Design Document
for
ANFS

Advisor:
Yuanfang Cai
Technical Stakeholder:
Bill Mongan

By:
Lee Trager
Arkadiy Sherman
Matt Wyatt
Harry Anuszewski
Mike Dalton
# Revision History

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Reason for Changes</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lee Trager, Harry Anuszewski, Mike Dalton, Arkadiy Sherman, Matt Wyatt</td>
<td>01/13/11</td>
<td>Initial version</td>
<td>1.0</td>
</tr>
<tr>
<td>Lee Trager, Harry Anuszewski, Mike Dalton, Arkadiy Sherman, Matt Wyatt</td>
<td>2/16/11</td>
<td>Added complexity sections and elaborated on the different modules</td>
<td>2.0</td>
</tr>
<tr>
<td>Lee Trager, Harry Anuszewski, Mike Dalton, Arkadiy Sherman, Matt Wyatt</td>
<td>2/27/11</td>
<td>Additional changes suggested by W. Mongan</td>
<td>2.1</td>
</tr>
<tr>
<td>Lee Trager, Harry Anuszewski, Mike Dalton, Arkadiy Sherman, Matt Wyatt</td>
<td>4/10</td>
<td>Changed the architecture pictures to be in B&amp;W</td>
<td>2.2</td>
</tr>
</tbody>
</table>
# Table of Contents

1 Introduction  
   1.1 Purpose  
   1.2 Scope  
   1.3 Design Goals  
      1.3.1 Redundancy  
      1.3.2 Encryption  
      1.3.3 Compression  
2 Overview  
   2.1 Description of Problem  
   2.2 Data/Control Flow Diagram  
   2.3 Functional Decomposition Diagram  
   2.4 Architectures of NFS and ANFS  
      2.4.1 Architecture of NFS  
      2.4.2 Architecture of ANFS  
   2.6 Existing Architecture Considerations  
   2.7 Object Model  
   2.8 Kernel User Space Communication  
3 Redundancy  
   3.1 Overview  
   3.2 Complexity of Algorithms  
      3.2.1 Verification of Data  
      3.2.2 Dijkstra’s Shortest Path Algorithm  
   3.3 Data Structures  
   3.4 Functions  
      3.4.1 Server Functions Added  
      3.4.2 Client Functions Added  
      3.4.3 User Land Tool Functions  
      3.4.4 Functions Modified  
4 Encryption  
   4.1 Overview  
   4.2 Complexity of Algorithms  
      4.2.1 Prime Number Generation - Primality Testing  
      4.2.2 RSA Public-private Key Pair Generation  
      4.2.3 AES Encryption/Decryption  
   4.3 Functions  
      4.3.1 Functions Added  
      4.3.2 Functions Modified  
      4.3.3 Functions Used from OpenSSL  
      4.3.4 Functions Used from Linux Kernel Crypto API  
      4.3.5 User Land Tool Functions Modified  
   4.4 Structures to be Used  
      4.4.1 OpenSSL  
      4.4.2 CryptoAPI
5 Compression
  5.1 Overview
  5.2 Complexity of Algorithms
  5.3 Functions
    5.3.1 Functions Added
    5.3.2 Functions Used From the zLib Library
    5.3.3 Functions Modified
    5.3.4 User Land Tool Functions Modified
  5.4 Configuration Flags
6 Acronyms
7 Glossary
8 Requirements Traceability Matrix
  8.1 Traceability by Design Component
    8.1.1 Redundancy
    8.1.2 Encryption
    8.1.3 Compression
1 Introduction

1.1 Purpose
The purpose of this document is to specify how the system will satisfy the requirements listed in the ANFS Requirements Document. This document describes the system architecture and specifies its components. Included is a description of the software, its desired purpose, and technical descriptions of the components used to fulfil the requirements.

1.2 Scope
This document will describe the details of implementation for the initial build of ANFS. Section 2 will layout the basic modules and architecture of ANFS and the following sections will provide a more detailed look at the individual modules for Redundancy, Encryption, and Compression. This document is intended to be viewed by the development team (currently Team ANFS), and any advisers or technical stakeholders associated with the project.

1.3 Design Goals

1.3.1 Redundancy
The Redundancy module will implement a RAID-1 style setup across multiple servers to provide real-time data protection. The module will allow current users of the standard NFS protocol to enhance their network file system by adding data redundancy across a theoretically unlimited number of servers.

1.3.2 Encryption
The Encryption module will implement an RSA key-based encryption algorithm to protect data as it is propagated across a network or the Internet during client to server or server to server interactions. The module will allow administrators to encrypt any or all network traffic to preserve the integrity of their data.

1.3.3 Compression
The Compression module will implement the zLib compression algorithm to compress data and reduce bandwidth across the network. The module will allow users to arbitrarily compress data to increase total throughput of the network file system.
2 Overview

2.1 Description of Problem

The NFS protocol was developed by Sun Microsystems in the 1980’s to address the problem of file storage over a network. It allows client systems to access data over a network as if the network were part of the local file system. This is accomplished by using a Remote Procedure Call (RPC) which allows the client machine to operate a subroutine on another machine across a network. This can be used to read or write files that are not on the local file system.

The current NFS protocol does not directly support encryption, compression, or redundancy. This means that NFS can be streamlined to use less bandwidth by implementing compression of data before transmission. In addition, data can be encrypted to prevent loss or malicious modification from man-in-the-middle attacks. Implementing redundancy in a RAID-1 like setup would prevent data loss in the event of a server failure.

2.2 Data/Control Flow Diagram

![Data/Control Flow Diagram](image)

Figure 2.2.1 Data/Control Flow diagram of ANFS.

The figure above shows how data can flow in ANFS. Data can, but doesn’t have to, flow from the client machine to the encryption module, compression module, or directly to a server where it can be propagated to other servers, depending on which modules are activated. Control in ANFS is defined as the ability to alter data and the control flow follows the same paths as the data flow described above.
2.3 Functional Decomposition Diagram

**Functions**

```c
int nfdsd_redunancy_init()
int nfdsd_redunancy_add_pristine(struct nfs_mnt_reqeust* srv)
int nfdsd_redunancy_add_new(struct nfs_server* srv)
int nfdsd_redunancy_add(struct nfs_server* srv)
int nfdsd_verify_files(struct nfs_server* srv)
VerifyList* nfdsd_list_files(struct vfs_mnt* mnt)
void nfdsd_find_hashes(VOID* args)
int nfdsd_find_hash(struct VerifyList* e)
int nfdsd_send_hash(struct VerifyList* e, struct nfs_server* srv)
int nfdsd_verify_hash(struct VerifyList* e, struct nfs_server* srv)
int nfdsd_send_file(struct VerifyList* e, struct nfs_server* srv)
int nfdsd_redunancy_remove()
int nfdsd_list_servers(int sock)
int nfdsd_check(struct nfs_server* srv)
int nfdsd_notify_servers(VOID* msg)
int nfdsd_notify_clients(VOID* msg)
int nfdsd_process_notify(VOID* msg)
int nfdsd_fwd_data(struct xdr_stream *xdr, struct rpc_rqst *req, struct nfs_readres *res)
int nfdsd_update_latency(struct RedundantList* e)
int nfdsd_req_latency_info(struct nfs_server* srv)
int nfdsd_send_latency_info(struct xdr_stream *xdr)

int nfdsd_redunancy_init()
int nfdsd_notify_servers(VOID* msg)
int nfdsd_process_notify(VOID* msg)
int nfdsd_redunancy_add(struct nfs_server* srv)
int nfdsd_redunancy_remove(struct nfs_server* srv)
int nfdsd_redunancy_list(int sock)
int nfdsd_redunancy_send(VOID* data)
struct RedundantList** nfdsd_redunancy_best_path()

int main(int argc, char** argv)
int addRedundant(char* addr)
int removeRedundant()
int listRedundant()

static void encode_write(struct xdr_stream *xdr, const struct nfs_writeargs *args,
struct compound_hdr *hdr)
static int decode_write(struct xdr_stream *xdr, struct nfs_writeres *res)
static initt__init init_nfsd(void)
```
Figure 2.3.1 Function Decomposition diagram of ANFS.
The figure above shows the functional decomposition of ANFS. The functions added, modified, and used by the redundancy, encryption, and compression modules are listed under their respective modules.
2.4 Architectures of NFS and ANFS

2.4.1 Architecture of NFS

Current Linux NFSv4 Architecture

- User's Space
  - ACLs
  - nfsd
  - IDMAP
  - Mount
  - Client

- Kernel's Space
  - NFS ACLs
  - VFS
  - RPC
  - FS
  - XDR
  - Lock Mgr
  - NFS v4
  - Compound
  - Kerberos
  - SPKM
  - gss
  - gssd
  - POSIX ACLs
  - TCP
  - IPv4
  - Drivers
  - Disks
  - Ethernet Cards

Figure 2.4.1.1 Architecture diagram of NFS.
<http://devresources.linuxfoundation.org/dev/nfsv4/site/architecture/NFSv4_current_arch.png>
The figure above show the current NFS architecture before the addition of the ANFS
2.4.2 Architecture of ANFS

Current ANFSv1 Architecture

Figure 2.4.2.1 Architecture diagram of ANFS. The figure above shows the architecture of NFS after the addition of the components of ANFS and breaks the data flow down into the user’s space, kernel’s space, and hardware.
Figure 2.5.1 Swim Lane diagram of ANFS.
The figure above shows the flow of data through the ANFS architecture, a read or write traveling from the client’s space to the server’s space to the hardware, and then a response traveling back.

2.6 Existing Architecture Considerations
The Linux kernel architecture is set up to be modular and to minimize dependencies. Attaching our project to the Linux kernel in modules will allow the features of ANFS to be easily enabled or disabled as necessary. This will not compromise the integrity of the current architecture as no further modules will rely on the ANFS modules.

2.7 Object Model

This project will be developed in C which is not an object oriented programming language and an Object Model is not applicable. In sections 3, 4, and 5, functions and data types will be broken down for each feature of ANFS.

2.8 Kernel User Space Communication

NFS uses Unix socket connections to communicate between user space and kernel space. ANFS will use this method to set all ANFS options from user space. Compression and encryption will both be set as configuration options set in /etc/exports. When that file is read by exportfs it will parse the arguments and pass the options to the kernel as is done with other options.

The redundancy module requires a specialized tool to interface with the kernel to access all features. This tool will use Unix socket connections in the same way exportfs does.
3 Redundancy

3.1 Overview
The redundancy module is responsible for all redundancy operations for both the client and server. This includes ensuring all data is identical between servers, all client write operations are written to all servers, and reading can be done from any server. The module will also ensure write operations are done to the server with the best connection to minimize response times.

3.2 Complexity of Algorithms

3.2.1 Verification of Data
For redundancy to work with two or more servers, the data they share must be identical. Since it’s impossible to know the state of the data on all servers at all times, checks must be performed to ensure all data is identical when a server is inserted into the redundancy loop.

Whenever a redundant ANFS system is set up, a server with the known pristine state must be chosen. Once the pristine server is chosen, an administrator can insert any number of servers into the redundancy loop. Before the new servers can be used by connected clients, however, the pristine server must run a check to ensure that the shared data is identical across the servers in the redundancy loop.

The Linux kernel already has a SHA512 hash function built in which can be used to fingerprint a file. This will be used to take a hash of each file being shared across the pristine servers by the server the file is on. To ensure that this is done in a timely fashion, the pristine server will send a list of all files and their proper locations to all the newly inserted servers. Any files on the new servers that are not on the pristine server will be deleted. The new servers will then take SHA512 hashes of all its files in the same order as the pristine server. The SHA512 hash of each file will then be sent to the pristine server for verification of against the pristine hash. In the case that the hash on the pristine server does not match the hash sent by the new server, the pristine server will push its copy of the file to the new server.

Users will have the ability to read and write any file on the pristine server while this hashing operation is taking place. Read operations will only involve pristine servers while write operations will cause the files being modified to be locked to prevent multiple users from modifying the files at the same time. SHA512 hashing will be postponed on any locked file until the lock is freed. Write operations will be pushed to the new servers assuming the files exist on the new servers. This will ensure that any file that has been verified will always be identical on all systems. For files that are not yet verified, this can prevent unnecessary copying during the hashing and synchronization process by updating the unverified files if they are already identical to the pristine server copies.
Once all files have been verified on the new servers, they are considered pristine. This way an administrator can insert additional servers into the redundancy loop.

3.2.2 Dijkstra’s Shortest Path Algorithm

In order for redundancy to work properly, any data sent to a server must be sent quickly enough to all the mirrored servers for a client connected to any of those servers to see the change almost instantly. When multiple servers are connected, they can be connected either in a LAN or over the Internet, in which case they require access to very high speed connections to minimize latency. Clients can be, but are not always, connected to servers via LAN and also need to be able to perform read and write operations in a timely manner. Read operations require connection to only one mirrored server and bandwidth usage is minimal, while write operations require connection to at least one server and require data modification to be propagated to all other servers, which greatly increases the amount of time and resources required for every single operation. This algorithm is designed to mitigate the effect of redundancy by minimizing the time necessary to propagate changes across the entire network.

Latency between each server and client will be stored after each network operation is performed. This data will be used to generate a map of the entire redundant network which will then be used to perform Dijkstra’s algorithm, which can be used to calculate the single-source shortest path in $O(n^2)$ time where $n$ is the number of mirrored servers. Servers with latency of less than 2ms between themselves will be assumed to be on the same network. When a write operation is performed the network will utilize the shortest possible paths to propagate the changes made by the client to all mirrored servers. The client will be able to request a server to forward the data it has received to another server in the network. A client will not be able to request another client to forward data as clients will not be aware of other clients on the network.

Figure 3.2.2.1 below shows two clients connected to the ANFS redundant network. Client A performs a write operation and needs to send its data to all mirrored servers. Without this algorithm it would take 122ms to write to each server cluster and for those server clusters to share the data amongst themselves. It is possible for the write operation to be performed simultaneously to both servers but this requires a robust network connection and the availability of a large amount of bandwidth. In addition, sending an increased amount of data increases the likelihood that data will be lost or slowed due to network congestion. This is especially relevant to clients utilizing an Internet connection provided by an ISP which sets usage limits on bandwidth or performs traffic shaping routines on high usage clients.

The optimal route for data would be for Client A to perform its write operation on either Network A or Network B and let the server clusters distribute the data to the network not written to. Every machine will generate and store a network map similar to the one shown in Figure 3.2.2.1 to calculate the shortest path between all machines on the ANFS redundant network.
3.3 Data Structures

3.3.1 Linked list element which stores the path to the file on the share and its hash.
struct VerifyList
{
    char* file;
    char* hash;
    struct VerifyList* next;
};

3.3.2 Linked list element which stores a pointer to the NFS server which is in redundancy.
struct RedundantList
{
    struct nfs_server* srv;
    int latency;
}
int verified;
int files;
struct RedundantList* next;
}

3.4 Functions

3.4.1 Server Functions Added

3.4.1.1 int nfdn_redundancy_init();
   Input: None
   Output: Function return code
   Description: Initialize data structures when running as a server

3.4.1.2 int nfdn_redundancy_add_pristine(struct nfs_mount_request* srv);
   Input: NFS server to add to the redundancy list
   Output: Function return code
   Description: While on a pristine server add another NFS to the list of redundant servers

3.4.1.3 int nfdn_redundancy_add_new(struct nfs_server* srv);
   Input: NFS server which is requesting we be added
   Output: Function return code
   Description: A pristine server contacts a server and requests it to be added

3.4.1.4 int nfdn_redundancy_add(struct nfs_server* srv);
   Input: New NFS server to be added into redundancy
   Output: Function return code
   Description: Add a server into redundancy which has already been verified by another redundant server

3.4.1.5 int nfdn_verify_files(struct nfs_server* srv);
   Input: NFS server to verify that all data is the same
   Output: Function return code
   Description: Verify that all data is identical on a remote server to the local server

3.4.1.6 VerifyList* nfdn_list_files(struct vfs_mount* mnt);
   Input: Exported directory that is being shared
   Output: List of files being shared
   Description: Find all of the files that are being shared and start a thread which finds the SHA512 sum of each file

3.4.1.7 void nfdn_find_hashes(void* args);
   Input: List of files being shared that need to have their SHA512 hash taken
   Output: None
   Description: Go through the list of shared files and find the SHA512 hash for each of them

3.4.1.8 int nfdn_find_hash(struct VerifyList* e);
   Input: VerifyList element to find the SHA512 value
   Output: Function return code
   Description: Find the SHA512 value of a file

3.4.1.9 int nfdn_send_hash(struct VerifyList* e, struct nfs_server* srv);
   Input: VerifyList element which contains the file and hash the new server wants to
confirm with the pristine server
Output: Function return code
Description: The new server sends a computed hash to the pristine server to verify that the data is correct
3.4.1.10 int nfssd_verify_hash(struct VerifyList* e, struct nfs_server* srv);
   Input: Data sent by the new server to verify locally
   Output: Function return code
   Description: Take the data sent from the new server and verify it is the same as the pristine copy
3.4.1.11 int nfssd_send_file(struct VerifyList* e, struct nfs_server* srv);
   Input: The file to send to the new server
   Output: Function return code
   Description: When the pristine server discovers that the hashes do not match send the correct copy to the client
3.4.1.12 int nfssd_redunancy_remove();
   Input: None
   Output: Function return code
   Description: Remove the machine from the list of redundant servers
3.4.1.13 int nfssd_list_servers(int sock);
   Input: Unix socket to send data to
   Output: Function return code
   Description: Send all redundant server info to the Unix socket
3.4.1.14 int nfssd_check(struct nfs_server* srv);
   Input: NFS redundant server which needs to be checked on
   Output: Function return code
   Description: If a client thinks the a server is unreachable another server can check on it to confirm
3.4.1.15 int nfssd_notify_servers(void* msg);
   Input: Message to send to all redundant servers
   Output: Function return code
   Description: Send a message about a server to all other known servers
3.4.1.16 int nfssd_notify_clients(void* msg);
   Input: Message to send to all connected clients
   Output: Function return code
   Description: Send a message about a server to all connected clients
3.4.1.17 int nfssd_process_notify(void* msg);
   Input: Message to process
   Output: Function return code
   Description: Process the sent notification and take appropriate action
3.4.1.18 int nfssd_fwd_data(struct xdr_stream *xdr, struct rpc_rqst *req, struct nfs_readres *res);
   Input: The XDR steam and data to foward
   Output: Function return code
   Description: Forward data to one or more other ANFS enabled servers
3.4.1.19 int nfssd_update_latency(struct RedundantList* e);
   Input: The server to update the latency info
   Output: Function return code
   Description: Update the latency information about a server
3.4.1.20 int nfsd_req_latency_info(struct nfs_server* srv);
   Input: Server to request latency information from
   Output: Function return code
   Description: Request from a remote server the latency info for all known servers

3.4.1.21 int nfsd_send_latency_info(struct xdr_stream *xdr);
   Input: XDR stream to reply to
   Output: Function return code
   Description: Send latency info about all servers to a server/client which has requested it

3.4.2 Client Functions Added

3.4.2.1 int nfs_redundancy_init();
   Input: None
   Output: Function return code
   Description: Initialize any data structs needed for redundancy

3.4.2.2 int nfs_notify_servers(void* msg);
   Input: Message to send to servers
   Output: Function return code
   Description: Send a message to all remote servers

3.4.2.3 int nfs_process_notify(void* msg);
   Input: Message to be proceeded
   Output: Function return code
   Description: Process the message the client has been sent by a server and take appropriate action

3.4.2.4 int nfs_redundancy_add(struct nfs_server* srv);
   Input: Server to add to redundancy list
   Output: Function return code
   Description: Once the client has been notified of an additional NFS server add it to the list of servers

3.4.2.5 int nfs_redundancy_remove(struct nfs_server* srv);
   Input: Server to remove from redundancy list
   Output: Function return code
   Description: Remove a server from the redundancy list

3.4.2.6 int nfs_redundancy_list(int sock);
   Input: Unix socket to send list to
   Output: Function return code
   Description: Send the list of known redundant servers to a Unix socket

3.4.2.7 int nfs_redundancy_send(void* data);
   Input: Data to send to all server
   Output: Function return code
   Description: Send modified data to all servers

3.4.2.8 struct RedundantList** nfs_redundancy_best_path();
   Input: None
   Output: An array of paths to send data to
   Description: Figure out the best way to get data to all servers given the known latencies

3.4.3 User Land Tool Functions
3.4.3.1 int main(int argc, char** argv);
   Input: User command line options
   Output: Program return code
   Description: Parse user arguments and notify the kernel of what he or she wants to do

3.4.3.2 int addRedundant(char* addr);
   Input: Server address that needs to be added into redundancy
   Output: Program return code
   Description: Add a server address to the list of redundant servers

3.4.3.3 int removeRedundant();
   Input: None
   Output: Function return code
   Description: Remove the system the user is on from the list of redundant servers

3.4.3.4 int listRedundant();
   Input: None
   Output: Function return code
   Description: Print out a list of redundant servers

3.4.4 Functions Modified

Functions will be modified in linux/fs/nfs/nfs4xdr.c and linux/fs/nfsd/nfsctl.c file to check if redundancy is enabled and then perform the necessary function calls.

3.4.4.1 static void encode_write(struct xdr_stream *xdr, const struct nfs_writeargs *args, struct compound_hdr *hdr);
   Description: This function will ensure data is sent to all servers

3.4.4.2 static int decode_write(struct xdr_stream *xdr, struct nfs_writeres *res);
   Description: This function will hand fault tolerance in the case a server fails

3.4.4.3 static init __init init_nfsd(void);
   Description: Initialization code will be called from here
4 Encryption

4.1 Overview
The encryption module is responsible for facilitating the encryption and decryption of data. We will use the OpenSSL API algorithms for generating public-private key pairs and the Crypto API for encrypting and decrypting with AES. The kernel will first access the user space and generate a public and private key with OpenSSL. Then the public key will be released so the server and client have the same key. After that the kernel will use the Crypto API to encrypt and decrypt with AES.

4.2 Complexity of Algorithms

4.2.1 Prime Number Generation - Primality Testing
When generating an RSA public-private key pair, primality testing is the source of complexity. A random n bit number is first generated and then it is tested to see whether or not it is prime. There are several algorithms to test if a number is prime. The one that OpenSSL uses is Miller-Rabin’s Probabilistic Primality Test. This algorithm runs in $O(k \cdot \log^3(n))$ where $n$ is the number of bits and $k$ is the number of iterations we use which is proportional to the accuracy of the test.

4.2.2 RSA Public-private Key Pair Generation
The generation of the keys’ main source of complexity is modular exponentiation. This algorithm will take $O(k^4)$ where $k$ is number of bits in the modules.

4.2.3 AES Encryption/Decryption
Most modern processors include an opcode for this so it will take $O(n)$ clock cycles where $n$ is the number of bytes.

4.3 Functions
The linux/fs/nfs/nfs4xdr.c file includes all the original nfsv4 data functions. It is necessary to modify this file to add our encryption module. The CryptoAPI headers will be added to the list of headers to grant access to the AES functions. It will also be necessary to access the user space in order to have access to the OpenSSL functions.

4.3.1 Functions Added
4.3.1.1 BIGNUM* generate_rsa_key_pair(int num);
    Input: number of bits
    Output: BIGNUM with a public and private key
    Description: generates a public private key pair with num bits
4.3.1.2 void encrypt_with_aes(BIGNUM* keypair, u8 private_key, u8 in, u8 out);
   Input: public private key pair, aes private key, plaintext input, encrypted output
   Output: nothing
   Description: encrypted plaintext
4.3.1.3 void decrypt_with_aes(BIGNUM* keypair, u8 private_key, u8 in, u8 out);
   Input: public private key pair, aes private key, encrypted input, decrypted output
   Output: nothing
   Description: decrypts encrypted text

4.3.2 Functions Modified
Functions will be added to the linux/fs/nfs/nfs4xdr.c file to check for the encryption flag and then
perform the necessary function calls.
4.3.2.1 static void encode_write(struct xdr_stream *xdr, const struct nfs_writable *args, struct
compound_hdr *hdr);
   Description: This function will call encrypt_with_aes to preform encryption if the
   encryption flag is is set
4.3.2.2 static int decode_write(struct xdr_stream *xdr, struct nfs_writeres *res);
   Description: This function will call decrypt_with_aes to preform decryption if the
   encryption flag is is set

4.3.3 Functions Used from OpenSSL
4.3.3.1 BIGNUM *BN_generate_prime(int num, BIGNUM *add, BIGNUM *rem);
   Input: number of bits num, BIGNUMs such that p % add == rem
   Output: BIGNUM that is prime
   Description: Generates a random prime number by generating random numbers until it
   verifies one with primality testing
4.3.3.2 RSA *RSA_generate_key(int num, unsigned long e);
   Input: number of bits num, public exponent e
   Output: RSA object that includes a public and private key pair
   Description: Generates a public private key pair that contain n bits and use the public
   exponent

4.3.4 Functions Used from Linux Kernel Crypto API
4.3.4.1 int crypto_aes_set_key(struct crypto_tfm *tfm, const u8 *in_key, unsigned int key_len);
   Input: crypto_tfm that is used, input key, and size of that key
   Output: 0 on success, flag is set otherwise
   Description: Sets the key used to encrypt or decrypt
4.3.4.2 static void aes_encrypt(struct crypto_tfm *tfm, u8 *out, const u8 *in);
   Input: crypto_tfm that is used, where to output encrypted, and plaintext
   Output: nothing
   Description: Encrypts data
4.3.4.3 static void aes_decrypt(struct crypto_tfm *tfm, u8 *out, const u8 *in);
   Input: crypto_tfm that is used, variable to output decrypted text, and encrypted text
   Output: nothing
   Description: Decrypts data
4.3.5 User Land Tool Functions Modified

Functions will also need to be modified in the user land tool exportfs in order for encryption to be enabled and turned on in the kernel these modifications will be done in nfs-utils/support/nfs/exports.c

4.3.5.1 static int parseopts(char *cp, struct exportent *ep, int warn, int *had_subtree_opt_ptr);
   Description: Needed to check for the compression configuration option

4.4 Structures to be Used

4.4.1 OpenSSL

4.4.1.1
   struct bignum
   {
       BNULONG *d; /* Pointer to an array of 'BN_BITS2' bit chunks. */
       int top; /* Index of last used d +1. */
       /* The next are internal book keeping for bn_expand. */
       int dmax; /* Size of the d array. */
       int neg; /* one if the number is negative */
       int flags;
   };

4.4.2 CryptoAPI

4.4.2.1
   struct crypto_tfm
   {
       u32 crt_flags;
       union {
           struct cipher_tfm cipher;
           struct digest_tfm digest;
           struct compress_tfm compress;
       } crt_u;
       struct crypto_alg *__crt_alg;
   };

5 Compressio

5.1 Overview
The compression module is responsible for facilitating the compression and decompression of data during transfers between client and server or server and server communication.

5.2 Complexity of Algorithms
The complexity of the compression module is determined by the complexity of the algorithms used in the zLib library. The zLib library uses dynamic Huffman codes to generate a Huffman tree. A Huffman tree is a binary tree that uses a priority queue data structure and efficient priority queue data structures require $O(\log n)$ time per insertion, where $n$ is the size of the file being compressed in bytes. For a full binary tree with $n$ leaves there are $2n-1$ nodes, and this algorithm operates in $O(n \log n)$ time.

The zLib library converts its output stream into blocks so that regions that can be compressed are and regions of uncompressed data are still included. The typical zLib compression ratios are on the order of 2:1 to 5:1. Since NFS’s biggest restriction is network latency, higher compression will use less bandwidth.

5.3 Functions
This section covers the functions that will be needed to compress and inflate data. This section will also list the files that will need to be updated or edited to support the new algorithms.

5.3.1 Functions Added
The linux/fs/nfs/nfs4xdr.c file includes all the original nfsv4 data functions. To add compression this file will need to be modified. The zLib header, linux/include/linux/zlib.h, will need to be added to the list of headers to grant access to the zLib compression and inflation functions.

5.3.1.1 zstream* Convert_XDR_to_Z(XDR_stream* strm);
   Input: Sun’s XDR data stream
   Output: zstream to be used by the compression library
   Description: The zlib library needs a zstream. ANFS will load the data to be transferred in xdr canonical format. This function will convert the XDR stream into a zstream to be compressed

5.3.1.2 XDR_Stream* Convert_Z_to_XDR(zstream* strm);
   Input: Pointer to the compressed data
   Output: Pointer to the XDR stream
   Description: After the standard XDR stream is compressed, zLib outputs a zstream. This zstream cannot be transferred over the network. The data needs to converted back to the XDR canonical format
5.3.2 Functions Used From the zLib Library

5.3.2.1 ZEXTERN int ZEXPORT deflateInit OF((z_streamp strm, int level));
   Input: Stream of data and level of compression
   Output: returns success or fail
   Description: Initializes the internal stream state for compression

5.3.2.2 ZEXTERN int ZEXPORT deflate OF((z_streamp strm, int flush));
   Input: Stream of date and and flush, how much data to compress before creating output.
   Output: Status of output
   Description: Compresses as much data as possible, and stops when the input buffer
   becomes empty or the output buffer becomes full. This function returns the output status.

5.3.2.3 ZEXTERN int ZEXPORT deflateEnd OF((z_streamp strm));
   Input: Data stream
   Output: Compressed data
   Description: Creates the compressed data

5.3.2.4 ZEXTERN int ZEXPORT inflateInit OF((z_streamp strm));
   Input: Data stream
   Output: Success or error
   Description: Determines the compression method from the zlib header and allocates all
   data structures accordingly

5.3.2.5 ZEXTERN int ZEXPORT inflate OF((z_streamp strm, int flush));
   Input: Data stream and Flush, How much data to process at a time.
   Output: Current progress if multiple parts, Success or fail
   Description: Decompresses data

5.3.2.6 ZEXTERN int ZEXPORT inflateEnd OF((z_streamp strm));
   Input: Stream to close
   Output: Program status
   Description: All dynamically allocated data structures for this stream are freed

5.3.3 Functions Modified

Functions will be added to the linux/fs/nfs/nfs4xdr.c file to check for the compression flag and
then perform the necessary function calls

5.3.3.1 static void encode_write(struct xdr_stream *xdr, const struct nfs_writeargs *args, struct
compound_hdr *hdr);
   Description: This function will call Convert_XDR_to_Z to perform compression.

5.3.3.2 static int decode_write(struct xdr_stream *xdr, struct nfs_writeres *res);
   Description: This function will call Convert_Z_to_XDR to perform inflation

5.3.4 User Land Tool Functions Modified

Functions will also need to be modified in the user land tool exportfs in order for compression to
be enabled and turned on in the kernel these modifications will be done in nfs-utils/support/nfs/
exports.c

5.3.4.1 static int parseopts(char *cp, struct exportent *ep, int warn, int *had_subtree_opt_ptr);
   Description: Needed to check for the compression configuration option
5.4 Configuration Flags

This section will cover the configuration flags that will be needed in order to configure the compression module. The file linux/fs/nfs/super.c contains the configuration flags for the compression module. The enum will need to be updated with the options:

5.4.1 Opt_compression_enabled
string Opt_compression_enabled: determines whether compression is enabled. The default value will be disabled

5.4.2 Opt_ratio
int Opt_ratio: A percent of how much data has to be compressed to justify continued compression

The static const match_table_t nfs_mount_option_tokens will have to be updated to set the default values for ANFS.
Opt_compression_enabled will be set to true
Opt_ratio will be set to 15, 15% reduction in size
## 6 Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>ANFS</td>
<td>Advanced Network File System</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>SHA</td>
<td>Secure Hashing Algorithm</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest, Shamir and Adleman algorithm for public-key cryptography</td>
</tr>
<tr>
<td>XDR</td>
<td>External Data Representation</td>
</tr>
</tbody>
</table>
7 Glossary

Administrator - The user on a system who has the ability to configure any part of the system.
Client - A machine which is connected to an ANFS server. All files are read and written to the server.
Compression - The process of encoding information using less bits than the original information would use.
Daemon - A process that runs in the background.
Encryption - The process of using an algorithm to make information unreadable by anyone except those possessing special knowledge.
Exportfs - An NFS user land tool which is used to pass configuration information to the kernel Mirrored Servers - Any other server connected and synchronized with the current server.
Network File System - File system that allows access to files from multiple hosts shared via a computer network.
OpenSSL - Toolkit that implements the SSL(Secure Sockets Layer) and TSL (Transport Layer Security) protocols. We use this library for its Cryptographic algorithms.
Pristine server - A server which is configured for ANFS but has not yet been synched with the data on other mirrored servers.
RAID 1 - A system used with computer hard drives in which the data written to each drive is identical. If one drive fails the system continues operating normally as it has another drive with the same set of data.
Redundancy - Repetition of information in order to avoid errors, computer failure, and slow connections.
Secure Shell - Network protocol that allows data exchange using a secure channel.
Secure Sockets Layer - A Handshake protocol that supports server and client authentication.
Server - The machine which has all of the data. It also keeps track of which servers are available and communicates with clients.
SHA512 - A hashing algorithm designed by the National Security Agency used to verify data integrity.
XDR Datastream - A data type standard for description and encoding of data (RFC 4506).
zlib - A software library used for data compression.
# 8 Requirements Traceability Matrix

## 8.1 Traceability by Design Component

### 8.1.1 Redundancy

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7.1.1</td>
<td>3.4.2</td>
</tr>
<tr>
<td>2.7.1.1.1</td>
<td>3.4.2.8</td>
</tr>
<tr>
<td>2.7.1.1.2</td>
<td>3.4.2.3</td>
</tr>
<tr>
<td>2.7.1.1.3</td>
<td>3.4.2.3</td>
</tr>
<tr>
<td>2.7.1.2</td>
<td>3.4.4.2</td>
</tr>
<tr>
<td>2.7.1.2.1</td>
<td>3.4.4.2</td>
</tr>
<tr>
<td>2.7.1.2.2</td>
<td>3.4.1.19</td>
</tr>
<tr>
<td>2.7.1.2.3</td>
<td>3.4.2.8</td>
</tr>
<tr>
<td>2.7.1.3</td>
<td>3.4.2.7</td>
</tr>
<tr>
<td>2.7.1.3.1</td>
<td>3.4.1.18</td>
</tr>
<tr>
<td>2.7.1.3.2</td>
<td>3.4.1.18</td>
</tr>
<tr>
<td>2.7.1.3.3</td>
<td>3.4.1.19</td>
</tr>
<tr>
<td>2.7.1.3.4</td>
<td>3.4.2.8</td>
</tr>
</tbody>
</table>

### 8.1.2 Encryption

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.2.1</td>
<td>4.3.5.1</td>
</tr>
<tr>
<td>2.4.2.1.1</td>
<td>4.3.5.1</td>
</tr>
<tr>
<td>2.4.2.1.2</td>
<td>4.3.5.1</td>
</tr>
<tr>
<td>2.4.2.2</td>
<td>4.3.2</td>
</tr>
<tr>
<td>2.4.2.3</td>
<td>4.3.2</td>
</tr>
<tr>
<td>2.4.2.4</td>
<td>4</td>
</tr>
</tbody>
</table>
2.4.2.5  4 and 5
2.4.2.6  4.3.4.1,2,3
2.4.2.7  4.3.2
2.4.2.7.1  4.3.2
2.4.2.7.2  4.3.1

8.1.3 Compression

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.2.1</td>
<td>5.4.1</td>
</tr>
<tr>
<td>2.5.2.1.1</td>
<td>5.4</td>
</tr>
<tr>
<td>2.5.2.1.2</td>
<td>5.3.2</td>
</tr>
<tr>
<td>2.5.2.2</td>
<td>5.4.1</td>
</tr>
<tr>
<td>2.5.2.3</td>
<td>5.4.1</td>
</tr>
<tr>
<td>2.5.2.4</td>
<td>5.4.2</td>
</tr>
<tr>
<td>2.5.2.4.1</td>
<td>5.4</td>
</tr>
<tr>
<td>2.5.2.4.2</td>
<td>5.4</td>
</tr>
<tr>
<td>2.5.2.4.3</td>
<td>5.4.2</td>
</tr>
<tr>
<td>2.5.2.5</td>
<td>5</td>
</tr>
<tr>
<td>2.5.2.6</td>
<td>5 and 4</td>
</tr>
</tbody>
</table>