# NFS File Handle Security

Avishay Traeger, Abhishek Rai, Charles P. Wright, and Erez Zadok
*Stony Brook University*

**Technical Report FSL-04-03**

## Abstract

Each file on an NFS server is uniquely identified by a persistent file handle that is used whenever a client performs any NFS operation. NFS file handles reveal significant amounts of information about the server. If attackers can sniff the file handle, then they may be able to obtain useful information. For example, the encoding used by a file handle indicates which operating system the server is running. The fields of the file handle contain information such as the date that the file system was created—often the same time that the OS was installed. Since an NFS file handle contains relatively little random data, it is not difficult to guess. If attackers can guess a file handle, then they can bypass the normal mounting procedures. This allows an attacker to access data without appropriate accounting and logging.

We have analyzed file handles on three common server operating systems: Linux, FreeBSD, and Solaris. Each one of them suffers from deficiencies when constructing file handles. We have modified the NFS server on Linux to use only randomly-generated file handles over the network. This makes it more difficult for an attacker to guess a file handle, or from utilizing information contained within a file handle. To persistently store file handles we use an in-kernel port of Berkeley DB. Our performance evaluation shows an acceptable overhead.

## 1 Introduction

NFS servers uniquely identify each file using a *file handle*. When an NFS client performs an operation, it passes the file handle to the server, which decodes the file handle to determine what object the file handle refers to. Since NFS is a stateless protocol, a client can present a file handle to the server at any point in the future. These file handles must be persistent, so they are typically in an easy-to-decode format. For example, a typical file handle contains a file system ID, a device number, an inode number, and a few other house-keeping fields. This leads to two problems. First, if attackers have access to NFS file handles, then they can learn information about the NFS server. Second, because there is little entropy within the file handle, an attacker can simply guess a file handle using brute force. For example, a typical Linux file handle takes about four days to guess on a 1Gbps network. Once attackers guess a file handle, they can bypass the normal mount procedures by communicating directly with the NFS daemon rather than initiating dialog with the mount daemon.

This work was done using NFSv3 in anticipation of applying the lessons to NFSv4, which is currently experimental and will not be widely used for some time. NFSv4 was designed to work over WANs, where packets may traverse many undfriendly networks and attackers could sniff packets without having local access to the server's network. In this case, the attacker cannot execute an attack that would require a login, but may be able to find another avenue of attack using information found in the file handles. In addition, although other attacks are possible, we are tackling some of them. We believe that any hole one can close is an improvement to security, however incremental. It should be noted that newer NFS servers are generally patched so that clients are authenticated with every RPC. Because of this, file handle guessing may have to be combined with IP spoofing for well-configured servers. Most servers though, have wildcards in their export lists (such as `*.example.com`) and so this is not as big of a problem.

We have analyzed the file handles on three common operating systems: Linux, FreeBSD, and Solaris. Each one of them suffers from deficiencies in the construction of the NFS file handles that may be exploited by an attacker.

When attackers have access to a file handle, they can glean useful information from it. In this scenario, the attacker has access to one or more file handles, most likely obtained by sniffing the network. Although it is the case that an attacker in this position can read files that are transmitted across the network, this information aids the attacker in gaining full access to all files. The operating system that the NFS server is running on is identifyable by the format of the file handle alone, which already narrows down the avenues of attack.

Fields within the file handle also contain information.

Some file handles include a file system identifier that is derived from the time that the file system was created. If the file system was created during the OS installation, the attacker can infer which patches are likely to have been installed because many system administrators install patches only when first installing the OS.

Of course `nmap` [6] can be used to guess some of this information, but it depends on some services to be available for probing. A conscientious system administrator would restrict this access, but the information is readily available in NFS file handles.

If attackers do not want an NFS mount request to go through the mount daemon, `mountd`, then they can guess a valid NFS file handle. This is done by crafting an RPC for a GETATTR operation, sending it directly to the NFS daemon, and using a positive reply to verify the guess. Because NFS file handles have been constructed with ease of decoding in mind, they contain little random data. For example, to guess the root file handle on Linux 2.4, only the device major and minor numbers need to be guessed. If the device is on a SCSI or IDE disk, then there are only six likely major numbers, and 256 minor numbers for each major number. An attacker can easily crack this file handle using brute force.

We have modified the Linux 2.4.22 NFSv3 server to use randomly-generated file handles over the wire. The changes were made at the entry and exit points of the procedures, leaving the NFS server internals unchanged. In addition, our enhancement requires no change to the protocol or client code. By using a random file handle, we prevent an attacker from utilizing information in the file handle and make it significantly more difficult for an attacker to guess. Because file handles need to be persistent, we store the mapping from our randomly-generated file handles to internal file handles using an in-kernel port of Berkeley DB. Berkeley DB allows us to efficiently store and retrieve keys and values in this mapping. We have evaluated the performance of our system, and have shown it to have a reasonable overhead under normal user access patterns.

The rest of this paper is organized as follows. Section 2 describes the current state of NFS security. Section 3 analyzes the security of NFS file handles on three operating systems. Section 4 describes our design for an improved NFS server. Section 5 evaluates the performance of our system. We conclude in Section 6.

# 2 Background

Although security features of the NFS protocol have improved with each subsequent version, many flaws still remain. We give an overview of NFS in Section 2.1. We describe the security of NFSv2 and NFSv3 in Section 2.2 and the security of NFSv4 in Section 2.3. In Section 2.4, we present examples of statefulness in NFS and NFS-like servers in order to show how this quality provides improved security and performance.

## 2.1 NFS Overview

The Network File System (NFS) [10] was developed by Sun Mircrosystems around 1984. It is a client/server model which implement a protocol that transparently provides clients remote access to shared files which reside on a server. Using NFS, the client can use the files as if they were on a locally mounted file system. Files are identified using file handles, which contain all the necessary information for the server to identify a file. The file handle is not interpreted by the client. An export point is a directory which is the root of a subtree that is to be exported. The server is configured with a list of which clients can access a given export point. The server then exports the directories, making them available to clients. A client that wishes to mount one of these directories first contacts the mount daemon on the server (`mountd`), and if it is authorized, receives a root file handle. This file handle can now be presented to the NFS daemon (`nfsd`) to access objects in the export point. From here, the client can perform LOOKUPs to obtain file handles for other exported objects.

## 2.2 NFSv2 and NFSv3

NFSv2 implemented minimal security, using only a client-supplied UID and GID for authentication (AUTH_UNIX) and only supporting UDP [16]. Since the authentication information is supplied by the client, it is easy to spoof. NFSv2 used host-based authentication for mounting, by checking the client's IP address against the export list.

When transitioning from NFSv2 to NFSv3 there were several security enhancements. Perhaps the most important addition to NFSv3 security was support for Kerberos (AUTH_KERB) [2]. Although this greatly improved security, it has not yet been implemented in the Linux and FreeBSD sources that we studied (versions 2.4.22 and 5.2, respectively). Additionally, since these operating systems did not implement DES authentication (AUTH_DES), the only method to authenticate users is AUTH_UNIX.

Another notable improvement in NFSv3 was support for TCP. This allows the server to use a more reliable connection, as well as set up a trusted route using a firewall or VPN. It is also more difficult for an attacker to inject packets when using TCP.

NFSv3 began supporting 64-byte file handles primarily to make it more difficult for an attacker to guess a file handle [19]. However, all of the operating systems that we studied did not use more than 32 bytes, and in some cases file handles were effectively as short as 12 bytes.

A major weakness of NFSv3 is that it does not pro-

vide strong authentication. The strongest authentication method that is widely implemented is AUTH_UNIX, which could be easily spoofed. Additionally, NFSv3 only uses host-based access control to provide authentication. Even though most servers now authenticate the client's IP address for every RPC, they are still susceptible to exploitation using guessed file handles. This can be accomplished by using IP spoofing or if the attacker is on the export list (usually because a wildcard was used).

NFSv3 only uses AUTH_UNIX or AUTH_NONE (no authentication) to authenticate a user while mounting. An attacker can exploit this weakness in two ways: by accessing some file or file system characteristics, and by intercepting a mount request and having the server use a weaker authentication method than the client requested [3].

## 2.3 NFSv4

The most notable security enhancement in NFSv4 is the RPCSEC_GSS security mechanism [4]. It works at the RPC level, so it can support both UDP and TCP, and can be implemented for older versions of NFS. NFSv4 specifies mandatory support for RPCSEC_GSS, which provides authentication, integrity, and privacy. It can provide support for private and public keys, data encryption, and strong authentication. It can support several security mechanisms, including Kerberos 5. In NFSv4, *principal-based* authentication is used rather than host-based. In host-based authentication, the client's IP address is used for authentication, whereas in principal-based authentication, a single user securely authenticates to the NFS server [13].

RPCSEC_GSS with server-side principal-based authentication provides a secure infrastructure that, if used properly, could make it difficult to gain unauthorized access using guessed file handles. NFSv4 champions this cause by making it mandatory for implementations to support a security model that uses end-to-end authentication, where an end-user on a client mutually authenticates to a principal on the NFS server. However, NFSv4 does have a provision for not using secure authentication. So for example, if an administrator turns off secure authentication in NFSv4 for reasons of speed, an attacker may be able to gain unauthorized access using a guessed file handle just as it would be done in previous NFS versions.

Rather than having several different daemons as in previous versions, the NFSv4 server is a single unified entity. Therefore, clients authenticate directly to the server instead of authenticating to a separate daemon at mount time. This improves security because now one daemon takes care of all operations. This also has the added benefit of being able to use one well-known port, so that setting up a firewall, a VPN, or a NAT is easier.

NFSv4 also supports volatile as well as persistent file handles. Volatile file handles can be set to expire at a certain time, for example, during file system migration or during file system renaming. This is useful for managing the server, as well as increasing security. If file handles only last for a certain amount of time, they are only valuable for a small window of time. This has the drawback of increased overhead due to expired file handles, which must be renewed.

The two problems addressed in this paper, file handle guessability and information leakage, do not appear to be solved in NFSv4. Although not all implementations are complete, the published material about the new protocol does not address either issue.

The NFSv4 protocol specification, as in the previous NFS versions, does not specify how the file handle should be constructed. The only requirement is that file handles should uniquely identify server objects, and that it is the server's responsibility to maintain this relationship. Since the specification does not directly provide for file handle security, NFSv4 file handles may not be more secure than before. In fact, the experimental implementation of NFSv4 in the Linux 2.6.0 kernel contains the same information as in NFSv3.

## 2.4 A Stateful Server

Using a database to map normal file handles to random file handles adds some state to NFS, but it does not break the protocol. The drawbacks from this added state are minimal, since the database handles crash recovery. Moreover, this is not the first time that state has been added to NFS.

Developers strived to make NFSv2 and NFSv3 stateless in order to provide better recoverability in case the server went down; once the server came back up, it did not have to reconstruct the state it was in before it crashed. In particular, clients could resume using a failed server that came back up without incurring failures. Although this was a good design goal, the statelessness that was present led to added complexity and problems with the code. The technique used by NFS to remove state was to create new stateful protocols so that the actual NFS daemon did not have state. One example of moving the state to a new protocol is file locking. File locking requires the NFS server to maintain state information about locks. To solve this problem, Sun added another RPC protocol and daemon (LOCKD) for lock operations. In case the server that was running this daemon went down, Sun added yet another RPC protocol to handle the recovery of locking information (STATD). This added complexity causes several problems. For example, if the client system goes down, the server is not notified about the lock being released until the client remounts the server again (there is no timeout) [8]. Even

after all of this protocol rearrangement, the NFS daemon still has state. It keeps an RPC reply cache in memory to handle non-idempotent operations: if a client sends a duplicate request, the operation would not be repeated and the previous server reply is resent. Even so, if the server crashed after completing an operation and before sending a reply to the client, the operation is repeated when the server is brought back up, and an error is returned for the duplicate operation.

The Andrew File System (AFS) [7], developed by Carnegie Mellon University and IBM decided to implement a stateful server in order to improve performance, a quality that NFS was lacking at the time.

Not Quite NFS (NQNFS) attempted to improve NFS performance at the expense of statelessness. Since the statelessness in NFS requires RPCs to be entirely synchronous, the client experiences much delay while waiting for RPC replies. The problem was addressed by adding some state and changing the protocol to implement client-side caching [9].

NFSv4 dropped the notion of statelessness, and all related protocols were merged into a single NFSv4 protocol. This simplified the interaction between clients and servers by using a single port to contact the server, improved security since clients interact with one entity, and improved performance by allowing client-side caching and compound operations (the client could now send more than one operation in a single RPC).

# 3 File Handle Security Analysis

NFS file handles contain system information that may be viewable by an attacker. The threat model assumes that the attacker has access to file handles, usually by sniffing the network. Most servers do not encrypt file handles (mainly because not all implementations of NFSv2 and NFSv3 support it), and so the system information contained in the handles is in plain view. It is also assumed that the server is not using strong authentication. This is reasonable, since weak authentication (AUTh_UNIX) is predominant.

File handles are not very difficult to guess. Since log files generally only record mount operations, if attackers bypass the mount daemon with a guessed file handle, then their actions are likely to go unnoticed. The threat model for this scenario assumes that the attacker can authenticate to the server, but does not have access to file handles. Authentication can be achieved if the attacker is on the export list (it is not uncommon for system administrators to use wildcards in their export lists), or can appear to be on the export list using IP spoofing.

In this section we will explore the file handle's contents for the servers of three major operating systems. The purpose of this is twofold. First, by inspecting the kernel source code and network traffic, we analyzed the type of information that is leaked in a file handle. Second, we estimated how much effort would be required to guess a file handle. The Linux file handle is discussed in Section 3.1, the FreeBSD file handle in Section 3.2, and the Solaris file handle in Section 3.3. In Section 3.4 we discuss the time estimates for guessing the file handles.

## 3.1 The Linux File Handle

Linux uses the same file handle format for NFSv2 and NFSv3, though it had an older file handle format that was used in NFSv2 and became obsolete in Linux 2.4.0. The current file handle is outlined in Table 1. In practice, the amount of data in the NFSv2 and NFSv3 file handle varies from six fields occupying 12 bytes to nine fields occupying 24 bytes. In NFSv2, 32-byte file handles are required, so the file handle is padded with zeros. NFSv3 allows variable length file handles so no padding is required. Since Ext2 and Ext3 are the most commonly used file systems in Linux, we concentrate on them when describing file system dependent fields. The following is an explanation of fields found in the Linux file handle:

**`fb_version`** is the file handle format's version number, which is always 1.

**`fb_auth_type`** is the authentication method, which is currently 0, but may take on other values in the future.

**`fb_fsid_type`** is the method for encoding the fsid of the export point. This field is currently 0, but may take on other values in the future.

**`fb_fileid_type`** is the method for encoding the file information. This byte determines how bytes 13–24 are set. If this field is set to 0, then this is the root file handle, and bytes 13–24 are left empty. The `fh_fileid_type` field is set to 1 if the NFS server is accessing files or directories, or performing a lookup on a directory. In this case, bytes 13–20 are set, but bytes 21–24 are left empty. If a lookup is performed on a file, then this field is set to 2, and bytes 13–24 are set. This field does not take on any other values. The root file handle is the easiest to guess since it contains less information than non-root file handles.

**`xdev`** consists of the major number of the exported device in its first two bytes, and the minor number in its second two bytes. These are generally small, easy to guess numbers. Additionally, this field reveals the type of drive that the export point is on, along with some idea of how many other drives of the same type are on the system (based on the minor number).

**`xino`** is the inode number of the export point. The root of a file system is usually exported so the inode number of the export point is generally a small number (almost always 2).

| Length | Bytes | Field Name | Meaning | Typical Values |
|---|---|---|---|---|
| 1 | 1 | `fb_version` | NFS version | Always 1 |
| 1 | 2 | `fb_auth_type` | Authentication method | Always 0 |
| 1 | 3 | `fb_fsid_type` | File system ID encoding method | Always 0 |
| 1 | 4 | `fb_fileid_type` | File ID encoding method | Always either 0, 1, or 2 |
| 4 | 5–8 | `xdev` | Major/Minor number of exported device | Major number 3 (IDE), 8 (SCSI) |
| 4 | 9–12 | `xino` | Export inode number | Almost always 2 |
| 4 | 13–16 | `ino` | Inode number | 2 for `/`, 19 for `/home/foo` |
| 4 | 17–20 | `gen_no` | Generation number | 0xFF16DDF1, 0x3F6AE3C0 |
| 4 | 21–24 | `par_ino_no` | Parent's inode number | 2 for `/`, 19 for `/home` |
| 8 | 25–32 | Padding for NFSv2 | | Always 0 |
| 32 | 33–64 | Unused by Linux | | |

*Table 1: Summary of contents for the NFSv2 and NFSv3 Linux file handle.*

**`ino`** is the inode number of the file or directory. Ext2 divides the file system into block groups, and allocates inode numbers by assigning the first unused inode number in the file's block group. Therefore the inode numbers will generally increase sequentially from the starting points of the block groups. Ext3 uses a similar algorithm.

**`gen_no`** is the generation number of the file or directory. In Ext2, there is a variable called `event` that is initialized to 0 at boot time and is incremented by one in many places such that the value is hard to predict. For example, the value is increased every few seconds, when the mouse is moved, and in various inode operations. In Ext3, the generation number variable is set to a random number at mount time and is incremented each time an inode is allocated.

**`par_ino_no`** is the inode number of the parent directory.

Assuming that the inode number of the export point is 2, then the only field that needs to be guessed for a root file handle is `xdev`. Common device major numbers can be looked up in the Linux kernel source in `Documentation/devices.txt` and minor numbers are generally small, so this field can be guessed relatively easily. The root file handle is what an attacker is likely to try to guess, so the added entropy in bytes 13–24 will not deter an attacker.

## 3.2 The FreeBSD File Handle

The FreeBSD file handle outlined in Table 2 is used both for NFSv2 and for NFSv3. NFSv2 expects 32-byte file handles, and since the file handle only utilizes 20 bytes, it is padded with zeros. NFSv3 does not require this padding because it allows variable length file handles. Since UFS is the most commonly used file system in FreeBSD, we concentrate on it when describing file system dependent fields. The following is an explanation of the fields found in the FreeBSD file handle:

**`fsid`** contains the file system ID. In UFS, the first four bytes are the Unix time when the file system was created, and the next four bytes are obtained from the `arc4random` function. This is the FreeBSD implementation of the RC4 algorithm that contains fixes to some problems in the original algorithm [5]. The `fsid` is hidden from non-root users in the `statfs` system call, but is present in clear text in file handles. The first four bytes do not have to be guessed by brute force alone, since the Unix time when the file system ID was set is a fairly small window, probably within the past few years. Additionally, having access to this information can give an attacker some clues about the exact operating system that the server is running. The file handle format will give away that the server is running FreeBSD, and the Unix time will give some estimate about which version it is. Since it is common for system administrators to update security patches when setting up a system, it would be wise for an attacker to probe for weaknesses discovered only after this time. This is especially true since not all administrators patch their systems regularly.

**`fid_len`** is the length of the rest of the structure. This is always 12.

**`fid_reserved`** is padding for word alignment and is always 0.

**`ufid_ino`** is the inode number. UFS inode numbers are allotted in a monotonically increasing fashion, so files created early will have small inode numbers. These are often directories such as `/` and directories in the `/home` directory, which may be of particular interest to an attacker.

**`ufid_gen`** is the generation number of the file. UFS generation numbers are created using the `arc4random` function. If this inode was already assigned an inode number, then it is just incre-

| Length | Bytes | Field Name | Meaning | Typical Values |
|---|---|---|---|---|
| 8 | 1–8 | `fsid` | File system ID | 0x3F607F14 1C47F86E |
| 2 | 9–10 | `fid_len` | Length of the rest of the structure | Always 12 |
| 2 | 11–12 | `fid_reserved` | Word alignment | Always 0 |
| 4 | 13–16 | `ufid_ino` | Inode Number | 3, 47 |
| 4 | 17–20 | `ufid_gen` | Inode Generation Number | 0x0C8F960C, 0x9CFF85C7 |
| 12 | 21–32 | Padding for NFSv2 | | Always 0 |
| 32 | 33–64 | Unused by FreeBSD | | |

*Table 2: Summary of contents for the NFSv2 and NFSv3 FreeBSD file handle.*

mented by one.

Guessing a FreeBSD file handle is more difficult than Linux since eight bytes are random (four bytes of the `fsid` field and the `ufid_gen` field). Additionally, FreeBSD uses a reliable random function as opposed to the method of obtaining generation numbers in Ext2 found on Linux.

## 3.3 The Solaris File Handle

The Solaris file handle outlined in Table 3 is used in both NFSv2 and NFSv3. Since UFS is the most commonly used file systems in Solaris, we concentrate on it when describing file system dependent fields. The following is an explanation of the fields found in the Solaris file handle:

**`fh_fsid[0-3]`** is the file system ID. This is the device major number and minor number, compressed to fit into 4 bytes.

**`fh_fsid[4-7]`** is the file system type, which is always 2 for UFS.

**`fh_len`** is the size of the `fh_data` field, which is always 10.

**`fh_data[0-1]`** is padding for word alignment, which is always 0.

**`fh_data[2-5]`** is the inode number of the file. This is identical to the `ufid_ino` field in the FreeBSD inode. UFS allocates inode numbers in a monotonically increasing fashion; therefore, files created early, which may be of interest to an attacker, will have small inode numbers. This greatly reduces the search space for inode numbers.

**`fh_data[6-9]`** is the generation number. UFS generation numbers are created using the `fsirand` [14] function.

**`fh_xlen`** is the size of the `fh_xdata` field, which is always 10.

**`fh_xdata[0-1]`** is padding for word alignment which is always 0.

**`fh_xdata[2-5]`** The inode number of the export point. For root file handles, the information contained in this field is identical to the information in `fh_data[2-5]`.

**`fh_xdata[6-9]`** The generation number of the export point. In root file handles, the information contained in this field is identical to the information in `fh_data[6-9]`.

In addition to the generation numbers being leaked, the attacker can identify the file system and device being exported from `fh_fsid`. As is the case for Linux, the root file handle in Solaris is the easiest to guess. This is because `fh_xdata` is not used.

## 3.4 Time Estimates

After analyzing the contents of each operating system's file handle, we estimated the time required to guess each one. For this test, we used two dual 900MHz Itanium computers with 8GB of RAM with two dedicated connections: 1Gbps and 100Mbps. One machine acted as the NFS server, and the other as the rogue host that was guessing the file handle. Table 4 shows the time required to guess a Linux file handle. For a frame of reference, the speeds for ping flooding are provided. Ping flooding sends packets as fast as they come back, or 100 packets/sec, whichever is greater. It should be noted that the network of a real-world server is a shared medium and therefore the times described will be vary depending on other traffic.

| | 1Gbps | 100Mbps |
|---|---|---|
| Guess attempts/sec | 13,119 | 6,996 |
| Guess Mbps sent | 17.0 | 8.8 |
| Guess Mbps received | 7.8 | 3.9 |
| Ping Mbps sent | 23.0 | 20.0 |
| Ping Mbps received | 23.0 | 14.0 |

*Table 4: Speeds for file handle guessing compared to speeds for ping flooding.*

The file handles for the three operating systems are roughly equivalent in size, so we can use the attempts/sec metric as an estimate for all three. Since the 1Gbps connection was roughly twice as fast as the 100Mbps, we will only discuss time estimates for the 1Gbps connection, and the 100Mbps can be obtained by doubling it.

The Linux root file handle, where only the major and minor numbers have to be guessed, can be guessed in under one second. The non-root file handle has an in-

| Length | Bytes | Field Name | Meaning | Typical Values |
|---|---|---|---|---|
| 4 | 1–4 | `fh_fsid[0-3]` | Derived from device major/minor numbers. | 0x01980000 |
| 4 | 5–8 | `fh_fsid[4-7]` | File system type | 2 for UFS |
| 2 | 9–10 | `fh_len` | size of `fh_data` | Always 10 |
| 2 | 11–12 | `fh_data[0-1]` | Make the inode number word aligned | Always 0 |
| 4 | 13–16 | `fh_data[2-5]` | Inode number | 4 for `/`, 37 for `/home/foo` |
| 4 | 17–20 | `fh_data[6-9]` | Generation number | 0x72D2970C, 0x2482AAF4 |
| 2 | 21–22 | `fh_xlen` | size of `fh_xdata` | Always 10 |
| 2 | 23–24 | `fh_xdata[0-1]` | Make the inode number word aligned | Always 0 |
| 4 | 25–28 | `fh_xdata[2-5]` | Inode number of the export point | 4 for `/` |
| 4 | 29–32 | `fh_xdata[6-9]` | Generation number of the export point | 0x72D2970C |
| 32 | 33–64 | Unused by Solaris | | |

*Table 3: Summary of contents for the NFSv2 and NFSv3 Solaris file handle.*

ode number which is generally small, a four byte generation number which must be obtained by brute force, and sometimes the parent inode number which is also generally small. This time required to guess four random bytes is about 3.5 days. To obtain the amount of time required to guess a file handle, this time must be multiplied by a constant to account for a small number of guesses to guess the device numbers and inode number (these can also be obtained with non-privileged commands like ls and mount if the attacker has an account on the server). Additionally, we don't need to guess a specific file handle, since once we have one we can obtain the rest through lookups. Therefore we should divide the time by the number of objects present on the file system.

The FreeBSD file handle is more difficult to guess because it contains eight random bytes: four bytes for the generation number, and four bytes in the fsid. It also contains the Unix time when the file system was created, which can be estimated, and an inode number which is generally low. This makes the FreeBSD file handle the most secure of the three file systems, since guessing the eight random bytes will take many years.

The Solaris root file handle contains only four random bytes for the generation number. The fsid and inode number can be easily guessed. Guessing this file handle would take about a week on the 1Gbps network and two weeks on the 100Mbps network. This is about as secure as the non-root file handle on Linux. Guessing the non-root file handle is more difficult, since now there are eight random bytes, making it as time-consuming as guessing the FreeBSD file handle.

Attempting to guess a file handle at full speed will probably flood the network, which will likely attract the attention of a system administrator. However, since the average file system lasts for years, the attacker can use a small fraction of the bandwidth for a longer amount of time and have a good chance of not being noticed. For example, if an attacker uses 10% of the bandwidth, the required time for a non-root Linux file handle will be 35 days, which is still very acceptable. Even if 1% is used, it will still be guessed in under a year. The information on the server is likely to persist for longer than this time.

## 4 Design

We had the following four goals while designing our system:

**File Handle Security** Our primary design goal was to prevent information from leaking through the NFS file handle and making the file handles more difficult to guess. To achieve this goal we use only random file handles on the wire. This was accomplished by obtaining random bytes from `/dev/urandom` [17]. This is a character special file in Linux which gathers environmental noise from sources such as device drivers to make a strong pseudo-random number generator.

**Compatibility** Since updating code on all clients is not practical, the implementation should only modify the NFS server and not change the NFS protocol or client code. The file handles should be persistent and the same design should work on all versions of NFS.

**Performance and Efficiency** If system administrators are to be expected to secure their systems with this server enhancement, it needs to run with minimal overhead and not consume too much memory. In our design, we addressed these issues by using only kernel-level code and an efficient in-kernel database.

**Simplicity** We kept our design simple so it would not introduce new security flaws or adverse side effects in the kernel. Most importantly, we strived to change the NFS daemon module code as little as possible.

In Section 4.1 we discuss how we used databases to carry out these goals; in Section 4.2 we describe the

changes made to the NFS daemon code; and in Section 4.3 we describe the database that was ported.

## 4.1 Database Schema

In order to realize the central goal of improving file handle security, we chose to create a mapping between normal NFS file handles and file handles consisting entirely of random bytes. Since this mapping must be persistent, we used a Linux kernel port of Berkeley DB (BDB) [12]. This database was previously ported by our research group when we saw that it could be useful in various parts of the kernel. Some uses include implementing extended attributes or ACLs in the VFS to enhance file systems that do not natively support them, per-page virus scanning states, per-file cipher keys and checksums, and implementing the `/proc` file system using the database so that it is persistent and easy to use.

The server must be able to translate one file handle type to the other. On the entry point to a function, the NFS server converts the random file handle that it receives to a normal NFS file handle. There are also cases where the server needs to convert a normal file handle to a random file handle. This is done most often to check for existing mappings. One of the drawbacks of BDB is that it does not adequately support bidirectional mappings, so we decided to use two databases, one to map random file handles to normal ones (`regular.db`), and one for the reverse mapping (`random.db`). To increase efficiency, we used hash tables as the indexing method. Additionally, since all systems studied have constant parts (see file handle tables in Section 3), it would have been possible to store only the file information in the databases and rebuild the file handles before use. We chose not to do this because it would add more complexity to the code and would have to be changed for different servers.

A system administrator can change and query the number of random bytes used for the random file handles through the `/proc` file system. Using more random bytes is beneficial because there are fewer conflicts in the databases and the random file handles are more difficult to guess. There are three drawbacks to using larger file handles: there is more overhead since more data has to be transmitted between the client and server, random byte generation takes more time, and more kernel memory is consumed.

## 4.2 NFS Server Changes

Next, we discuss the necessary changes we made to the NFS server code for the various types of procedures.

**Exporting** If the databases do not already exist, they are created on a separate drive when the server exports a mount point. The databases should be on a separate, unexported drive to ensure that they are not accessible from the outside. The databases are now ready to be queried and updated.

**Mounting** `random.db` is queried to check if this mount point is already mapped to a random file handle. This would occur if this directory was previously mounted. In this case, the random file handle associated with the mount point is returned to the user. If this is the first time that a client is mounting this directory, a new random file handle is generated, the regular and random file handles are stored in both databases, and the random one is sent back to the client.

**General NFS Operations** The input to all operations is a file handle to operate on, but most do not return a file handle to the user. These procedures are GETATTR, SETATTR, ACCESS, READLINK, READ, WRITE, REMOVE, RMDIR, RENAME, LINK, READDIR, FSSTAT, FSINFO, PATHCONF, and COMMIT. In these cases, the random file handle that the server receives is translated to a regular file handle, and execution proceeds normally.

**Create Operations** The CREATE, MKDIR, SYMLINK, and MKNOD operations receive a file handle of the directory where the new file is being created and the name of the new file, and send a new file handle back to the client. The execution of these operations is shown in Figure 1. These procedures begin their execution in the same manner as the general operations described above. However, after the NFS procedure executes, a new file handle must be sent back. We first check if this file handle is already in `random.db`. If it is, we send back its corresponding random file handle. Otherwise we create a new random file handle, store both file handles in the databases, and return the random file handle. We also ensure that the random file handle chosen has not already been used.

**Unlink Operations** The REMOVE and RMDIR operations receive a file handle as a parameter, but do not return one back to the client. For these operations, the database entries are removed if the object is removed (i.e., the reference count reaches zero).

**LOOKUP** The LOOKUP operation usually behaves like a general NFS operation, but if the file being looked up has not been seen before (for example, on a legacy file system), the file handles are stored in the databases as in the create operations.

**READDIRPLUS** The READDIRPLUS operation receives a file handle for a directory, and returns a list of entries in the directory along with their attributes. The incoming file handle is first translated to a normal file handle as in the general NFS operations. The file handle for each file in the directory is converted to a random file handle before it is returned to the client.

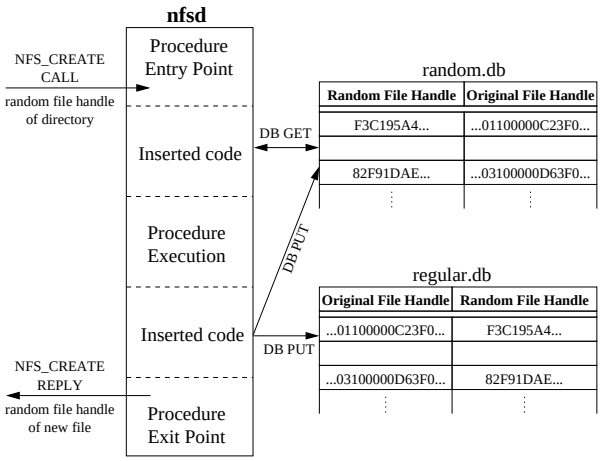


*Figure 1: A create operation*

## 4.3 Berkeley DB

Having the database reside in the kernel improves both speed and security. Berkeley DB (BDB) is a widely-used and reliable product. It provides the ability to efficiently and persistently store key-value pairs using hash tables, B+-trees, queues, and logical record number indexing. BDB is a stable, fully featured, and widely-used embedded database with support for transactions and replication. BDB can store data both on disk and in memory. An in-memory database is useful because kernel code can take advantage of a complex and efficient data structure that supports concurrent access. If an on-disk database is used, then a backing store is provided to the application with only small development costs thanks to the high-level BDB API. BDB also fully supports concurrency and recoverability. Since BDB takes care of concurrency, there was no need to be concerned with cases such as `mountd` and `nfsd` accessing the databases simultaneously. Also, since BDB handles recoverability, the client will not incur additional stale file handles in the event of a server crash.

## 4.4 Porting Berkeley DB

We decided to port the Berkeley DB (BDB) package into the Linux kernel because there is a definite need to efficiently store data inside the Linux kernel without having to reimplement complex data structures. As mentioned previously, our research group ported the database after finding many uses for it in the kernel.

Figure 2 shows the components of the BDB source that were identified and ported to the kernel to build a fully-functional in-kernel database that has support for BDB's core functionality. In the first prototype, replication, RPC, cryptography, and other useful but non-essential features were not included. The components have been grouped by their functionality. The arrows indicate the components' calling conventions.

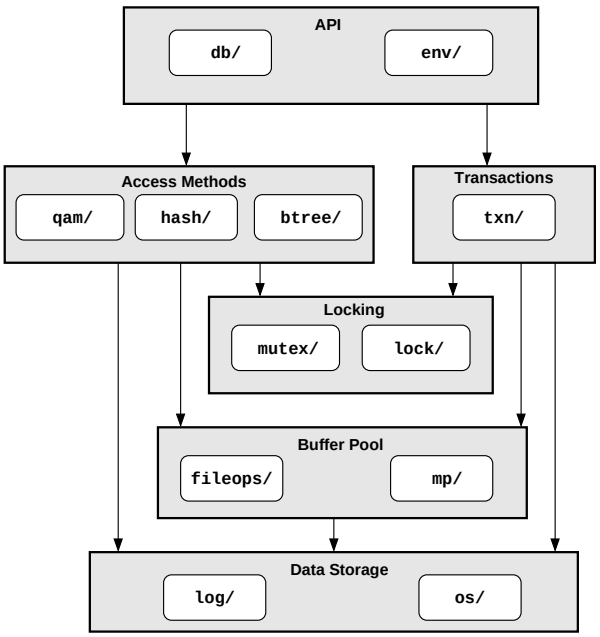


*Figure 2: In-Kernel Berkeley DB Architecture*

The application accesses the database methods using the APIs provided by the `db/` and `env/` directories. The APIs use one of the two access methods we have ported which are in the `btree/` and `hash/` directories. These directories provide algorithms to efficiently store and retrieve data.

The transaction component provided by the `txn/` directory allows a set of operations to be applied atomically. There is complete support for roll-back in all transactions. Critical sections that require atomic operations need to make use of the lock component in the `lock/` directory.

During initialization, Berkeley DB allocates a large buffer pool for storing in-kernel memory images of the database. All transactions and queries access the memory provided by the `fileops/` and `mp/` directories of the buffer pool component.

The in-kernel memory image is copied onto a physical storage device periodically using the data storage methods in the `os/` directory. All components record database operations in log files using the methods in the `log/` directory. The data storage component is responsible for translating database operations like `put` into file system operations like `write`.

The file `sync` method commits the data to the backing store.

# 5 Evaluation

We developed a prototype for the NFSv3 server enhancement on Linux 2.4.24. We added 513 lines of code to the in-kernel NFS daemon (4 lines to `export.c`, 49 lines to `nfs3proc.c`, 3 lines to `nfs3xdr.c`, 64

lines to `nfsctl.c`, and 393 lines in our own files). We tested the enhancement using 4, 32, and 64-byte random file handles to represent the full range of possibilities. We compare these results with those obtained from the vanilla NFS server.

Our NFS client machine was a 1.7 GHz Pentium 4 with 1 GB of RAM running Red Hat 9 with a vanilla 2.4.24 kernel. The server machine was a 500MHz Pentium 3 with 128MB of RAM running Red Hat 9 with a vanilla 2.4.24 kernel (using the KBDB module and our modified `nfsd` module). We chose to use the faster machine as the client to ensure that the machine issuing the RPCs would not be the bottleneck. The mount point on the server was on an 18.3GB 15,000 RPM Maxtor Atlas SCSI disk formatted with Ext2. To isolate performance characteristics, the databases were written to a different identical disk. The two machines were directly connected using a dedicated gigabit Ethernet link.

We ran all tests at least 10 times and computed 95% confidence intervals for the means using the student-$t$ distribution. In each case, the half-width of the interval was less than 5% of the mean.

In Section 5.1 we describe the configurations we tested; in Section 5.2 we describe the SPEC SFS benchmark; and in Section 5.3 we describe the Am-Utils benchmark.

## 5.1 Configurations

We used the following seven configurations to evaluate our NFS server enhancement:

**VAN** A vanilla NFS server running on the 2.4.24 kernel. It serves as a baseline for performance of other configurations.

**ENH-4** Our enhanced NFS server using 4 byte random file handles. This is the smallest file handle that can be used.

**ENH-32** Our enhanced NFS server using 32 byte random file handles. This is a commonly used file handle size and also the median size file handle that can be used.

**ENH-64** Our enhanced NFS server using 64 byte random file handles. This is the largest file handle size that the NFSv3 protocol allows.

**ENH-NODB** Our enhanced NFS server which generates 32-byte random numbers, but does not store them in the database or use them. This demonstrates the overhead of generating random numbers without using the database.

**ENH-NORAND** Our enhanced NFS server that does not generate random numbers. The database mapping is the identity function, and all database operations are still performed. This demonstrates the overhead of using the database without generating random numbers.

**ENH-NORDP** Similar to ENH-NORAND except that it does not perform database operations for the READDIRPLUS RPC. This was tested because READDIRPLUS is an expensive and seldom used operation.

## 5.2 SPEC SFS

One of the tests we used to evaluate the performance of our enhancements to the NFS server was the SPEC SFS 3.0 benchmark [11, 15], which is the official benchmark for measuring NFS server throughput and response time. SFS's predecessors invoked system calls and therefore results were inaccurate because they were dependent on the client's implementation. SFS 1.0 addressed this issue by crafting its own RPCs. Given a requested load, SFS generates an increasing load of operations and measures the response time until the server is saturated. This is the maximum sustained load that the server can handle under this requested load. As the requested load increases, response time diminishes.

SFS 1.0 had four major deficiencies:

- It used the LADDIS [18] workload which contained a mix of operations obtained from an inaccurate study.
- All files created were 136KB, which is unrepresentative for modern workloads.
- It was only able to measure the throughput of the server, not latency.
- It still had some dependencies on the NFS client and was not entirely portable.

SFS 2.0 fixed these shortcomings and added support for NFSv3 and TCP. SPEC SFS 3.0 updated some important algorithms such as time measurement, workload regulation, and file operations.

We ran SPEC SFS 3.0 with various requested loads (multiples of 50) generated by one process. It tested NFSv3 over UDP. All other parameters were left at the default settings. Results for sustained loads are shown in Figures 3 and 4. Note that in these two figures, both the X and Y axes do not begin at zero.

During this discussion, overheads will be given for peak loads, and the overhead for a requested load of 700 (where different configurations diverged the most) will be given in square brackets. When benchmarking VAN, the server was able to satisfy requested loads of up to 500 ops/sec. When the requested load was increased to 600 ops/sec, the actual performance began to degrade. ENH-4, ENH-32, and ENH-64 were able to satisfy requested loads of up to 450 ops/sec. The overheads for the three file handle sizes were 11.1% [23.3%], 11.3% [29.5%], and 13.3% [31.7%], respectively. The small decline in performance between different size file handles is due to added network I/O, increased time for looking up larger

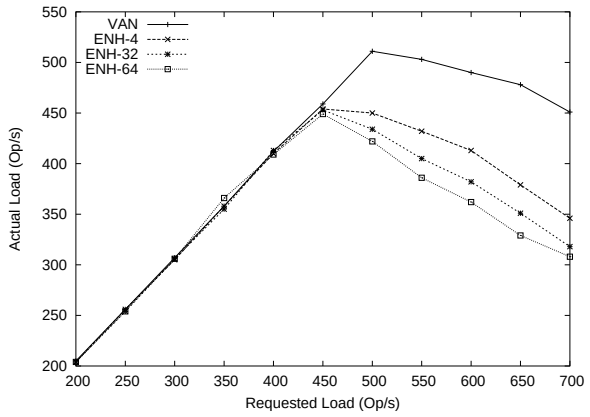


*Figure 3: SFS-3.0 operations per second. Comparison between vanilla NFS server (*VAN*) and enhanced server using 4, 32, and 64-byte random file handles (*ENH-4, ENH-32, ENH-64*).*

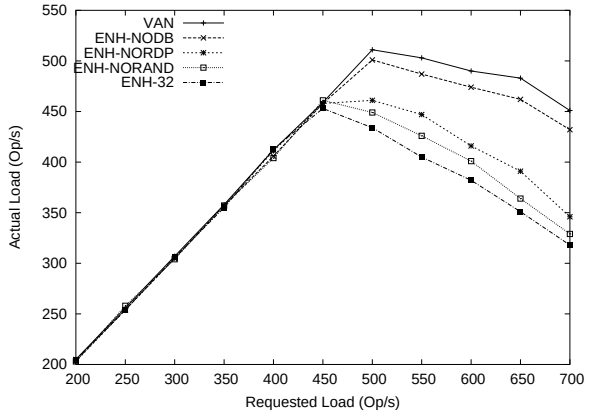


*Figure 4: SFS-3.0 operations per second. Comparison between vanilla NFS server (*VAN*), the enhanced server without the database (*ENH-NODB*), the enhanced server without the* READDIRPLUS *operation (*ENH-NORDP*) the enhanced server without the random byte generation (*ENH-NORAND*), and the enhanced server using 32-byte file handles (*ENH-32*).*

keys in the database, and increased time in obtaining random numbers.

To target the cause of the performance degradation for the enhanced server, we isolated two costly aspects: database operations (ENH-NORAND) and random number generation (ENH-NODB). ENH-NODB performed similarly to VAN with only a 1.2% [4.2%] overhead, whereas ENH-NORAND was close to ENH-4 with a 9.8% [27.0%] overhead. It is clear that the database is the main reason for the performance degradation of the enhanced server. This is because each NFS RPC requires at least one database operation, which is costly.

Table 5 shows the breakdown for NFS operations in the SFS 3.0 benchmarks for NFSv3. It should be noted that READDIRPLUS accounts for 9% of the operations. This is a particularly expensive operation for our server enhancement, because it requires a database operation for every file in the directory. To determine how much this operation affected the results of the enhanced server, we tested the ENH-NORDP configuration. It satisfied requested loads up to 450 ops/sec, and had an overhead of 9.8% [17.9%] load. The ENH-NORDP configuration is 11.1% slower than ENH-NORAND at a requested load of 700 ops/sec (they have the same peak).

There are still problems with SFS, which we came across when using the benchmark. The first issue is that the clients bombard the server with RPCs, which is fine for measuring peak sustained throughput, but does not simulate a normal workload. It should therefore be used together with another benchmark that includes CPU activity and is not so I/O intensive, as we did with Am-Utils (Section 5.3). The second issue is that since the workload was obtained from Sun Microsystems' network, the operations mix is only truly characteristic of Solaris clients and servers in 1997 when SFS was standardized. After observing the network traffic for the three operating systems discussed in this paper, we have determined that Linux and FreeBSD do not use READDIRPLUS. This implies that our enhancement would actually perform close to ENH-NORDP on these systems in practice. Instead of issuing this RPC, Linux and FreeBSD clients do a LOOKUP and GETATTRs for each object in the directory. Since SPEC SFS already has different source code for each operating system, it should not be difficult to use a different mix of operations for each client while keeping the results comparable (for example, Linux clients should use READDIR and GETATTRs, while Solaris will use READDIRPLUS).

| NFS Operation | SFS 3.0 NFSv3 |
|---|---|
| LOOKUP | 27% |
| READ | 18% |
| WRITE | 9% |
| GETATTR | 11% |
| READLINK | 7% |
| READDIR | 2% |
| CREATE | 1% |
| REMOVE | 1% |
| FSSTAT | 1% |
| SETATTR | 1% |
| READDIRPLUS | 9% |
| ACCESS | 7% |
| COMMIT | 5% |

*Table 5: Percentage of NFS operations for the SFS 3.0 Benchmark using NFSv3.*

## 5.3 Am-Utils

We also tested our NFS server enhancements using builds of Am-Utils version 6.1b3. This version of Am-

Utils contains 430 files and more than 60,000 lines of C code. The build process begins by running several hundred small configuration tests to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation: a total of 152 new files and 19 new directories.

Though the Am-Utils compile is CPU intensive, it contains a fair mix of file system operations. We used the Tracefs aggregate driver to measure the operation mix [1]. For this benchmark, 25% of the operations are writes, 22% are `lseek` operations, 20.5% are reads, 10% are `open` operations, 10% are `close` operations, and the remaining operations are a mix of other operations such as `readdir` and `lookup`. Am-Utils provides a more balanced mix of CPU and I/O operations compared to SPEC SFS 3.0, which exercises I/O quite heavily. Although the mix of NFS operations for the SFS workload is based on actual use, it does not take CPU-driven operations into account. This workload demonstrates the performance impact a user sees when using our NFS server enhancement under a normal workload.

| | VAN | ENH-32 |
|---|---|---|
| Elapsed time | 151.9s | 156.0s |
| System time | 37.7s | 37.6s |
| Wait time | 39.2s | 49.9s |

*Table 6: Am-Utils results. Comparison between vanilla NFS server (*VAN*) and enhanced server with 32-byte random file handles (*ENH-32*).*

We tested the performance of ENH-32 against VAN since 32 bytes is the most common file handle size. Table 6 shows the results. ENH-32 only had a 2.7% overhead for elapsed time and a 27.3% overhead for wait time. The system times were statistically indistinguishable. The wait time is primarily I/O with some time spent by the scheduler. In this case, I/O consists of client-side disk I/O (for binaries like the compiler), network I/O, and server response time (time spent by the client waiting for an RPC reply). The overhead for wait time is the cause of the elapsed time overhead. The elapsed time overhead is lower because of I/O and CPU interleaving, which is characteristic of actual usage.

In summary, we evaluated our NFS server enhancement using both SPEC SFS and Am-Utils. We showed that the security enhancement has an acceptable overhead over the vanilla NFS server.

# 6 Conclusions

Our work has two contributions. First, we have created a simple and effective solution to prevent information leakage in NFS file handles and make it considerably more difficult for attackers to guess file handles. We have also laid the groundwork for future improvements, not only to NFSv2 and NFSv3, but also to NFSv4. The enhancement also runs with acceptable overhead, making it practical.

Second, Berkeley DB is the first port of such an efficient and robust database to the Linux kernel. Not only was this useful for improving NFS security, but it can now also be used for other applications.

## 6.1 Future Work

In the future we plan on improving the performance by using encryption instead of the databases. In this case, only the encrypted file handles would be present on the wire.

This work will also extend to NFSv4. As mentioned earlier, this project was done with the goal of applying the lessons to NFSv4. Since the NFSv4 file handles in the Linux 2.6.0 kernel contain the same information as those for NFSv3, we can easily use the same technique to secure the new file handles. We will also use NFSv4's volatile file handles to implement short-lived file handles so that the window in which an attacker can use a stolen or guessed file handle is as small as possible. It will also be possible to assign a different file handle to every client, so that stronger authentication and more versatile access controls can be implemented.

# 7 Acknowledgments

This work was partially made possible by an NSF CAREER award EIA-0133589, NSF award CCR-0310493, and HP/Intel gifts numbers 87128 and 88415.1.

# References

[1] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 129–143, San Francisco, CA, March/April 2004.

[2] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. Technical Report RFC 1813, Network Working Group, June 1995.

[3] M. Eisler. NFS Version 2 and Version 3 Security Issues and the NFS Protocol's Use of RPCSEC_GSS and Kerberos V5. Technical Report RFC 2623, Network Working Group, June 1999.

[4] M. Eisler, A. Chiu, and L. Ling. RPCSEC_GSS Protocol Specification. Technical Report RFC 2203, Network Working Group, September 1997.

[5] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the Key Scheduling Algorithm of RC4. In *Proceedings of Eighth Annual Workshop on Selected Areas in Cryptography*, pages 1–24, August 2001.

[6] Fyodor. *NMAP(1)*, 2003. `www.insecure.org/nmap/data/nmap_manpage.html`.

[7] J.H. Howard. An Overview of the Andrew File System. In *Proceedings of the Winter USENIX Technical Conference*, February 1988.

[8] S. Kerr. Use of NFS Considered Harmful. `www.time-travellers.org/shane/papers/NFS_considered_harmful.html`, November 2000.

[9] R. Macklem. Not Quite NFS, Soft Cache Consistency for NFS. In *Proceedings of the Winter USENIX Technical Conference*, pages 261–278, San Francisco, CA, January 1994.

[10] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3 design and implementation. In *Proceedings of the Summer USENIX Technical Conference*, pages 137–52, June 1994.

[11] D. Robinson. The advancement of NFS benchmarking: SFS 2.0. In *Proceedings of the 13th USENIX Systems Administration Conference (LISA '99)*, pages 175–185, Seattle, WA, November 1999.

[12] M. Seltzer and O. Yigit. A new hashing package for UNIX. In *Proceedings of the Winter USENIX Technical Conference*, pages 173–84, January 1991. `www.sleepycat.com`.

[13] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 Protocol. Technical Report RFC 3530, Network Working Group, April 2003.

[14] SMCC. *fsirand(1M)*, September 1996. SunOS 5.8 Reference Manual, Section 1M.

[15] SPEC: Standard Performance Evaluation Corporation. SPEC SFS97_R1 V3.0. `www.spec.org/sfs97r1`, September 2001.

[16] Sun Microsystems. NFS: Network file system protocol specification. Technical Report RFC 1094, Network Working Group, March 1989.

[17] Theodore Ts'o. *urandom(4)*, October 2003.

[18] A. Watson and B. Nelson. LADDIS: A multi-vendor and vendor-neutral SPEC NFS benchmark. In *Proceedings of the Sixth USENIX Systems Administration Conference (LISA VI)*, pages 17–32, October 1992.

[19] E. Zadok. *Linux NFS and Automounter Administration*. Sybex, Inc., May 2001.