionality through the reuse of existing file systems. Cryptfs, another cryptographic file system, provides end-to-end encryption of file data and file meta-data. The granularity of encryption in Cryptfs is at the directory level. All files for a user are encrypted using a single key. Cryptfs is a kernel level module and requires system administrator assistance to install it.

CFS is a user level file system that encrypts files with symmetric-key encryption. CFS provides the ability to encrypt files on both local and remote directories. The encryption used in CFS changes the size of file data and file meta-data; this results in the encrypted data being bigger than the corresponding clear text.

TCFS is a user level file system that is made up of a modified NFS client and server and an RPC based attributes server. TCFS is only available for Linux; this means that both the client and the server must also run the Linux operating system. All files in TCFS are encrypted using a single key.

The Truffles file system is a distributed system that provides file sharing and replication functionality using the Fiscus file system. It does not require assistance from the system administrator. Truffles relies on centralised certification authorities to name users. The Truffles file system is implemented in the kernel.

6.2 Where does CryptosFS fit in?

CryptosFS provides users with the ability to securely store information on a remote server safe in the knowledge that the information is secure from compromise. Current trends in the growth of mobile communications as a mechanism to access the Internet

requires ever increasing data storage capabilities. The capabilities of current mobile devices such as PDA's are limited by power consumption, battery size, design, and CPU power and data storage capabilities. As mobile devices are limited in their capabilities, the demand for secure access to remote storage is sure to increase.

CryptosFS could help to satisfy this demand as it does not explicitly trust the server to store information securely. CryptosFS uses encryption to maintain the security of information stored on the server. CryptosFS can be used in conjunction with NFS on the local area network (LAN). Users who require higher security in a wide area network setting can use CryptosFS. It is not the intention for CryptosFS to replace NFS but instead provide another option for remote file access. Increasingly sophisticated techniques are being developed to break security mechanisms. For this reason it is important that the security of the keys used for encryption in CryptosFS is maintained.

The use of asymmetric-key cryptography in CryptosFS provides users with a mechanism to grant access to files with others by giving them a copy of the public-key for read access and the private key for write access. Distribution of the relevant keys allows users access to files on remote system. Most other distributed file systems use access control lists or some variation of it for providing protection of files. Access control lists are an example of a centralised system. Centralised systems suffer from problems of scalability, they are a single point of failure and they require complicated management. The CryptosFS prototype is a distributed file system that abolishes the traditional file access control mechanisms that NFS uses and replaces it with a distributed access control mechanism that relies on asymmetric-key cryptography.

The owner of the files in CryptosFS must distribute the symmetric-key before other users can decrypt the contents of the actual file. It is possible that an attacker could successfully corrupt the data on the server machine by overwriting it with garbage. The client in CryptosFS can detect this situation because it validates the information returned from the server by authenticating the blocks read using the public-key of the file.

6.3 Further work

The prototype of CryptosFS developed leaves plenty of room for improvement. The goal of future work should be to stabilise the prototype by fixing the shortcomings identified during the development process. The problems identified included:

1. CryptosFS uses the GNU MP library to generate large integers in the kernel. The large integers are essential for the production of the RSA asymmetric-keys. The GNU MP library is linked in to the CryptosFS module and results in a very large kernel module. The GNU MP library contains a significant amount of functionality that is not used by the CryptosFS prototype. This additional functionality could be removed without affecting CryptosFS. This is important because the kernel can't be swapped out to disk and memory use is at a premium.

2. The CryptosFS prototype has an error in the memory module of the GMP MP library. If an asymmetric-key size of over 1024-bits is specified, an internal array used in the key generation program causes a segmentation fault in the kernel. The reason for this is that the extended Euclidean algorithm performs numerous calculations on large integers and stores the results in internal arrays. The GNU MP package uses a memory function called "realloc" to increase or decrease the size of an array. This memory reallocation is done to

allow the library to allocate the exact memory required dynamically. When an asymmetric-key size of over 1024-bits is used the kernel tries to allocate more than the kernel limit of 128k of memory. Fixing this problem requires modification of the memory allocation routines of the GNU MP library to use virtual memory when the kernel memory limit is reached.

3. Currently the keys generated in CryptosFS are stored in a link-list in the memory of the kernel and are transferred into and out of a file from memory when the CryptosFS module is inserted and removed from the kernel. This method performs adequately for small numbers of files but it is not scaleable. A more efficient mechanism would require the modification of the ext2 file system to store the keys directly.

4. The current prototype of CryptosFS stores the keys generated for each vnode in clear text format. Applying encryption to the key files provides security of this information. A password program could be developed to allow decryption of these keys. This is a simple solution although it is not the preferred solution as the difficulty of compromising the system is reduced to the difficulty of breaking the password. A more secure solution would be to investigate the use of a magnetic card to store the access key on.

6.4 Conclusion

File system development is a slow and laborious process. It requires specialist knowledge of the operating system and can result in file systems that are not easily portable. The development of CryptosFS builds upon a large body of work on stackable layers performed over the last fifteen years. This includes the work of E. Zadok et al in “Cryptfs: A Stackable Vnode Level Encryption File System”, J.S

Heidermann in “Stackable Design of File Systems” [38] and J.S. Heidemann and G.J. Popek in “File System Development with Stackable Layers” [39].

Stackable layers are promoted as a means of developing file system functionality with the performance levels of a kernel implementation but with the ease of development of a user level file system. From our experience in developing the CryptosFS prototype, stackable layers do offer a viable alternative to developing file systems. Existing code from the Cryptfs implementation, and the Linux NFS client and server implementations were reused in the development of the CryptosFS prototype.

During the development of CryptosFS, a significant amount of time was saved by reusing existing code. This is one of the biggest advantages of using stackable layers and stacking vnodes in file system development. Reusing the functionality of the Cryptfs and NFS file system allowed effort to be concentrated on developing the specific functionality of the cryptographic file system.

The major problem of developing file system functionality in the operating system kernel is that the kernel is kept in memory. As the kernel can never be swapped out to disk memory use is at a premium in the kernel. The cryptographic functionality added to the kernel results in a kernel that is about 20% larger than traditional kernels. The reason is the GNU MP package. The GNU MP package contains a lot of extraneous functionality that could be removed.

Developing CryptosFS has been a worthwhile experience. It has allowed us to learn a great deal about file system implementation and the UNIX operating system. Our experience of file system development prior to the development of the CryptosFS prototype was limited. Using cryptography allowed the traditional access control

mechanisms to be bypassed. The CryptosFS prototype is a fast secure distributed file system that demonstrates how it is possible to use cryptography as a form of distributed access control.

Bibliography

[1] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, “Design and implementation of the Sun network Filesystem,” presented at the summer USENIX, 1985.

[2] B. Taylor and D. Goldberg, “Secure Networking in the Sun environment,” presented at the summer USENIX, 1986.

[3] R. L. Rivest, “The MD5 Message-Digest Algorithm,” Internet Activities Board RFC 1321, April 1992.

[4] M. Blaze, “A Cryptographic File System for Unix,” presented at 1st ACM Conference on Computer and Communications Security, 1993.

[5] G. Cattaneo and G. Persiano, “Design and Implementation of a transparent cryptographic file system for Unix,” Universita de Salerno, Salerno July 1997.

[6] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart, “The Echo Distributed File System,” Digital Equipment Corporation, Palo Alto, California September 1993.

[7] R. Card, T. Ts'o, and S. Tweedie, “Design and Implementation of the Second Extended Filesystem,” presented at Dutch International Symposium on Linux, The Netherlands, 1986.

[8] E. Zadok and I. Badulescu, “A Stackable File System Interface For Linux,” Columbia University, Technical Report 1998.

[9] Y. A. Khalidi and M. N. Nelson, “Extensible File Systems in Spring,” Sun Microsystems, Mountain View, CA, Technical Report SMLI TR-93-18, September 1993.

[10] R. G. Guy, J. S. Heidermann, W. Mak, T. W. Page Jr, G. J. Popek, and D. Rothmeier, “Implementation of the Fiscus Replicated File System,” presented at the summer USENIX Conference, Anaheim, CA, 1990.

[11] E. Zadok, I. Badulescu, and A. Shender, “Cryptfs: A Stackable Vnode Level Encryption File System,” Columbia University 1998.

[12] S. R. Kleiman, “Vnodes: An architecture for Multiple File System Types in Sun UNIX,” presented at the summer USENIX, 1986.

[13] National Bureau of Standards, “Data Encryption Standard,” U.S. Department of Commerce FIPS Publication 46, January 1977.

[14] P. Reiher, T. Page Jr, G. Popek, J. Cook, and S. Crocker, “Truffles - A secure service for widespread file sharing,” presented at the PSRG Workshop on Network and Distributed System Security, 1993.

[15] B. Schneier, “Description of a new variable-length key, 64-bit block cipher (blowfish),” presented at Fast Software Encryption, Cambridge Security Workshop Proceedings, Springer-Verlag, 1993.

[16] M. J. Weiner, “Efficient DES Key Search,” presented at Advances in Cryptology CRYPTO '93 Proceedings, 1993.

[17] E. Biham and A. Shamir, “Differential Cryptanalysis of the Data Encryption Standard,” presented at Springer-Verlag, 1993.

[18] M. Matsui, “Linear Cryptanalysis Method for DES Cipher,” presented at Advances in Cryptology CRYPTO, 1994.

[19] J. Linn, “Privacy Enhancement for Internet Electronic Mail: Part 1 - Message Encryption and Authentication,” DEC, Technical Report 1992.

[20] X. Lai and J. Massey, “A proposal for a new block encryption standard,” presented at Advances in Cryptology - Eurocrypt, 1990.

[21] B. Lampson, “Requirements and Technology for Computer Security,” in *Computers at Risk*, 1991, pp. 74-101.

[22] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining Digital Signatures and Public-Key Cryptosystems,” , 1977.

[23] Portable Applications Standards Committee (PASC), “Draft Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interfaces [C Language],” IEEE Computer Society May 12 1993.

[24] National Computer Security Centre, “Integrity in Automated Information Systems,” September 1991.

[25] J. B. Dennis and E. C. Van Horn, “Programming Semantics for Multiprogrammed Computations,” presented at ACM, 1966.

[26] A. S. Tanenbaum, R. van Renesse, H. van Staveren, and G. J. Sharp, “Experiences with the Amoeba Distributed Operating System,” presented at Communications of the ACM, 1990.

[27] D. Mazières, “Security and Decentralised Control in SFS Global File System,” in *Computer Science Department*: MIT, 1997.

[28] J. Regan, “CapaFS: A globally accessible file system,” in *Computer Science, Distributed Systems*. Dublin: University of Dublin, 1999, pp. 75.

[29] P. R. Zimmerman, *Pretty Good Privacy. In The Official PGP User's Guide*: MIT Press, 1995.

[30] B. Schneier, “Data Guardians,” in *MacWorld*, 1993, pp. 145-151.

[31] J. H. Saltzer, D. P. Reed, and D. D. Clark, “End-to-end arguments in system design,” *ACM Transactions on Computing Systems*, vol. 2, 1984.

[32] B. Schneier, *Applied Cryptography; Protocols, Algorithms, and Source Code in C*, 2 ed: IEEE, 1995.

[33] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, *Linux Kernel Internals Second Edition*, 2nd ed: Addison Wesley, 1998.

[34] A. K. Lenstra and P. Leyland, “Large Integer Package,” , 1997.

[35] T. Granlund, “GNU Multi Precision Arithmetic library,” , 2000.

[36] B. Ikenaga, “An Example Using the Extended Euclidean Algorithm,” , vol. 2000, 1998.

[37] R. Srinivasan, “XDR: External Data Representation Standard,” Sun Microsystems, Request for Comment 1832, August 1995.

[38] J. S. Heidemann, “Stackable Design Of File Systems,” University of California, Los Angeles, Technical Report UCLA-CSD-950032, September 1995.

[39] J. S. Heidemann and G. J. Popek, “File System Development with Stackable Layers,” University of California, Los Angeles, California, Technical Report CSD-930019, July 1993.

Appendix A: Components of CryptosFS prototype

The prototype implementation of CryptosFS contains the following components:

- bfkeys - Generates Blowfish symmetric keys
- cryptos_genasym - Generates RSA symmetric keys.
  - Generates and validates Digital Signatures.
- hashfuncs - Generates MD5 message digests.
- cryptos_list - Stores generated keys.
- cryptos_file - Writes keys from link list to a key file.
  - Reads keys from key file in to link-list.
- cryptos - Loadable Kernel Module provides file system functionality.
- gmp-3.0.1 - Kernel Modified version of GNU MP produces large integers in the Linux Kernel.
- nfs - Modified Linux Kernel NFS client creates digital signatures for file operation and for validation of results.
- nfsd - Kernel NFS server modified to validate file operations by decrypting digital Signatures on file operations and creating digital signatures for results.