Enhancing NFS Cross-Administrative Domain Access *

Joseph Spadavecchia and Erez Zadok
Stony Brook University
{joseph,ezk}@cs.sunysb.edu

Abstract

The access model of exporting NFS volumes to clients suffers from two problems. First, the server depends on the client to specify the user credentials to use and has no flexible mechanism to map or restrict the credentials given by the client. Second, when the server exports a volume, there is no mechanism to ensure that users accessing the server are only able to access their own files.

We address these problems by a combination of two solutions. First, *range-mapping* allows the NFS server to restrict and flexibly map the credentials set by the client. Second, *file-cloaking* allows the server to control the data a client is able to view or access, beyond normal Unix semantics. Our design is compatible with all versions of NFS. We have implemented this work in Linux and made changes only to the NFS server code; client-side NFS and the NFS protocol remain unchanged. Our evaluation shows a minimal average performance overhead and, in some cases, an end-to-end performance improvement.

1 Introduction

NFS was originally designed for use with LANs [17, 22], where a single administrative entity was assumed to control all of the hosts in that site and create unique user accounts and groups. The access model chosen for exporting NFS volumes was simple but weak. In a different administrative domain, the password database may define different users with the same UIDs; a UID clash could occur if files in one domain are accessed from another. Worse, users with local root access on their desktops or laptops can easily access files owned by any other user via NFS, by simply changing their effective UID (i.e., using `/bin/su`).

Therefore, NFS servers rarely export volumes outside their administrative domain. Moreover, administrators resist opening up access even to hosts within the domain, if those hosts cannot be controlled fully. Today, users and administrators must compromise in one of two ways. Either volumes are exported across administrative domains and security is compromised, or the volumes are not exported across administrative domains, preventing users from accessing their data. Neither solution is acceptable.

* Appears in proceedings of the 2002 Annual Usenix Technical Conference, FreeNIX track.

Although NFSv4 [19] promises to provide strong authentication and provides a convenient framework for fixing these problems, it will not be available for many platforms and in wide use for several years. The transition between NFSv2 [22] and NFSv3 [2] took around 10 years and corresponds to relatively small changes compared to the changes between NFSv3 and NFSv4. Even today, NFSv3 is not fully implemented on all platforms. Moreover, the NFSv4 specification does not address all of the problems that we wish to fix. Nevertheless, the techniques described in this paper can enhance NFSv4 functionality. For example, whereas NFSv4 optionally supports ACLs (Access Control Lists), it does not specify how to use them to hide files or consider the idea of hiding files.

Current NFS servers implement a simple form of security check for the super user, intended to stop a root user on a client host from easily accessing any file on the exported NFS volume. However, current NFS servers do not allow the restriction and mapping of any number of client credentials to the corresponding server credentials.

We present a combination of two techniques that together increase both security and convenience: range-mapping and file-cloaking. *Range-Mapping* allows an NFS server to map any incoming UIDs or GIDs from any client to the server's own known UIDs and GIDs. This allows each site to continue to control their own user and group name-spaces separately while allowing users on one administrative domain to access their files more conveniently from another domain. Range-mapping is a superset of the usual UID-0 mapping and Linux's *all-squash* option which maps all UIDs or GIDs to –2.

Our second technique, *file-cloaking*, lets the server determine which ranges of UIDs or GIDs should a client be allowed to view or access. We define *visibility* as the ability of an NFS server to make some files visible under certain conditions. We define *accessibility* as the NFS server's ability to permit some files to be read, written, or executed. Cloaking extends normal Unix file permission checks by restricting the visibility and accessibility of users' files when those files are exported via NFS. Cloaking can be used to enforce the NFS client options `nosuid` and `nosgid` which prevent the execution of set-bit files.

Range-mapping and cloaking complement each other.

Together, they allow NFS servers to extend access to more clients without compromising the existing security of those files. Whereas ACLs can allow a greater degree of flexibility than cloaking, ACLs are not available on all hosts and all file systems, are not supported in NFSv2, and are partially implemented in NFSv3. Furthermore, ACLs are often implemented in incompatible ways; this is one reason why the new NFSv4 protocol specification lists ACL attributes as optional [19].

Our system is implemented in the Linux kernel-mode NFS server. No changes were made to the NFS client side and our system is compatible with existing NFS clients. This has the benefit that we can deploy our system fairly easily by changing only NFS servers.

We performed a series of general-purpose benchmarks and micro-benchmarks. Range-mapping has an overhead of at most 0.6%. File-cloaking overheads range from 72% for a large test involving 1000 cloaked users—to an *improvement* of 26% in performance under certain conditions, reflecting a 4.7 factor reduction in network I/O.

The rest of this paper is organized as follows. Section 2 describes the design of our system and includes several examples. We discuss interesting implementation aspects in Section 3. Section 4 describes the evaluation of our system. We review related works in Section 5 and conclude in Section 6.

2 Design

Range-mapping and cloaking are features that offer additional access-control mechanisms for exporting NFS volumes. We designed these features with three goals: compatibility, flexibility, and performance.

First, we are compatible with all NFS clients by requiring no client-side or protocol changes. Range-mapping and cloaking are performed entirely by the NFS server. The server forces a behavior on the client that maintains compatibility with standard Unix semantics. Second, we provide additional flexible access-control mechanisms that allow both users and administrators to control who can view or access files. We allow administrators to mix standard Unix and cloaking semantics to define new and useful policies. Third, our design is economical and efficient. We utilize hash tables and caches to ensure good overall performance.

2.1 Range-Mapping

Traditionally, NFS supports two simple forms of credential mapping called *root-squashing* and *all-squashing*. Root-squashing allows an NFS server to map any incoming UID 0 or GID 0 to another number that does not have superuser privileges, often –2 (the *nobody* user). All-squashing is a Linux NFS server feature that allows the server to map all incoming UIDs or GIDs to another number, representing a user or group with minimal privileges (e.g., –2).

These two crude forms of credential mapping are not sufficient in many situations. First, there is no way to map arbitrary ranges of client IDs to server IDs. Therefore, squashing is not sufficient in cases where it is desirable to map to more than a single ID. Second, it is useful in some cases to map one set of IDs while squashing all others. For example, if the file system was exported to a client where only UIDs 400–500 should be able to access their files on the server, then it is desirable to assure that all other UIDs coming from the client are squashed to –2.

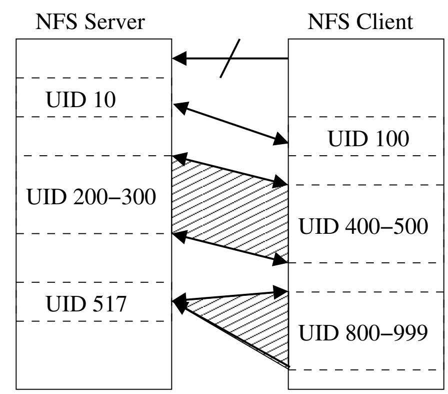

Figure 1: Range-mapping: client UID 100 to server UID 10 (1-to-1); client UIDs 400–500 to server UIDs 200–300 (N-to-N); client UIDs 800–999 to server UID 517 (N-to-1). Note that server UID 517 is reverse-mapped to client UID 800. All other UIDs are restricted.

Range-mapping allows greater flexibility in mapping credentials. Range-mappings are defined per *export* (exported file system) and allow the NFS server to restrict or map any set of an NFS client's credentials to the corresponding set on the server. Figure 1 shows an example of several range-mappings. First, client UID 100 is mapped to server UID 10 (1-to-1). Second, client UIDs 400–500 are mapped to server UIDs 200–300 (N-to-N), respectively. Third, client UIDs 800–900 are mapped to server UID 517 (N-to-1); this case is also called *squashing*. For N-to-1 mappings, the server reverse-maps the single server UID to the first corresponding client UID in the squashed range: server UID 517 is reverse-mapped to client UID 800 in Figure 1. Finally, the arrow with a slash through it means that all other UIDs are restricted (squashed to –2).

Range-mapping is done bidirectionally. Forward mapping is done when a client sends a request to the NFS server and the server maps the user's client UID and GID to the corresponding server UID and GID. Reverse mapping is done when the server responds to the client (i.e., when returning file attributes) and must map the user's server UID and GID back to the corresponding client UID and GID.

2.1.1 Range-Mapping Configuration Examples

Range-mapping definitions are specified per export in `/etc/exports` as part of the option list. To provide human-readable formatting of the range mapping definitions, we extended the file's format to support whitespace and line continuation. The syntax for the range mapping option is as follows:

```
range_map = rmap_def [rmap_def...]
rmap_def := <uid|gid> rm-low [rm-high]
            <map|squash> lc-low
```

The first option in `rmap_def` specifies whether the definition applies to UIDs or GIDs. The *rm-low*, *rm-high*, and *lc-low* values specify the remote lower bound, remote upper bound, and local lower bound ID values, respectively. The client ID range *rm-low* to *rm-high* is mapped to the server's ID range *lc-low* as follows:

$$lc\text{-}low + (rm\text{-}high - rm\text{-}low)$$

A one-to-one mapping from ID *rm-low* on the client to ID *lc-low* on the server is performed if *rm-high* is not specified or is equal to *rm-low*.

The `map` or `squash` option specifies that an N-to-N, or N-to-1 ID mapping is being performed, respectively. The following example shows a single range mapping definition:

```
/home *.example.com(rw, \
                    range_map = \
                    uid 100 250 map    12314 \
                    gid 100 200 squash  6000)
```

The definition above specifies two range-maps for clients who are members of the `example.com` domain and are accessing the `/home` volume. First, client UIDs 100–250 will be mapped to server UIDs 12314–12464. Second, client GIDs 100–200 be squashed to server GID 6000.

Range-mapping is a superset of root-squashing and all-squashing. Root squashing is a feature of NFS that allows the NFS server to map the root UID (0) and GID (0) to the nobody UID (typically –2) and nobody GID (typically –2). The following example shows how to perform root squashing using range-mapping:

```
/home  *.example.com(rw, \
                     range_map = \
                     uid  0  squash  -2 \
                     gid  0  squash  -2)
```

All-squashing is a feature supported in Linux that maps all client UIDs and GIDs to –2. All-squashing may be done with range-mapping as follows:

```
/home  *.example.com(rw, \
                     range_map = \
                     uid  0  -1  squash  -2 \
                     gid  0  -1  squash  -2)
```

2.2 File-Cloaking

File-cloaking is a mechanism that abstracts the concept of file permissions to allow data to be hidden. With file-cloaking, if a file is not visible then it is not accessible. When a request is made to access a file or list a directory, file-cloaking uses the NFS credentials, file protection bits, and cloaking-mask to determine access and visibility.

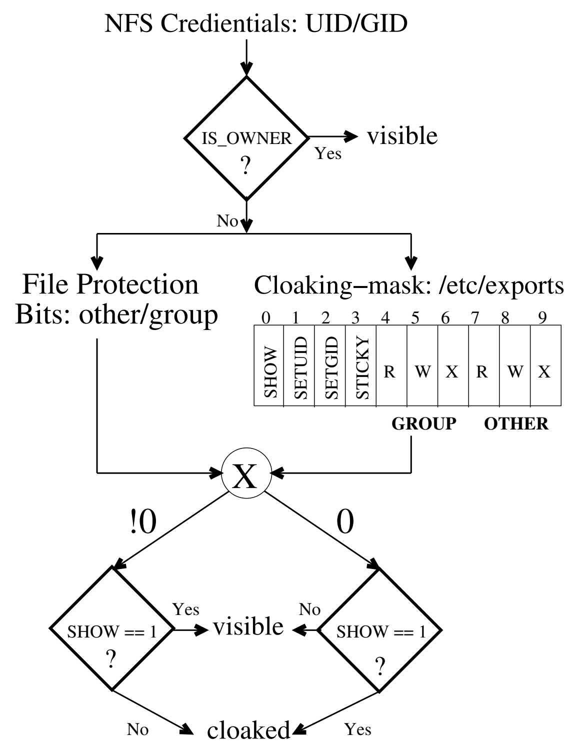

Figure 2: File-cloaking algorithm running on the NFS server.

File	Permission	User	Group
J1	0600	joe	src
J2	0640	joe	src
J3	2666	joe	src
J4	0700	joe	src
E5	0750	ezk	src
E6	0750	ezk	fac
E7	4775	ezk	src
E8	0775	ezk	fac
E9	6700	ezk	src
E10	0000	ezk	src

Table 1: An example listing of files. User `joe` *belongs to group* `src`*, and user* `ezk` *belongs to groups* `src` *and* `fac`*. The leftmost 3 bits attached to the permission mask represent the SETUID, SETGID, and sticky bits, respectively.)*

Figure 2 shows how file-cloaking works. The NFS credentials are checked against the owner of the file.

Cloak Mask	User **ezk**				User **joe**						Meaning for files J1–E10
	J1	J2	J3	J4	E5	E6	E7	E8	E9	E10	
+000											Show files to owners only
+007			A				A	A			Show files to owners and others
+070		A	A		A		A	A			Show files to owners and group members
+077		A	A		A		A	A			Show files to all people with any access
–007	v	v		v	v	v			v		Hide world-accessible files
–070	v			v					v		Hide from group members
–077	v			v							Hide from groups members and others
–004	v	A		v	A	v	A	A	v	v	Hide world readable files
–400	v	A	A	v	A	v		A		v	Hide SETUID files
–200	v	A		v	A	v	A	A		v	Hide SETGID files
–000	v	A	A	v	A	v	A	A	v	v	Unix standard

Table 2: File visibility and accessibility for various cloak masks, given the files in Table 1. A "+" in front of the mask implies that files are cloaked by default; a "–" means that files are visible by default. The letter "A" means that the file is visible and accessible. The letter "v" means the file is visible but not accessible. A blank cell means that the file is neither visible nor accessible.

If the credentials match then the file is visible. Otherwise two checks are done, each with an AND between the cloaking-mask and the file's protection bits. The first check is done if the credentials have group ownership on the file. The group RWX bits of the cloaking-mask are ANDed with the file's protection bits. The second check is done with an AND between all the bits in the cloaking-mask (excluding the group rwx bits) and the file's protection bits (excluding the group and user bits). If the result of either AND is non-zero and the SHOW bit of the cloaking-mask is set then the file is visible; else if the SHOW bit is unset then the file is cloaked. The inverse behavior takes place if the result of the AND is zero. In this way, the SHOW bit can determine whether files are visible or cloaked by default, allowing the other mask bits to reverse the default behavior.

Tables 1 and 2 illustrate the concept of cloaking and provide some examples of useful cloaking configurations. Table 1 shows a sample of files owned by two different users: `joe` and `ezk`. The permissions for each file are listed including the SETUID, SETGID, and sticky bits. User `joe` belongs to the `src` group, and user `ezk` belongs to the `src` and `fac` groups. Table 2 shows a visibility/accessibility matrix of the files listed in Table 1 for eleven useful cloaking masks. In Table 2, "A" implies "v" and the "+" or "–" preceeding the mask sets or clears the mask's SHOW bit, respectively. Next, we describe a few of the examples in Table 2.

With mask +000, the administrator enforces a strict policy that users can only see their own files. On the other hand, with mask –077 the administrator chooses a policy that allows users to decide if their files should be hidden or not by setting the group and world bits. To hide a file from everyone, the user may set any of the "other" bits. To hide a file from group members, the user may set one of the group bits. Note that this mask is non-intuitive because it restricts access by performing an action that would normally increase others' access. Also, note that this option is not fail-safe. If the administrator changes the export option hidden files might not only be visible, but also accessible.

The next to last two lines in Table 2 show that cloaking allows the server to enforce the client-side `nosuid` and `nosgid` mount options. The `nosuid` and `nosgid` options are achieved by cloaking using the masks –400 and –200, which translate to hide SETUID and hide SETGID files, respectively. Note that cloaking here is more restrictive than the `nosuid` and `nosgid` options. These options allow files to be executed, but without their set-bits. Cloaking hides such files and thus disallows their execution altogether.

Another example is the mask –002, which hides all world-writable files. This mask can be especially useful on multi-user systems to protect layman users from leaving their files world writable. In this scenario the administrator enforces a policy on the systems users.

2.2.1 File-Cloaking Configuration Examples

Cloaking definitions are specified with range-mappings in the `/etc/exports` file. The syntax for cloaking definitions is as follows:

```
cloak_list = clist_def [clist_def...]
clist_def := <uid|gid> mask lc-low [lc-high]
```

As with range-mapping, the first option specifies whether the definition applies to UIDs or GIDs. Next, `mask` is a 10-bit field for defining the cloaking policy. Figure 2 shows the different bits comprising the mask.

The range *lc-low* to *lc-high* is the range of server IDs to cloak based on the policy given by the mask. If *lc-high* is omitted, then the definition applies only to the server

ID *lc-low*. The following example shows a cloaking definition:

```
/home  *.example.com(rw, \
                      cloak_list = \
                      uid  +000  500  1000 \
                      gid  +077  100   200)
```

In this example, there are two cloaking definitions. The first places the restriction for UIDs 500–1000 that only the owners may see or access their files. That is, files on the server owned by UIDs 500–1000 cannot be seen or accessed by any user other than the owner. The second definition states that files owned by UIDs 100–200 are only visible if there is world access to them, or if there is group access to them and the user listing the file belongs to the group of the file.

3 Implementation

We implemented this project in two places: the NFS Utilities (`nfs-utils`) package version 0.3.1 [8] and the NFS server code (both NFSv2 and NFSv3) in the Linux 2.4.4 kernel. We added 1678 lines of code to `nfs-utils`, an increase of 5.8% to its size. This code primarily handles parsing `/etc/exports` files for range-mapping and cloaking entries, packing them into exports structures, and passing these structures to the kernel using a special-purpose `ioctl` used by the in-kernel NFS server.

Most of the code we added was to the kernel: 1330 lines of additional code, or an increase of 15.1% to the total size of the NFS server sources. Although this increase is substantial, the bulk of our changes to the kernel code are in new C source and header files, and in stand-alone functions we added to existing source files. The placement of our changes in the kernel sources made it easier to develop: two first-year graduate students spent a combined total of 12 man-weeks developing and testing the code.

3.1 Range-Mapping

For range-mapping we faced three questions: where to do forward mapping, where to do reverse mapping, and how to get the mapping context from the export structures for each client request.

Forward mapping is done in the `nfsd_setuser` function, which is passed a pointer to the relevant export structure; the latter contains the information we need to perform the mapping. Implementing reverse mapping was more difficult. The best place for it was in the server's outgoing path, where it encodes file attributes into XDR structures before shipping them back to the NFS client. This is done in the `encode_fattr3` routine (or `encode_fattr2` for NFSv2). We find a response packet inside the request structure passed to this function; from this we get the NFS file handle. The latter contains the export information we need to compute the range-mapping.

3.2 Cloaking

Cloaking was more challenging to implement than range-mapping because of the restriction that we only modify the server. Cloaking needs to display different directory listings to each user on the same client. Since clients cache directory contents and file attributes, we have to force the NFS clients to ignore cached information (if any) and reissue an NFS_READDIR procedure every time users list a directory. We investigated two options: (1) lock the directory, and (2) fool the client into thinking that the directory's contents changed and thus must be re-read. We chose the second option because locking the directory permanently would have serialized all access to that directory and prevented more than one NFS client from making changes to that directory (such as adding a new file).

To force the client to re-read directories, we increment the in-memory modification time (mtime) of the directory each time it is listed; we do not change the actual directory's mtime on disk. This technique has been used before to prevent client-side caching in NFS-based user level servers [14, 25, 29]. NFS clients check the mtime of remote directories before using locally cached data. Since the mtime always changes, the clients re-read the directory each time and effectively discard their local cache. The mtime field has a resolution of one second, but sometimes several `readdir` requests come in one second. We therefore had to ensure that the mtime is always incremented on each listing. This has a side effect that the modification time of directories being listed frequently could move into the future. In practice this was not a problem because directory-reading requests are often bursty and in between bursts the real clock has a chance to catch up to a directory's mtime that may be in the near future. Furthermore, future Linux kernels will increase the mtime resolution to microseconds, thus practically eliminating this problem.

We expected that forcing the clients to ignore their directory caches will reduce performance. However, if a client machine has only one user (as is the case with most personal workstation and laptops), we can allow the client to cache directory entries normally since there is little risk that another user on that client will be able to view cached entries. We made client caching optional by adding a server-side export option called `no_client_cache` that, if enabled, forces the directory mtime to increase and cause clients not to cache directory entries. If `no_client_cache` is not used (the default), we do not increase the mtime and NFS clients cache directory entries normally.

Cloaking requires that some files be hidden from users and therefore those files' names should not be sent back to the NFS client. We implemented this in the `encode_entry` function. Given a file's owner, group, mode bits, and the export data structures, we compute whether the file should be visible or not. If the file is invisible, we simply skip the XDR encoding of that file. If the file is not invisible, then we allow access to that file based on normal Unix file permissions. A user could try to lookup (perhaps guess) a file that is hidden to that user. To catch this we perform a cloaking check also in `nfsd_lookup` and if the file should be invisible to the calling user, we return an error code back to the lookup request.

4 Evaluation

To evaluate range-mapping and cloaking in a real world operating environment, we conducted extensive measurements in Linux comparing vanilla NFS systems against those with different configurations of range-mapping and cloaking. We implemented range-mapping and file-cloaking in NFSv2 and NFSv3; however, we report the results for NFSv3 only. Our benchmarks for NFSv2 show comparable performance. In this section we discuss the experiments we performed with these configurations to (1) show overall performance on general-purpose workloads, and (2) determine the performance of individual common file operations that are affected the most by this work. Section 4.1 describes the testbed and our experimental setup. Section 4.2 describes the file system workloads we used for our measurements. Sections 4.3 and 4.4 present our experimental results.

4.1 Experimental Setup

We ran our experiments between an unmodified NFS client and an NFS server using five different configurations:

1. **VAN:** A vanilla setup using an unmodified NFS server. Results from this test gave us the basis on which to evaluate the overheads of using our system.
2. **MNU:** Our modified NFS server with all of the range-mapping and cloaking code included but not used. This test shows the overhead of including our code in the NFS server while not using those features.
3. **RMAP:** Our modified NFS server with range-mapping configured in `/etc/exports`. Since our range-mapping code works exactly the same when a single UID or a range of UIDs are mapped, for simplicity these tests mapped a single UID.
4. **CLK:** Our modified NFS server with cloaking configured in `/etc/exports`. To illustrate the worst-case performance for cloaking, we set the cloaking mask to +000, indicating the most restrictive cloaking possible. Using +000 ensures that the code which determines if a file should be visible or not checks as many mask conditions as possible, as we described in Section 2.
5. **RMAPCLK:** Our modified NFS server with both range-mapping and cloaking configured in `/etc/exports`. To ensure worst-case performance, we set the user entries that are range-mapped so they are also cloaked.

The last three configurations were intended to show the different overheads of our code when each feature is used alone or combined. Since our system runs the same range-mapping or cloaking code with either UIDs or GIDs we evaluated range-mapping and cloaking only for UIDs.

All experiments were conducted between two equivalent Dell OptiPlex model GX110 machines that use a 667MHz Intel Pentium III CPU, 192MB of RAM, and a Maxtor 30768H1 7.5GB IDE disk drive. The two machines were connected to a stand-alone dedicated switched 100Mbps network.

To ensure that our machines were equivalent and our setup was stable, we ran the large-compile Am-utils benchmark (see Section 4.2.1) on both machines, alternating the server and client roles of the two. We compiled the package on the client, using an exported file system from the server. We compared the results and found the difference in elapsed times to be 1.003% and the difference in system time (as measured on the client) to be 1.012%. The standard deviations for these tests ranged from 2.2–2.7% of the mean. Therefore, for the purposes of evaluating our systems, we consider these machines equivalent, but we assume that benchmark differences smaller than 1% may not be significant.

We installed a vanilla Linux 2.4.4 kernel on both machines. We designated one machine as client and installed on this machine an unmodified NFS client-side file system module. On the server we installed both a vanilla NFS server-side file system module and our modified NFS server module that included all of the range-mapping and cloaking code. Since our code exists entirely in the NFS server, we could use a vanilla kernel on the server and simply load and unload the right kNFSd module.

On the server we installed our modified user-level NFS utilities that understand range-mapping and cloaking; these include the `exportfs` utility and the `rpc.nfsd`, rpc.lockd, and rpc.mountd daemons. Finally, the server was configured with a dedicated 334MB EXT2FS partition that we used exclusively for exporting to the client machine.

All tests were run with a cold cache on an otherwise quiescent system (no user activity, periodic cron(8) jobs turned off, unnecessary services disabled, etc.). To ensure that we used a cold cache for each test, we unmounted all file systems that participated in the given test after the test completed, and we mounted the file systems again before running the next iteration of the test (including the dedicated server-side exported EXT2FS partition). We also unloaded all NFS-related modules on both client and server before beginning a new test cycle. We verified that unmounting a file system and unloading its module indeed flushes and discards all possible cached information about that file system.

We ran each experiment 20 times and measured the average elapsed, system, and user times. In file system and kernel benchmarks, system times often provide the most accurate representation of behavior. In our tests we measured the times on the NFS client, but the code that was modified ran on the server's kernel. System times reported on a client do not include the CPU time spent by the server's kernel because when the server is working on behalf of the client, the client's user process is blocked waiting for network I/O to complete. I/O wait times are better captured in the elapsed time measurements (or, alternatively, when subtracting system and user times from elapsed times). Since our testbed was dedicated and the network connection fast, we chose to report elapsed times as the better representatives of the actual effort performed by both the client and the server.

Finally, we measured the standard deviations in our experiments and found them to be small: less than 3% for most benchmarks described. We report deviations that exceeded 3% with their relevant benchmarks.

4.2 File System Benchmarks

We measured the performance of our system on a variety of file system workloads and with the five different configurations as described in Section 4.1. For our workloads, we used three file system benchmarks: two general-purpose benchmarks for measuring overall file system performance and one micro-benchmark for measuring the performance of common file operations that may be affected by our system.

4.2.1 General-Purpose Benchmarks

Am-utils The first general-purpose benchmark we used to measure overall file system performance was am-utils (The Berkeley Automounter) [13]. This benchmark configures and compiles the am-utils software package inside a given directory. We used am-utils-6.0.7: it contains over 50,000 lines of C code in 425 files. The build process begins by running several hundred small configuration tests intended to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation: a total of 265 additional files. Overall, this benchmark contains a large number of reads, writes, and file lookups, as well as a fair mix of most other file system operations such as unlink, mkdir, and symlink. The main usefulness of this benchmark is to show what the overall performance might be for regular use of our system by users.

NFSSTONE The second general-purpose benchmark we used was an NFS-specific benchmark called NFSSTONE [18]. This traditional benchmark performs a series of 45522 file system operations, mostly executing system calls, to measure how many operations per second can an NFS server sustain. The benchmark performs a mix of operations intended to show typical NFS access patterns [16]: 53% LOOKUPs, 32% READs, 7.5% READLINKS (symlink traversal), 2.3% GETATTRs, 3.2% WRITEs, and 1.4% CREATEs. This benchmark performs these operations as fast as it can and then reports the average number of operations performed per second, or *NFSSTONES*.

For these two general benchmarks, we set up the NFS server with the an /etc/exports file configured as follows:

- For the VAN and MNU tests, the exports files contained only one entry exporting a single file system. No range-mapping or cloaking were configured.
- For the RMAP test we configured 10 range-mapped users, representing a small configuration we believe will be common. The user we ran these benchmarks on the client was one of the mapped UIDs. This way we caused the server to do some work, bidirectionally mapping one user ID to another.
- For the CLK test we configured 10 cloaked users. The user we ran the benchmarks on the client was one that was allowed to access and modify their files on the server. This test shows the worst-case scenario for cloaking, when the server has to look at the entire list of cloaked users and not find the user who is accessing the files.
- For the RMAPCLK test we configured an /etc/exports file that contained 10 range-mapped entries and also 10 cloaked entries. The user we ran the test on the client was range-mapped and had permission to view and modify their files.

We investigated three additional benchmarks that we did not use to evaluate our work. First, the Modified

Andrew Benchmark (MAB) [11] is also a compile-based benchmark but it is too small for modern hardware and completes too quickly as compared to the larger am-utils compile. Second, a newer version of NFSSTONE called *NHFSSTONE* [7] uses direct RPC calls to a remote NFS server instead of executing system calls on the client because the latter can result in a different mix of actual NFS server operations that are executed. Unfortunately, the only available NHFSSTONE benchmark for Linux [8] supports only NFSv2, whereas we wanted a benchmark that could run on both NFSv2 and NFSv3 servers [2, 12, 17, 22]. Third, the SFS 2.0 benchmark, a successor to LADDIS [24], is a commercial benchmark that provides an industry-standardized performance evaluation of NFS servers (both NFSv2 and NFSv3), but we did not have access to this benchmark [14, 23]. Nevertheless, we believe that the benchmarks we did perform represent the performance of our system accurately.

4.2.2 Micro-Benchmarks

GETATTR: The third benchmark we ran is the primary micro-benchmark we used. Our code affects only file system operations involving accessing and using file attributes such as owner, group, and protection bits. This benchmark runs a repeated set of recursive listing of directories using `ls -lR`. That way, the micro-benchmark focuses on the operations that are affected the most: getting file attributes and listing them.

Since this benchmark is the one that is affected the most by our code, we ran this test repeatedly with several different configurations aimed at evaluating the performance and scalability of our system. To ensure that the server had the same amount of disk I/O to perform, we used fixed-size directories containing exactly 1000 files each.

To test the scalability, we ran some tests with a different numbers of range-mapped or cloaked entries in `/etc/exports`: 10, 100, and 1000. Ten entries intends to represent a small site whereas one-thousand entries represents a large site.

For the VAN and MNU tests we used a directory with 1000 zero-length files owned by 1000 different users. These two tests show us the base performance and the effect on performance that including our code has, respectively.

For the RMAP test we also kept the size of the directories being listed constant, but varied the number of UIDs being mapped: 10 mapped users each owning 100 files, 100 mapped users each owning 10 files, and 1000 users each owning one file. The directories were created such that each user's files were listed together so we could also exercise our range-mapping UID cache. The user that ran the `ls -lR` command for this benchmark on the client was one of the mapped UIDs. (It does not matter which of the mapped users was the one running the test since the entire directory was listed and for each file the server had to check if range-mapping was applicable.) These tests show what effect range-mapping has on performance for different scales of mapping.

For the CLK test we used a similar setup as with range mapping: directories containing 1000 files owned by a different number of cloaked users each time: 10, 100, and 1000. To make the server perform the most work, we ran the benchmark on the client using a user whose UID was not permitted to view any of the files listed. This ensured that the server would process every file in the directory against the user who is trying to list that directory, but would not return any file entries back to the client. This means that although the directory contains 1000 files on the server, the client sees empty directories. This has the effect of reducing network bandwidth and the amount of processing required on the client.

The RMAPCLK test combined the previous two tests, using a different number of range-mapped users all of whom were cloaked: 10, 100, and 1000. The directories included 1000 files owned by a corresponding number of users as were mapped and cloaked. The user we ran the test as, on the client, was one of those users to ensure that the server had to process that user's UID both for mapping and cloaking. This guaranteed that only the files owned by the cloaked user (out of a total of 1000 files) would be returned to the client: 100 files were returned when there were 10 cloaked users, 10 files returned when there were cloaked 100 users, and only one file returned when there were 1000 cloaked users. This test therefore shows the combination of two effects: range-mapping and cloaking make the server work harder, but cloaking also results in reducing the number of files returned to the client, and thus saving on network bandwidth and client-side processing.

Finally, for all benchmarks involving cloaking (CLK and RMAPCLK) we ran the tests with and without the `no_client_cache` option (Section 3.2), to evaluate the effects of circumventing NFS client-side caching.

4.3 General-Purpose Benchmark Results

Am-Utils Figure 3 shows the results of our am-utils benchmark. This benchmark exercises a wide variety of file operations, but the operations affected the most involve getting the status of files (via `stat(2)`), something that does not happen frequently during a large compilation; more common are file reads and writes. Therefore the effects of our code on the server are not great in this benchmark, as can be seen from the individual results.

When our code is included but not used (MNU vs. VAN)

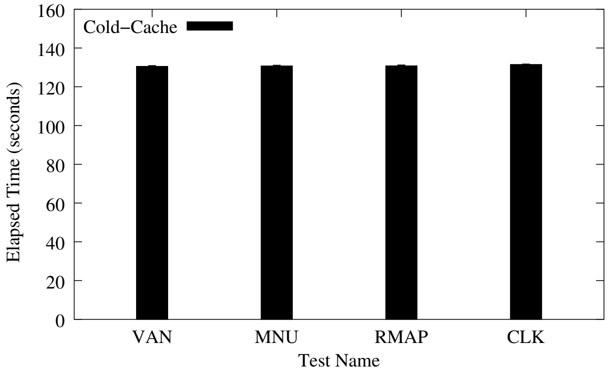

Figure 3: Results of the am-utils benchmark show a small difference between all of the tests—less than 1% between the fastest and slowest results.

we see a 0.14% degradation in performance. Adding range-mapping (RMAP vs. MNU) costs an additional 0.026% in performance. Adding cloaking (CLK vs. MNU) costs an additional 0.54% in performance. These results suggest that range-mapping and cloaking have a small effect on normal use of NFS mounted file systems.

NFSSTONE Figure 4 shows the results of the NFSSTONE benchmark. This benchmark exercises the operations that would be affected by our code on the server (GETATTR) more times than the Am-utils benchmark. Therefore, we see higher performance differences between the individual tests.

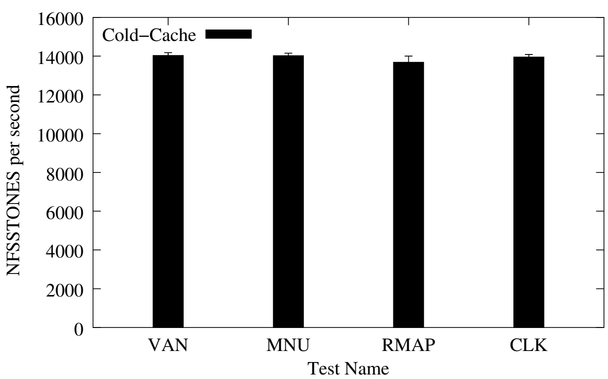

Figure 4: Results of the NFSSTONE benchmark show a small difference between all of the tests—less than 2.6% between the fastet and slowest.

When our code is included but not used (MNU vs. VAN) we see a small 0.05% degradation in performance, suggesting that when our code is not used, it has little effect on performance. Adding range-mapping (RMAP vs. MNU) costs an additional 2.5% in performance. Adding cloaking (CLK vs. MNU) costs an additional 0.54% in performance. These results also suggest that range-mapping and cloaking have a small effect on normal use of NFS mounted file systems.

4.4 Micro-Benchmark Results

Our code affects operations that get file attributes (`stat(2)`). The micro-benchmarks tested the effect that our code has on those operations using a variety of server configurations.

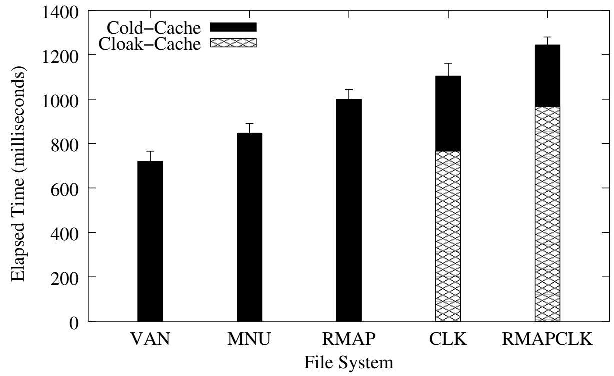

Figure 5: Results of recursive listing of directories containing exactly 1000 entries on the server. The tests were configured to result in exactly 1000 files being returned to the client each time (i.e., worst case). The Cloak-Cache bars show results when NFS clients used cached contents of cloaked directories (i.e., the `no_client_cache` export option was not used). Standard deviations for this tests were 4.3–6.5% of the mean.

Figure 5 shows results of our first micro-benchmark: a recursive listing (`ls -lR`) we conducted on an NFS-mounted directory containing 1000 entries. The `/etc/exports` file on the server was configured with range-mapping and cloaking such that the user listing that directory on the client would always see all files. This ensures that the amount of client-side work, network traffic, and server-side disk I/O for accessing the directory remained constant, while making the server's CPU experience different workloads.

The black bars in Figure 5 show the worst-case results, when NFS clients were ignoring their cache: `no_client_cache` was used on the server, described in Section 3.2. The Cloak-Cache bars show the results when this option was not used, thus allowing clients to use their directory caches.

When our code is included but not used (MNU vs. VAN) we see a 17.6% degradation in performance. This is because each time a client asks to get file attributes, our code must check to see if range-mapping or cloaking are configured for this client. This checking is done by comparing four pointers to NULL. Although these checks are simple, they reside in a critical execution path of the server's code—where file attributes are checked often.

Adding range-mapping (RMAP vs. MNU) costs an additional 18.1% in performance. This time the server scans a linked list of 1000 range-mapping entries for the client, checking to see if the client-side user is mapped or not. Each of 1000 files were owned by a different user and one user was range-mapped. Therefore the server had to scan the entire list of range mappings for each file listed; only one of those files actually got mapped. Such a test ensures that the range-mapping cache we designed was least effective (all cache misses), to show the worst-case overhead.

The cloaking test (CLK vs. MNU) costs an additional 30.3% in performance. This test ensured that the files in the directory were not owned by a cloaked user. Cloaking must guarantee that NFS clients do not use cached file attributes. Therefore the client gets from the server all of the files each time it lists the directory. Moreover, the code that determines if a file should be cloaked or not is more complex as it has to take into account cloak masks. Range-mapping, in comparison, uses a simpler check to see if a UID or GID is in a range of mapped ones; that is why cloaking costs more than range-mapping (10.4% more).

The cumulative overhead of range-mapping and cloaking, when computed as the worst case time difference between cloaking and range-mapping, compared against MNU, is 48.4%. However, when combining cloaking and range-mapping together (RMAPCLK vs. MNU) we see a slightly smaller overhead of 46.9%. This is because the range-mapping and cloaking code is localized in the same NFS server functions. This means that when the NFS server invokes a function look up a file, both range-mapping and cloaking tests are performed without having to invoke that function again, pass arguments to it, etc.

The worst-case situation (RMAPCLK vs. VAN) has an overhead difference of 72.3%. This overhead is for the inclusion of our code and its use on a large directory with 1000 files. Although this overhead is high, it applies only when getting file attributes. Under normal user usage, performance overhead is smaller, as we have seen in Section 4.3.

Finally, for single-user NFS clients, the server can safely allow such hosts to cache directory entries, thus improving performance. The Cloak-Cache results in Figure 5 show what happens when we did not use the `no_client_cache` export option. For the CLK test, client-side caching improves performance by 30.6%. Client-side caching improves performance of the RMAPCLK test by 22.3%.

The micro-benchmarks in Figure 5 show the worst case performance metrics, when both the server and client have to do as much work as possible. The second set of micro-benchmarks was designed to show the performance of more common situations and how our system scales with the number of range-mapped or cloaked entries used. These are shown in Figure 6.

The range-mapping bars (RMAP) show the performance of listing directories with 1000 files in them, but

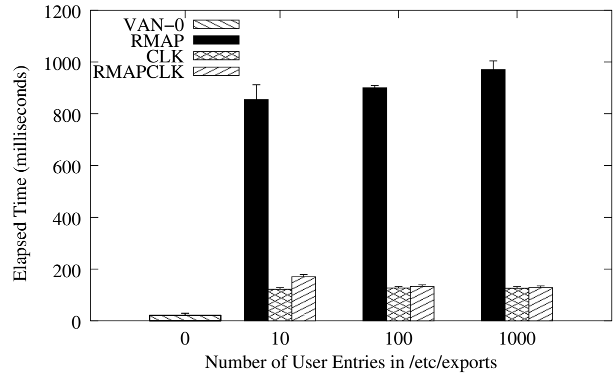

Figure 6: Results of recursive listing of directories containing a different number of files while the server is configured with a different number of mapped or cloaked entries. Standard deviations for this tests were 4.3–6.7% of the mean.

varying the number of users that were range-mapped. Range-mapping with 10 users implies that each user owns 100 files. When one of those users lists the directory, that user sees 1000 files, but 100 of those files have their UID mapped by the server. For 100 files and users, only 10 files are mapped; for 1000 files and users, only one file is mapped. The largest cost for range-mapping is the number of range-map entries listed in `/etc/exports` because the server has to scan the entire (in-memory) list to determine the range-mapping needed for each user that is accessing from the client and for each file that the server wants to present to the client (reverse mapping). Using 100 mappings slows performance by 5.3% compared to 10 mappings; using 1000 mappings costs another 7.9% in performance. Despite two orders of magnitude difference in number of entries used in these three bars, the overall overhead is just 13.6%.

The bars for cloaked configurations (CLK) show a different behavior than range-mapped configurations. Here, all 1000 files were owned by cloaked users, whether there were 10, 100, or 1000 cloaked users. But the user that listed the files on the client was not one of those cloaked users and therefore was not able to see any of those files; what they listed appeared on the client as an empty directory. This test fixes the amount of work that the client has to do (list an empty directory) and the server's work (scan 1000 files and apply cloak rules to each file). This test differs in the number of cloaked user entries listed in `/etc/exports`. The bars show a small difference in the amount of work that the server had to do to process the cloak lists: 4.1% performance difference between the largest and smallest lists. However, just using the cloaking code costs in performance, even if the client receives no files. We compared this cloaking to listing an empty directory on a vanilla NFS server without cloaking code (marked as VAN-0 in Figure 6);

cloaking is 17.8 times slower than listing an empty directory, showing that whereas the client has little work to do, the server must still process a large directory.

Although cloaking consumes more CPU than range-mapping (as seen in Figure 5), the difference between the bars is smaller than with range-mapping and the bars themselves are smaller: 7.3 times faster. The main reason for this is that this cloaking test returns no files back to the client, saving a lot on network bandwidth and client-side CPU processing. This shows on one hand an interesting side effect to cloaking: a reduction in network I/O and client-side processing. On the other hand, to make cloaking work reliably, we had to ensure that the clients would not use cached contents of directories with cloaked files. This potentially increases the amount of network I/O and client-side processing as clients have to scan directories through the server each time they list a directory. We explore these opposing interactions next.

The last set of bars in Figure 6 shows the performance when combining range-mapping with cloaking (RMAPCLK). Since all of the users' files were cloaked and range-mapped, and the user that listed the directory on the client was one of those users, then that user saw a portion of those files (the files they own). With 10 cloaked users, 100 files were seen by the client; with 100 cloaked users, 10 files were seen; and with 1000 cloaked users, only one file was seen. This means that the amount of work performed by the client should decrease as it lists fewer files and has to wait less time for network I/O. The RMAPCLK bars indeed show an *improvement* in performance as the number of cloaked user entries increases. The reason for this improvement is that the savings in network I/O and client-side processing outweigh the increased processing that the server performs on larger cloak lists. Listing the same directory when we use 100 cloaked and range-mapped entries is 22.3% faster than the directory with 10 entries, because we are saving on listing 90 files. Listing the directory with cloaked 1000 entries is only an additional 4% faster because we are saving on listing just 9 files.

To find out how much cloaking saves on network I/O, we computed an estimate of the I/O wait times by subtracting client-side system and user times from elapsed times. We found that for a combination of cloaking and range-mapping with 10 users, network I/O is reduced by a factor of 4.7. Since cloaking with the `no_client_cache` option forces clients to ignore cached directory entries, these immediate savings in network I/O would be overturned after the fifth listing of that directory. However, without the `no_client_cache` option, network I/O savings will continue to accumulate.

5 Related Work

The closest past related works to ours are BSD-4.4 umapfs and the older and now defunct user-level NFS server for Linux, Unfsd [5]. BSD-4.4 umapfs works on the client and is not enforced by the server; thus credentials cannot be controlled by the server and this solution is not as secure as our server-side range-mapping. The Linux Unfsd server included extensions that supported UID and GID mapping or squashing via NIS or an external RPC server called `ugidd`. This user-level NFS server had several serious deficiencies: it was slow due to context switches and data copies between user level and the kernel; it did not reliably support many important features such as the RPC LOCKD protocol; and it had several security flaws that could not be solved unless the server ran in kernel mode. For those reasons current versions of Linux include a kernel-mode NFS server named *kNFSd*. When that server code was written (from scratch), it did not include support for range-mapping. Our work has added support for range-mapping, squashing, and cloaking into the Linux kernel. We achieved good performance while offering flexible features that were not available before (e.g., cloaking), not even in Unfsd.

Today's NFS servers include a feature to suppress access from UID 0 or GID 0, also known as *root squashing* [2, 12, 17, 22]. Linux also includes a feature called *all squashing* that maps all incoming UIDs or GIDs to a single number. As we saw in the design section, our work is a superset of these forms of UID or GID squashing (root or all). With file-cloaking, we support a superset of masking features that includes the ability to disable SETUID or SETGID binaries from executing over NFS.

NFS runs on top of RPC (Remote Procedure Calls) and it is possible to secure NFS by securing the RPC layer. Several secure-RPC systems exist that support even strongly-secure systems such as Kerberos [1, 9, 21]. Unfortunately, past secure RPC systems used with NFS did not always interoperate well with each other and are not available for all existing NFS implementations. To use a system such as Kerberos, NFS clients, servers, and even some user applications have to be modified to support Kerberos; consequently, most NFS systems use the weaker form of user authentication known as AUTH_UNIX, where NFS clients inform the servers what UID and GID they use. Even with strong RPC security, NFS servers still do not support features such as cloaking and range-mapping. Our work does not aim to replace strong security, but rather to show how additional access methods that improve security can be implemented and deployed easily, and that these changes have little effect on overall performance.

A newer version of the protocol, NFSv4, promises to support many new features including mandatory strong security, protocol extensibility, support for wide-area file access, and better interoperability between Unix and non-Unix systems [19, 26]. In NFSv4, users need not be determined by their UID and GID, but by a universal identifier such as an Email address or an electronic signature. Doing so will help identify users uniquely throughout the Internet and would alleviate the need for range-mapping.

Although the current NFSv4 specification does not support cloaking, the protocol was designed for extensibility. Our cloaking techniques do not require a change in the NFS protocol and can be implemented in exactly the same way: our work is therefore compatible with NFSv4 as well as with older NFS protocols. However, as discussed in Section 3.2, our cloaking implementation may be configured to force NFS clients not to cache file attributes or use them, to ensure the correctness of the data that is given to different users on the same client. This costs in performance as we saw in Section 4.4. With NFSv4, cloaking could be added easily as extensions to the protocol that cooperative NFSv4 clients and servers can agree on dynamically. An NFSv4 server can allow an NFSv4 client to cache these entries. If another user tries to list a directory that is cached on the server, the server can then issue a *callback* request to the client (now acting as a small server for these RPC callbacks) to flush the cache before the server sends the client a list of files for that directory; this new list can be different based on the particular view of that directory for another user.

NFSv4 also supports Access Control Lists (ACLs) as optional file attributes. ACLs are also possible with NFSv3, but they are not part of the specification and not all vendors implement ACL support. The reason is that not all operating systems and file systems support ACLs, and those that do are often incompatible (for example, one system supports ACLs only for directories and another system supports them for any file). Whereas ACLs can be an effective and powerful access-control mechanism and are compatible with our cloaking techniques, ACL support remains an optional feature of existing and future NFS protocols.

NFSv2 was widely used in 1994, when NFSv3 was introduced. The protocol has not changed fundamentally: two defunct RPC operations were removed and a few more added; support for TCP and 64-bit files was included too. Still, it took several years before most major vendors began supporting NFSv3 and a few more years to stabilize their code. Today, eight years later, not all vendors who support NFSv2 also support NFSv3, and of those that do, interoperability and stability problems remain. In comparison, NFSv4 represents a large departure from NFSv3: the now stateful protocol integrates all previous RPC services needed to support NFS (such as MOUNTD, LOCKD, and more) and the specification is larger too. The NFSv4 RFC is 212 pages, compared with only 126 pages for the NFSv3 RFC. We expect that it would be several years before NFSv4 is deployed by all vendors and is stable. Even then, older versions of NFS are likely to remain in use. For those reasons, we believe that the work we presented in this paper remains viable for the foreseeable future.

There are several commercial products that can map credentials transparently, such as Network Appliances's filer [10]. Also, AFS [4] can map users transparently by referring to their usernames. However, these solutions are often expensive, are not available as Open Source Software, or not in wide use compared to Linux and NFS.

6 Conclusions

The main contribution of this work is to allow NFS servers to export their file systems to hosts they would not have allowed access before for security reasons. We used two techniques: *range-mapping* and *file-cloaking*.

Range-Mapping translates UIDs and GIDs of users and their files between NFS servers and clients that exist on different administrative domains. This lets users access their files on different sites where their user accounts have different credentials. File-cloaking prevents certain files from being seen or accessed by some users. This lets administrators and users control the accessibility and visibility of their files. One use of this is to allow only the files' owners to see their own files; or to prevent world-writable or SETUID files from being seen or accessed by anyone other than their owners; or ensuring that only members of the same Unix group could see and write to a shared file. Cloaking gives administrators the flexibility to safely export multi-user file systems to clients of a single user where that user may not be trusted with the remaining files on the exported volume.

We designed and implemented this work to be contained entirely inside the NFS server and require no changes to the NFS protocol or clients. This ensured that our work can interoperate with all existing NFS clients without modification.

Special care was taken to ensure that performance of our system was good and that kernel resource consumption remained low. Our benchmarks show a good performance even with a large number of range-mapping and file-cloaking entries in use.

6.1 Future Work

Cloaking is a feature that is useful not just for NFS but for all file systems. We plan on moving our cloaking code to the Virtual File System (VFS) [6, 15]. This way, cloaking features could be used with other native file systems such as EXT2FS on local hosts, or with stackable file systems [3, 20, 27, 28, 30], as well as NFS and other network-based file systems.

We plan to explore methods to improve the performance of cloaking. To improve cloaking performance, we have to allow multi-user NFS clients to cache files. However, the VFS's directory cache is not aware of which users are looking up entries in the cache. If one user lists a directory, all the files in that directory are cached and can be seen by another user that can list the same directory. For cloaking to work with caching, we need to support *user views* of the directory cache. That is, each entry in the client-side cache should have enough information about cloaking to determine how to present the contents of a directory to a user based on their immediate credentials, filtering files that they should not see or access. To achieve this, it would be important to change the directory cache and the VFS in a way that does not change existing NFS protocols.

An alternative to changes to the directory cache is to implement cloaking functionality in both the NFS server and the client. However, changing many existing NFS clients is impractical and changing NFS protocols in use is nearly impossible. A more suitable platform for such changes would be NFSv4 as we discussed in Section 5. NFSv4 standardizes the mechanisms to extend the protocol and provides callback methods for servers to initiate contact with clients. In this way we can allow clients to cache directories that were filtered due to cloaking, and force new client-side users to contact the server to get an updated list of files for a directory, based on a different view of that directory for the new user.

7 Acknowledgments

We thank Nenad Dedic for contributing to the early range-mapping and file-cloaking code, for Charles Wright for his comments while the paper was being written, and for R. Sekar for his suggestion to add a `no_client_cache` export option. Thanks go to the anonymous Usenix reviewers, and to Chris Demetriou for his helpful comments. This work was partially made possible by an HP/Intel gift number 87128.

The work described in this paper is Open Source Software and is available for download from `ftp://ftp.fsl.cs.sunysb.edu/pub/enf`.

References

[1] S. M. Bellovin and M. Merritt. Limitations of the Kerberos authentication system. In *Proceedings of the Winter USENIX Technical Conference*, pages 253–67, Winter 1991.

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

[3] J. S. Heidemann and G. J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[4] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[5] O. Kirch. The Linux user-space NFS server. ftp://linux.mathematik.tu-darmstadt.de/pub/linux/people/okir, December 1997.

[6] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer USENIX Technical Conference*, pages 238–47, Summer 1986.

[7] Legato Systems, Inc. NHFSSTONE – Network File System benchmark program. ftp://wuarchive.wustl.edu/languages/c/unix-c/benchmarks/nhfsstone.tar.Z, July 1989.

[8] H. J. Lu. Linux NFS utility package. http://nfs.sourceforge.net, February 2001.

[9] S. Lunt. Experiences with Kerberos. In *Proceedings of the Second USENIX Security Workshop*, pages 113–20, August 1990.

[10] Network Appliance, Inc. NetApp files. http://www.netapp.com, 2002.

[11] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer USENIX Technical Conference*, pages 247–56, Anaheim, CA, Summer 1990. USENIX.

[12] 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.

[13] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.0.4 edition, February 2000. http://www.am-utils.org.

[14] 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.

[15] D. S. H. Rosenthal. Evolving the Vnode interface. In *Proceedings of the Summer USENIX Technical Conference*, pages 107–18, Summer 1990.

[16] R. Sandberg. The Sun network file system: Design, implementation and experience. Technical report, Sun Microsystems, 1985.

[17] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proceedings of the Summer USENIX Technical Conference*, pages 119–30, Summer 1985.

[18] B. Shein, M. Callahan, and P. Woodbury. NFSSTONE: A network file server performance benchmark. In *Proceedings of the Summer USENIX Technical Conference*, pages 269–275, Baltimore, MD, Summer 1989.

[19] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS Version 4 Protocol. Technical Report RFC 3010, Network Working Group, December 2000.

[20] G. C. Skinner and T. K. Wong. "Stacking" Vnodes: A progress report. In *Proceedings of the Summer USENIX Technical Conference*, pages 161–74, June 1993.

[21] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter USENIX Technical Conference*, pages 191–202, Winter 1988.

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

[23] The Standard Performance Evaluation Corporation. SPEC SFS97 (2.0) benchmark. http://www.spec.org/osg/sfs97, June 2001.

[24] 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.

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

[26] E. Zadok. *Linux NFS and Automounter Administration*, chapter 6: NFS Version 4, pages 151–180. Sybex, Inc., May 2001.

[27] E. Zadok and I. Bădulescu. A stackable file system interface for Linux. In *LinuxExpo Conference Proceedings*, pages 141–151, May 1999.

[28] E. Zadok, I. Bădulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, June 1999.

[29] E. Zadok and A. Dupuy. HLFSD: Delivering Email to your $HOME. In *Proceedings of the Seventh USENIX Systems Administration Conference (LISA VII)*, pages 243–254, Monterey, CA, November 1993.

[30] E. Zadok and J. Nieh. FiST: A language for stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.