Access Control Lists for the Self-Certifying Filesystem

by

George Savvides

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2002

Author .....
Department of Electrical Engineering and Computer Science
August 20, 2002

Certified by .....
M. Frans Kaashoek
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
David Mazières
Assistant Professor of Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Access Control Lists for the Self-Certifying Filesystem

by

George Savvides

Submitted to the Department of Electrical Engineering and Computer Science
on August 20, 2002, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

The Self-certifying File System (SFS) currently exports Unix filesystems. Consequently, file owners on SFS servers who want to give other users access to their files can do so only through the coarse-grained Unix access control mechanisms, which are based on locally defined user identifiers and group identifiers. Therefore, despite the fact that SFS was designed to be a *global* filesystem, it is impossible for a file owner to grant access permissions to a remote SFS user who does not maintain a Unix account on the local machine. Moreover, creating new user identifiers or group identifiers for the purpose of access control requires the involvement of the local machine's administrator, a limitation that runs counter to the egalitarian spirit of SFS.

This thesis describes the design and implementation of an alternative access control mechanism, which is based on Access Control Lists (ACLs). This mechanism affords SFS users much more flexibility in managing who and how can access their files, and does not require the assistance of the local realm's system administrator. By allowing the use of public key hashes as identifiers of remote users, ACLs allow access control to extend beyond the local machine's realm to *all* SFS users, in line with the spirit of SFS. Furthermore, in the future, our ACL mechanism could easily be extended to support access control for groups of users, such as "all MIT students", whose composition is defined and maintained by a third party on any SFS server.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: David Mazières
Title: Assistant Professor of Computer Science

Acknowledgments

I would like to thank David Mazières and Frans Kaashoek for their immense help and guidance in designing, implementing, and refining the SFS ACL mechanism. I would also like to express my gratitude to the members of the PDOS group for their help with various aspects of SFS, especially Frank Dabek, Eric Peterson, Chuck Blake and Michael Kaminsky.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

SFS [1, 2, 3, 4] is a secure and decentralized network file system designed to span the Internet. It provides a common namespace for all files in the world. Users can access their files from any machine they trust, anywhere in the world. They can share files across organizational boundaries by merely exchanging file names. Like the web, anyone can set up an SFS server, any client can access any server, and any server can link or point to any other server. Unlike the (mostly insecure) web, though, SFS provides a mechanism based on self-certifying pathnames which allows clients and servers to communicate *securely* over an insecure network such as the Internet. Several cryptographic protocols are used to ensure file system security. One of the novelties of SFS is that setting up secure communications does not require the intervention of network administrators, or the involvement of a certification authority, such as Verisign.

1.1 Relation of SFS to NFS

In order to ease its adoption and ensure portability, SFS was built on top of the widely used Network File System (NFS)[24]. An SFS client requires the presence of an NFS client, to which it "pretends" to be an NFS server. Likewise, an SFS server requires the presence of an NFS server, to which it acts as an NFS client. On the client

side, the SFS client receives the local NFS client's requests and sends them encrypted over the network to the remote SFS server. There, the SFS server decrypts them and acting as an NFS client, relays them to the local NFS server. The interaction is depicted in Figure 1-1. The reply takes the same (secure) path in reverse, and ends up at the NFS client that issued the request.

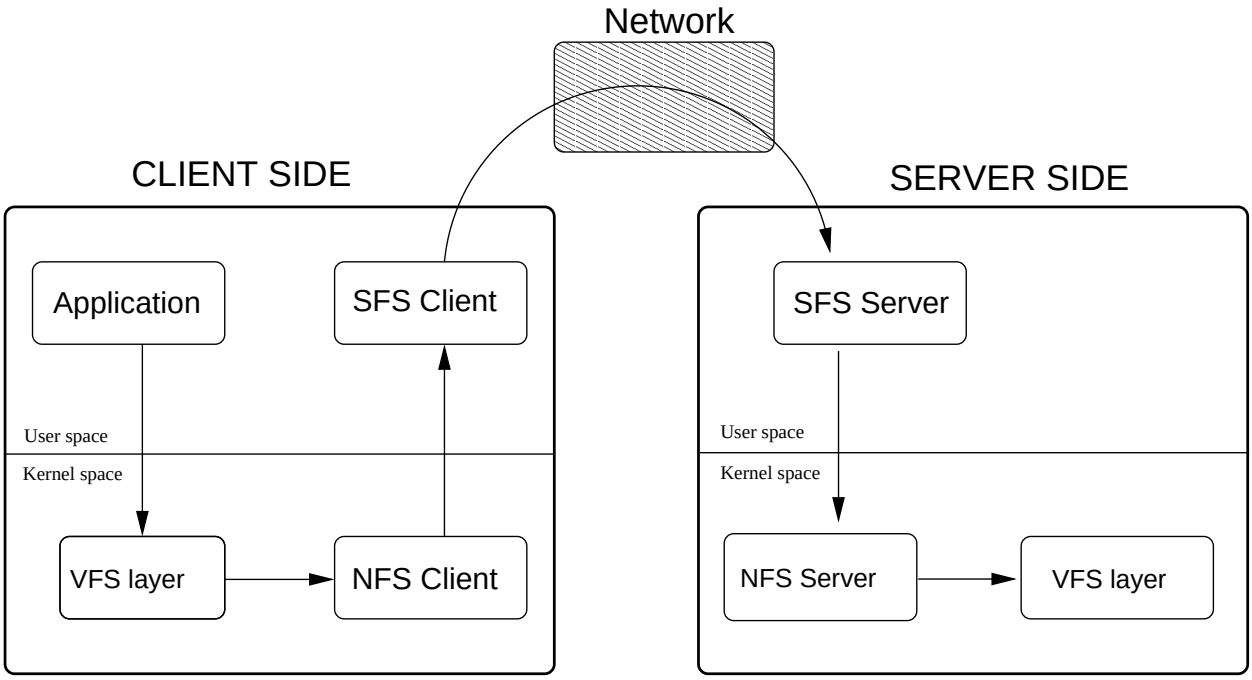

Figure 1-1: The path taken by a filesystem request originating from an application on the client. The response takes the same path in reverse.

1.2 Current access control mechanism and its limitations

SFS effectively delegates access control to the underlying NFS server. Essentially, remote SFS users get whatever file permissions they would get if they were to log into the machine. Moreover, anonymous users who would not be able to log into the machine at all are (optionally) allowed some restricted level of access to the filesystem (with "nobody" credentials). Depending on the server's configuration, anonymous

users may be restricted to accessing only certain parts of the file system.

1.2.1 Credentials mapping

When a remote user logs into an SFS server, the server generates an authentication request and sends it to the user. The user's agent signs the request, and appends a copy of the user's public key. The server checks the signature, thereby verifying that the user indeed has possession of the corresponding private key. If authentication succeeds, SFS searches a database mapping public keys to local Unix usernames. If a match is found, SFS obtains the corresponding Unix credentials from the local `etc/passwd` file, and associates them with the remote user.

If any of the above steps fails, the remote user's agent has to restart the process, requesting anonymous login. In this case, the remote user receives "nobody" credentials.

1.2.2 NFS Access Control

In order to delegate access control to NFS, the SFS server tags the RPCs it sends to the local NFS server with the remote user's (local) Unix credentials that were obtained at login time. By doing so, it "tricks" the NFS server into thinking that the remote user is issuing the requests directly. Whether these requests will be honored or not is decided by the NFS server, based entirely on the remote user's credentials on the local machine.

Consequently, just like in Unix, directories and files in SFS are associated with three sets of permissions, for three different types of users: *owner*, *group* and *other*. For each of these three types, the access permissions consist of 3 bits (r, w, x) corresponding to the permission to read/write/execute a file (or list/modify/traverse a directory). The owner of the filesystem object can change the access permissions with the `chown` call.

1.2.3 Limitations

The fact that only local Unix credentials are used for access control severely restricts a file owner's flexibility in determining who and how can access her files. More specifically:

1. For each file, one can only specify 3 different sets of access permissions, corresponding to the 3 types of user (*owner*, *group* and *other*.) It is impossible to assign different access permissions to different users or different groups.

2. Even though SFS was designed to be a global file system, it is impossible to grant access privileges to an SFS user who does not happen to have an account in the local administrative realm. This restriction results from the fact that all such users have to log in as "anonymous" and can therefore only get whatever permissions correspond to *other*.

3. Registering a new user in the local administrative realm or creating a new group of users for the purpose of access control requires the assistance of the realm's system administrator, a limitation which conflicts with the egalitarian nature of SFS.

4. It is impossible to give access rights to variable groups such as "all MIT undergraduates" whose composition is managed and regularly updated by a third party, which is only trusted to provide the correct composition of the group and is not given any administrative privileges on the server.

1.3 Access Control Lists

In order to address the limitations mentioned above, we designed and implemented an access control mechanism that relies on Access Control Lists (ACLs) instead of Unix credentials. This mechanism has the following properties:

1. ACLs allow fine-grained access control, giving file owners the flexibility to assign different access permissions to different users or groups.

2. ACLs allow access control to extend to *any* SFS user, whether she has an account in the local administrative realm or not.

3. Maintaining ACLs does not require the involvement of the local realm's system administrator.

4. ACLs are compatible with future extensions that will provide support for groups that are defined and maintained by a third party on any SFS server.

1.3.1 Extending access control to all SFS users

For authentication purposes, every SFS user is currently required to have a public/private key pair. Since with overwhelming probability each user's public key is *unique*, public keys can effectively be used to establish a global user namespace. By allowing the use of public keys as global user identifiers, the ACL mechanism extends access control beyond the local NFS realm to all SFS users.

For practicality, only the (much shorter) SHA-1 hash [5] of a public key appears in the ACL. This optimization does not impair security: even though in theory two different public keys could produce the same SHA-1 hash, such an event is extremely unlikely to occur in practice since SHA-1 is considered to be a cryptographically secure, collision-resistant hash function [6, 7]. Indeed, despite extensive research, no SHA-1 collisions have been found to date.

1.3.2 ACL example

Each file and each directory in SFS has an associated ACL similar to the one below:

```
ACLBEGIN
sys:anyuser:rl:
alias:savvides:rldiwa:
alias:dm:rldiwa:
localgroup:sfsdev:rliw:
pk:7aa6rh3dsv8sxpddkh4emb4ie75a7adr:rldiwa:
pk:ibuqjvxvxvw5wqb8qct42n7c36fzs8m8:rliw:
group:[self-certifying-pathname]:iw:
ACLEND
```

Each ACL line assigns permissions to one of 5 different types of users:

sys: a system group. Currently, only `sys:anyuser` is defined.

alias: a (human-friendly) username in the local NFS realm

localgroup: a (human-friendly) group name in the local administrative realm

pk: a user's (hashed) public key

group: a self-certifying pathname pointing to a text file listing a set of public keys all of which are to be treated as members of a group and given the same access permissions. This file can reside on any SFS server. This feature has not been implemented yet.

The permissions that one can set in each ACL line are reminiscent of those used in AFS[15], and are summarized in Table 1.1. The permissions assigned to the remote user correspond to the logical OR of the permissions found in each matching ACL line.

1.4 Contributions of this thesis

The original ACL design that was presented in the thesis proposal had suggested a pathname-based approach to access control, whereby the target file's pathname

Permission	Effect on files	Effect on directories
r	read contents of file	no effect
w	write to file	no effect
l	no effect	`cd` to a directory, list its files
i	no effect	insert new files/dirs into directory
d	no effect	delete files/dirs from the directory
a	modify the file's ACL	modify the directory's ACL

Table 1.1: Access rights available in ACLs

would be mapped to a set of access permissions. Upon further investigation, this approach proved to be difficult to implement in practice. The main difficulty arose from the fact that NFS requests specify the filesystem object they want to operate on in terms of opaque *filehandles*. Since SFS is running in user mode, mapping an NFS filehandle back to a pathname would require issuing a call to get the attributes of the target object and then, by looking at the i-node field of the attributes, performing a prohibitively expensive exhaustive search on all i-nodes to find the matching pathname.

In order to make it possible to efficiently locate the ACL corresponding to an NFS filehandle, we chose to store the ACLs inside the files themselves, in a 512-byte section at the beginning of the file. Thus, upon receiving an NFS request from a remote client, we can obtain the corresponding ACL by reading the first 512 bytes of the file pointed to by the filehandle contained in the incoming request. A similar, but slightly more complicated procedure is used for obtaining the ACLs of other filesystem objects, such as directories.

The presence of the ACLs inside files must of course be hidden from end users. We used the RPC traversal mechanism that comes with SFS to adjust incoming and outgoing RPCs so that the embedded ACL and its effects, such as a larger file size, are not visible to clients.

Since ACLs on the server are hidden from file manipulation utilities on the client, we also designed and implemented a mechanism for viewing and modifying ACLs from the client without going through the regular filesystem interface.

Finally, we added caching of ACLs and user permissions on the server in order to reduce the overhead incurred by the extra NFS calls that are needed to locate and read ACLs on the server.

1.5 Thesis overview

- Chapter 2 describes related work.
- Chapter 3 describes the design of the ACL mechanism.
- Chapter 4 describes the implementation of the ACL mechanism.
- Chapter 6 gives the conclusion and suggestions for future work.

Chapter 2

Related work

The related work falls into two categories: mechanisms for user authentication and mechanisms for filesystem access control. We discuss each in turn.

2.1 User Authentication Mechanisms

Secure user authentication mechanisms rely on a variety of cryptographic protocols for verifying the identity of a user. While some mechanisms allow *any* user to authenticate himself to any remote server, others allow user authentication only within the same administrative realm, or between administrative realms that have previously been cross-linked by their administrators.

2.1.1 Kerberos

Kerberos [19] is a trusted, third-party authentication service based on symmetric key cryptography. It allows users to authenticate themselves to servers (or, more generally, *services*) on the network and vice versa. It is "trusted" in the sense that the users and the services must trust Kerberos to provide an accurate judgment concerning the identity of the remote party.

A principal (i.e., a user or a service) that requires Kerberos authentication must first register with a central Kerberos database, and create a secret key known only to Kerberos and the principal. In the case of users, the secret key is simply an encrypted password.

User authentication is performed by means of short-lived *tickets* that are issued by a centralized *Ticket Granting Server* (TGS). At the beginning of a session, the user sends his username and receives a ticket, which is encrypted with the user's secret key (which, as mentioned above, is based on his password). By entering his password, the user can decrypt the ticket and, for the period of its validity (approximately 8 hours), use it to authenticate himself to services, which know and trust the TGS that issued it. Such tickets can also be used to authenticate a service to a user, or to set up a secure, encrypted channel between a user and a service.

The centralized approach to user authentication taken by Kerberos makes cross-realm authentication difficult, but not impossible. Registering two administrative realms' Ticket Granting Servers with one another makes it possible for users of one administrative realm to authenticate themselves to services in the other. However, the registration procedure requires the involvement of administrators and scales poorly. For example, registering 100 different administrative realms with one another would require $2\binom{100}{2} = 9,900$ cross-realm registrations (two for each pair of administrative realms).

This scalability issue has been partly addressed in the latest version of Kerberos (Kerberos V5), which supports realm hierarchies [20]. In order to contact a service in another realm, it may be necessary to contact the remote TGS of one or more intermediate realms. The names of each of these realms are recorded in the ticket. While this approach greatly reduces the number of cross-realm registrations that need to be performed in order to allow cross-realm authentication of users, it does not eliminate the need for cross-registration, which, of course, remains at the discretion of each realm's administrators. Going one step further, some proposals [21, 22, 23] have suggested various ways of incorporating public key cryptography into Kerberos,

thereby greatly facilitating cross-realm authentication.

Kerberos and SFS thus share the ambition of providing an infrastructure for establishing secure and authenticated communication across realms. While several filesystems have used Kerberos for their user authentication needs (AFS, Kerberized NFS, DFS), the scalability issues mentioned above make it rather ill-suited for adoption by SFS, which is designed to have a global image, accessible from anywhere in the world. Moreover, Kerberos' authentication infrastructure is based on central, trusted servers that are under the authority of system administrators. These requirements conflict with the decentralized and egalitarian nature of SFS.

2.1.2 Taos

The Taos Operating System [9, 10] provides an elaborate security model designed to meet the security and authentication needs of distributed systems. All entities in this model (users, servers, printers, etc) have a public/private key pair. This allows each entity to write certificates delegating privileges and roles (such as "John" or "member of group X") to other entities.

User authentication is mediated by a novel, logic-based engine for reasoning about certificates and delegation. The engine supports certificate chains, thereby allowing administrators and users to define the precise trust relationship governing the authentication process. The chaining of certificates can also be used for cross-realm authentication.

2.1.3 SDSI

SDSI (Simple Distributed Security Infrastructure) [12, 13] is an egalitarian, decentralized public key naming infrastructure that incorporates a certificate-based approach to delegating authority and defining relations between its principals (public keys).

SDSI certificates can be chained, thereby giving a user the ability to define com-

plex relationships that transcend any notion of administrative realm or namespace. However, dealing with such certificate chains requires complex algorithms for chain discovery, as well as complex proof systems for chain validation [14]. To date, no truly distributed chain discovery algorithms have been discovered, and consequently, no truly distributed implementation of SDSI exists.

In the context of user authentication, SDSI certificates can be used to prove that a remote user has access to a certain protected resource on the server. The remote user needs to present a certificate chain proving that the owner of the resource directly or indirectly grants him the desired level of access on that resource.

2.2 Filesystem Access Control Mechanisms

Access Control Lists (ACLs) are widely used for controlling access to filesystem objects. However, the capabilities of ACL-based access control vary widely from filesystem to filesystem.

2.2.1 NFS

NFS [24] currently lacks support for Access Control Lists. Some vendors, such as Solaris and Irix, have created ancillary protocols to NFS to extend the server's ACL mechanism across the network. Even though the server still interprets the ACL and has final control over access to a file system object, the client is able to manipulate the ACL via these additional protocols.

Generally these efforts have focused on homogeneous operating environments [17]. Compatibility and interoperability issues were not addressed. Each model defines its own variants of entry types, identifies users and groups differently, provides different kinds of permission bits, and describes different procedures for ACL creation, defaults, and evaluation.

The next version of NFS will probably not suffer from these limitations: ACL support

is one of the requirements for NFSv4 proposed by the Internet Engineering Task Force (IETF) [18].

2.2.2 AFS

In AFS [15], access to the filesystem is controlled by Access Control Lists (ACLs). Each directory has its own ACL, which also governs access to all files inside the directory. Users "own" their home directory and all its subdirectories. A directory's owner can specify which users and groups appear in the ACL and what privileges each of them has. Table 2.1 summarizes the various types of access permissions that are available in AFS ACLs.

Permission	**Description**
r	permission to read contents of files in the directory
l	permission to lookup (list) contents of the directory
i	permission to insert files into the directory
d	permission to delete files from the directory
w	permission to write files in the directory
k	enables programs to place advisory locks on a file
a	permission to administer the ACL of the directory

Table 2.1: Types of access permissions in AFS ACLs

With the help of the system administrator, each AFS user can define up to 20 private *protection groups*. These groups consist of lists of `UID`s of users who are members of the group, and can be treated in the same way as users when assigning permissions in ACLs. Their use simplifies the administration of work groups. For example, a professor can create a group consisting of the students in a class. A student can then grant the entire class access permissions to a directory he owns, without having to keep up with who is or is not in the class or what their individual `UID`s are.

AFS does not support access control for remote users. In other words, a directory's ACL can only contain users that are registered in the local realm. However, when using Kerberos for user authentication, it is possible to grant access permissions to users in remote administrative realms. Before accessing protected resources in the lo-

cal realm, remote users must first authenticate themselves to the realm. As described in Section 2.1.1, such cross-realm authentication requires prior cross-registration of the realms' Kerberos services by the administrators. Unlike users, groups defined in other administrative realms cannot be used in AFS ACLs.

2.2.3 Echo

Echo is a distributed filesystem, which, similarly to SFS (and unlike AFS and NFS), provides a global namespace where all clients see the same image, regardless of where they are in the network. The namespace has a hierarchical structure similar to that of the Domain Name System (DNS). It is basically a tree of labeled arcs with a single common root, where filenames are paths through the naming tree. The tree is navigated with the help of servers running DNS.

All Echo objects (files, directories, volumes, and servers) are protected by Access Control Lists (ACLs), which specify what operations each principal is allowed to perform on each object. Similarly to AFS, an ACL in Echo is a set of names and access rights. The names in ACLs can be either principals (users) or groups. Unlike AFS, it is possible to have globally-named principals and groups in ACLs.

In contrast to SFS, which does not prescribe a particular user authentication mechanism, user authentication in Echo is achieved through a *built-in* certificate system which is based on Taos's [9, 10] authentication infrastructure. Different levels of the namespace can have different levels of trust, and secure cross-links can be used to bypass untrusted levels.

Even though Echo was designed to be a truly distributed system, it (purposefully) retained the centralized administration features of time-sharing systems. In this sense it is very different from SFS, which strives to avoid unnecessary centralized control by allowing the administrator to be "out of the loop" if users are allowed to make decisions for themselves.

Chapter 3

Design of ACL-based Access Mechanism

The ACL-based access control mechanism for SFS that we designed and implemented is intended as a replacement of the current Unix-style NFS access control mechanism. It gives file owners more fine-grained control over who can access their files, and does not suffer from the limitations of the NFS-based mechanism mentioned in Section 1.2.3.

3.1 Description of ACL structure

ACLs start with the string string `ACLBEGIN` and end with the string `ACLEND`. Every other line specifies a set of access permissions for a user or a group.

3.1.1 Supported types of users and permissions

Table 1.1 summarizes the available permissions and their effects on files and directories.

A file owner can give another SFS user, or group of users, any subset of the permissions

that appear in Table 1.1. Each ACL entry ends in a newline, and each line is of the format

`entry_type:entry_description:entry_permissions:`

The various ACL entry types are presented below.

Public key: The entry specifies the user through the SHA-1 hash of her public key. For example:

`pk:7aa6rh3dsv8sxpddkh4emb4ie75a7adr:di:`

Alias: The SFS user is specified through his username in the local NFS realm (if available, of course). For example:

`alias:savvides:rldiwa:`

Localgroup: Describes a locally-defined group. All SFS users who have Unix credentials in the local NFS realm and belong to the group will get the corresponding permissions. For example:

`localgroup:sfsdev:rliw:`

System group: The only system group that is currently supported is `sys:anyuser`, which represents all SFS users. For example, the following gives all users read and list access:

`sys:anyuser:rl:`

Group: A group is an SFS self-certifying pathname, such as

`/sfs/ny.lcs.mit.edu:tqqzicz2qsik3k57s2mfnupcdq84xerz/sfs_users`

pointing to a plaintext file. Each line of the file contains either an entry of type `pk` or, recursively, another group (SFS pathname). Notice that both types of entry are independent of the local namespace of the server on which they appear. Example:

`group:[self-certifying-pathname]:rl:`

The colon (":") that appears inside self-certifying pathnames will *not* be interpreted as a component delimiter.

3.1.2 Example of an ACL

```
ACLBEGIN
sys:anyuser:rl:
localgroup:sfsdev:rlwi:
pk:7aa6rh3dsv8sxpddkh4emb4ie75a7adr:wdia:
alias:savvides:wi:
alias:alice:wid:
alias:steve:lrwdia
group:/sfs/ny.lcs.mit.edu:tqqzicz2qsik3k57s2mfnupcdq84xerz/sfs_users:iw:
ACLEND
```

The set of permissions each user gets corresponds to the union (logical OR) of the permissions on each matching line. In the example above, user `alice` is given (w, i, d) directly, and (r, l) as a member of `sys:anyuser`.

Negative permissions are not allowed. That is, one ACL line one cannot take away from a user permissions that were granted to him elsewhere in the ACL. The absence of negative permissions makes both the implementation and the the use of ACLs more straightforward. It also makes it more difficult for someone to find "loopholes" in the ACLs and exploit them to gain access against the file owner's wishes. For example, one might notice that he is given read access to a file as a member of group A, but prevented from reading the file as a member of group B. If he manages to temporarily remove himself from group B, he will get access to the file, something which is presumably contrary to the file owner's original intentions.

3.2 Initializing ACLs

A newly-created file or directory automatically inherits the ACL of the parent directory. A consequence of this policy is that including file-specific permissions, such as (w)rite, in a directory's ACL is meaningful, since these permissions will be inherited by default by all newly-created files in that directory.

The root of the exported directory must somehow have an ACL associated with it. Since it was not created by SFS, and thus has no parent directory in SFS, it does not inherit an ACL at the time of its creation. One thus needs to initialize its ACL manually (through the regular Unix interface) when initially setting up the SFS server. This initialization can be done by simply storing the ACL in a file called `.SFSACL` residing inside the root directory. (See Section 4.2.)

3.3 Viewing ACLs

The command `viewacl target` issued from the client displays the ACL associated with `target` where `target` is the (relative) pathname to a file or directory. In order to view the ACL associated with the current directory, one can set `target` to "." .

Any user can view the ACL associated with a file/directory provided that he is allowed to list the contents of the parent directory (in other words, provided he can obtain the object's NFS *filehandle*, which is required in order to place the request).

3.4 Modifying ACLs

In order to change an ACL, the remote user must have administer permission for the object (file or directory) whose ACL she wants to modify. The command

`setacl target acl_file`

can then be used to set the ACL of `target` to the contents of the file `acl_file`.

A convenient way to modify an ACL is to pipe the output of `viewacl` to a file, edit the contents of the file to reflect the desired ACL contents, and then use `setacl` to replace the ACL with the contents of the file.

Obtaining the SHA-1 hash of a user's public key

The command `sfskey gethash pk_string` can be used to obtain the SHA-1 hash of a user's public key when the public key is known. Note that users' public keys are typically stored in `/var/sfs/authdb`.

Chapter 4

Implementation

The ACL-based access control mechanism fully replaces the standard NFS-based mechanism. The SFS read/write server is now responsible for deciding whether a request should go through, or whether an "access denied" error should be returned instead. As far as the NFS server is concerned, all the files that are served by SFS are owned by a single Unix user, `sfs-owner`, who has all access permissions. When installing SFS, `sfs-owner` must be created as a new user on the machine.

4.1 User login policy

In order to support public-key based access control, the login policy had to be modified so that the SFS server does not reject users who do not have an account on the local machine.

When a remote user authenticates himself to the SFS server, SFS assigns him an *authentication number* which maps to the following set of credentials:

```
string pkhash<>;
string username<>;
string homedir<>;
string shell<>;
unsigned uid;
unsigned gid;
unsigned groups<>;
```

The credentials include the SHA-1 hash of the user's public key, as well as the user's local Unix credentials, if available. The Unix credentials are stored in order to allow the use of human-friendly local usernames and group ids in ACLs.

4.2 ACL storage

4.2.1 Files

SFS stores the ACLs of files inside the files, in a fixed 512-byte section at the beginning of the file. This way, SFS can obtain a file's ACL by simply issuing a read request for the first 512 bytes of the file specified by the filehandle which is contained in the request sent by the remote user.

Of course, the fact that every file now contains a fixed-size ACL must be hidden from the client, so as not to disrupt regular file operations. The RPC traversal mechanism that comes with SFS is immensely helpful in achieving this in a straightforward way. By adjusting the *offset* of incoming `read` and `write` requests and the *size* field in the attributes returned by the local NFS server, no difference is visible to the NFS client on the other side.

4.2.2 Directories

The approach above will not work for directories, since a directory can only contain pointers to other filesystem objects. Instead, the ACL corresponding to a directory `dir` is stored in a file inside the directory called `dir/.SFSACL`. When a request involving a directory arrives at the SFS read/write server, the server can read the directory's ACL by first issuing a `lookup` request to the NFS server to get the filehandle corresponding to `.SFSACL`, and then by proceeding as in the case of files.

4.2.3 Other types of object

NF3LNK: One cannot store an ACL inside a link. However, there is no need to store an ACL since the policy regarding links is to allow any user to read the contents of a link (provided that he is allowed to obtain the filehandle of the link.)

NF3FIFO: Implemented on the server as a file. At the time of creation, objects of type NF3FIFO are tagged with a different group-id, so that the attributes returned to the client can later be adjusted to identify the object as NF3FIFO instead of NF3REG (file).

NF3SOCK: Same approach as for NF3FIFO.

NF3BLK: The creation of such objects is not supported.

NF3CHR: The creation of such objects is not supported.

4.3 Access Control Algorithm

The read/write server determines whether an incoming NFS request should go through based on the target object's ACL and the credentials of the remote user who issued the request. These credentials are obtained at login time, as described in Section 4.1.

If the remote user has adequate permissions for the requested operation, the read/write server forwards the request to the local NFS server, tagging the RPC with the credentials of `sfs-owner`. This guarantees that the operation will succeed unless there's an NFS error unrelated to access permissions. If instead the read/write server decides to reject the call, an `NFS3ERR_ACCES` error is returned.

4.3.1 Getting the target object's ACL

The read/write server needs to determine the filehandle where the target object's ACL resides, and then issue a `read` request to the NFS server to get its contents. How exactly this is done depends on the type of NFS call. For `read` and `write` calls, for example, the NFS request includes the filehandle of the file to be read (resp. written), and hence we already know the filehandle of the ACL. For calls involving directories (such as `readdir`), where we know the file handle of the directory, we must first issue a `lookup` request to determine the filehandle of `.SFSACL` in that directory. For calls where the target is of unknown type (e.g. `setattr`), we first issue a `getattr` call, and then determine from the attributes whether we should treat the object as a file or as a directory.

The above calls, including the final `read` call to get the contents of the ACL can all be done in parallel with other operations.

4.3.2 Determining the user's permissions

When we start reading the ACL, the user starts with no permissions. For each remaining line, we check whether it contains any permissions that the user does not already have. If so, we check whether the user appears on the line either directly or indirectly (as member of a group). In case of a match, the permissions that appear on the line are added to the permissions that the user has already accumulated. At the end, the user's permissions will correspond to the logical OR of the permissions on all matching lines.

4.3.3 Determining if the permissions suffice

In order for the read/write server to accept the remote NFS request, the user should have the necessary permissions for that type of call. The policy on which this is based is described in Table 4.1

NFS Call	**Permissions required**
NFSPROC3_NULL	always allowed
NFSPROC3_GETATTR	always allowed
NFSPROC3_SETATTR	see note below
NFSPROC3_LOOKUP	list
NFSPROC3_ACCESS	always allowed
NFSPROC3_READLINK	always allowed
NFSPROC3_READ	read
NFSPROC3_WRITE	write
NFSPROC3_CREATE	insert
NFSPROC3_MKDIR	insert
NFSPROC3_SYMLINK	insert
NFSPROC3_MKNOD	insert
NFSPROC3_REMOVE	delete or administer
NFSPROC3_RMDIR	delete or administer
NFSPROC3_RENAME	delete or administer [from], insert [to]
NFSPROC3_LINK	insert
NFSPROC3_READDIR	list
NFSPROC3_READDIRPLUS	list
NFSPROC3_FSSTAT	always allowed
NFSPROC3_FSINFO	always allowed
NFSPROC3_PATHCONF	always allowed
NFSPROC3_COMMIT	always allowed

Table 4.1: Required access permissions per NFS call

Clarifications:

- For `NFSPROC3_SETATTR` read/write permission suffices to set `atime/mtime` to server time, but administer permission is required to set to client time or to change the e(x)ecutable bit for an inode.

- For `NFSPROC3_RENAME` it suffices to have delete permission in the object's current directory, and insert permission in the target directory.
- Administer permission also implies delete permission. This is useful in some scenarios, such as for maintaining `/tmp` or bit-bucket directories.

4.4 Caching ACLs and user permissions

Since most NFS calls depend on the user having permission to carry them, reading the ACL every time requires at least one extra call to the local NFS server for every incoming call from the client. This more than doubles the load of the NFS server and can impact performance. The server caches ACLs, thereby reducing this extra overhead.

The permissions for each user-ACL combination are also cached, with the key being a composite hash of the user's public key and the contents of the ACL. Even though this extra level of caching does not lead to any significant performance increase now, the advantage of caching permissions will become apparent when groups are supported, since in that case determining whether a user is a member of a remotely-maintained group will need time which is at least equal to the latency between the SFS server and the remote server hosting the group definition.

It is reasonable to expect that in a typical scenario, the number of different ACLs corresponding to a file owner's files will be much smaller than the number of files. This means that the permissions cache can be efficient since the number of ACL-user combinations will be relatively small.

4.5 Maintaining compatibility with NFS

The SFS server adjusts both incoming requests and outgoing replies in order to hide the existence of the ACLs and, where applicable, approximate NFS access seman-

tics. The RPC traversal mechanism that comes with SFS is immensely helpful in performing these adjustments in a simple, straightforward, and robust way.

4.5.1 Adjusting incoming requests

Before (authorized) incoming requests are forwarded from the read/write server to the NFS server, the following changes are made:

read offset: It is increased by 512 bytes to account for the presence of the ACL

file size: It is similarly increased by 512 bytes (occurs in `setattr` requests only).

setattr requests: The various fields are affected according to the following rules:

- Setting `uid`, `gid` is prohibited in all cases.
- Setting modebits is prohibited, except for the e(x)ecutable bit in files, provided that the user has administer permission.
- Setting the size of a file is allowed only if the user has write permission.
- Setting the `atime/mtime` fields to server time requires read/write permission.
- Setting the `atime/mtime` fields to client time requires administer permission.

4.5.2 Adjusting outgoing requests

The following changes are made before sending back replies from the NFS server:

file size: It is decreased by 512 bytes, to hide the existence of the ACL

access permissions: when the remote NFS server issues an `access` call, the returned access bits must approximate the permissions described in the ACL. How the SFS permissions are translated into NFS permissions is described in

Table 4.2. After the conversion, only the bits that the client requested for are sent back. Doing otherwise would violate the NFS protocol.

NFS permission	**SFS (files)**	**SFS (directories)**
ACCESS3_READ	read	list
ACCESS3_MODIFY	write	insert or delete
ACCESS3_EXTEND	write	insert
ACCESS3_EXECUTE	read	never set
ACCESS3_LOOKUP	never set	list
ACCESS3_DELETE	never set	delete

Table 4.2: Approximating NFS permissions for files and directories based on SFS permissions

mode bits: The mode bits that a user on the client sees by doing `ls -l` do not determine the actual permissions that he has. However, it is helpful to set these bits to something that approximates those permissions. One reason for this is that some applications issue warnings based on those permissions. For example, Emacs will complain that "the file is write-protected" unless the user has the *w*-bit on in the file's mode. Since being "owner", "group" or "other" no longer makes sense on the client side, the mode bits for these three types of user are set to the same value. See Table 4.3.

Modebit	**SFS (files)**	**SFS (directories)**
r	read	list
w	write	insert or delete
x	same as in inode	list

Table 4.3: Approximating mode bits for files and directories based on SFS permissions

It should be noted that since access permissions and mode bits are only advisory in nature, when there is not an exact correspondence between the SFS and NFS/Unix permissions, the more liberal approach is taken (e.g. the NFS permission `ACCESS_MODIFY` is set to true if the user has permission to either insert or delete and not necessarily both). Doing otherwise might cause the client to refuse to issue requests that it thinks it doesn't have permissions for, even though this may not be the case. This decision

does not affect security in any way: whether the request will go through at the server will be decided independently, based on the SFS permissions.

4.6 Viewing and modifying ACLs from the client

The only changes that need to be made on the client side relate to the ability of remote users to view and modify ACLs on the server. Since the ACL mechanism is transparent to clients, a "backdoor" had to be used to access the (hidden) ACLs on the server.

In order to achieve this, we extended the NFS protocol to support two new calls, `NFSPROC3_GETACL` and `NFSPROC3_SETACL`, in addition to the standard NFS calls that are listed in Table 4.1. The access policy governing these two new calls is described in Table 4.4.

NFS Call	**Permissions required**
NFSPROC3_GETACL	always allowed
NFSPROC3_SETACL	administer

Table 4.4: Required access permissions for the new NFS calls

On the client side, two command-line programs, `viewacl` and `setacl` are responsible for generating the appropriate calls to view/set the ACLs on the SFS server. Just like regular NFS calls, these calls are tagged by the SFS client software with an *agent ID* which is derived from the process's user ID and group list.

On the server side, the calls are received by the read/write server which, as usual, looks at the credentials of the remote user who issued the request and the ACL of the target object, and checks whether the user's permissions suffice for the requested operation (see Table 4.4). If so, the server carries out the requested operation by issuing read/write NFS calls to the local NFS server that read or modify the first 512 bytes of the target files. The outcome of the operation is then relayed back to the client.

The client-side programs (`viewacl`, `setacl`) communicate with the read/write server on the SFS server through an RPC program called `SFSCTL_PROG`. The interaction is depicted in Figure 4-1.

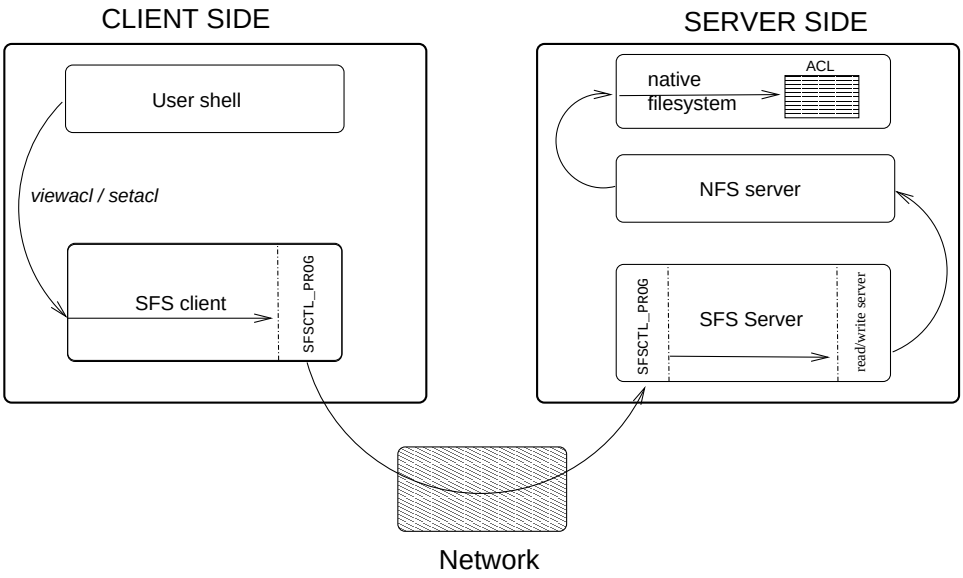

Figure 4-1: The path taken by a user request to view or set the ACL of a file on a remote server. The response takes the same path in reverse.

Chapter 5

Performance

The ACL mechanism introduces a penalty in the overall performance of the SFS server. This penalty is mainly due to the extra NFS requests that the read/write server needs to issue in order to locate and read (or write) the ACLs associated with the incoming requests from the client.

We used the Sprite LFS small file benchmark [26] to measure the performance penalty associated with our ACL mechanism. The methodology and the results we obtained are presented below.

5.1 Methodology

We measured file system performance between two machines running FreeBSD 4.5 and connected by 100 Mbit/sec switched Ethernet. The server machine had a dual Athlon 1 GHz processor, 1 Gb of memory, and an 11 Gb Seagate SCSI hard drive. The client machine had an Athlon 1.3 GHz processor, 768 Mb of memory, and an IBM 10 Gb SCSI hard drive.

We used the Sprite LFS small file benchmark, which creates, reads, and deletes $1,000$ 1024-byte files. We set the cache size to a large enough value $(2,000)$ in order to make sure that cached ACLs will not have been flushed out of the cache by the time

they are needed again. By ensuring that the cache does not overflow during the test, we can better understand how the performance penalty relates to the extra NFS calls that are needed for access control (the cache leads to a predictable reduction in NFS calls).

Using the same test, we measured the performance of the original SFS, our ACL SFS, and our ACL SFS with the cache turned off.

5.2 Results

Table 5.1 shows the results of running the benchmark on the three versions of the SFS software. The results correspond to the average of multiple rounds. The percentages reported are relative to the performance of the original (unmodified) SFS server.

Operation	Original SFS (files/sec)	ACL SFS w/ cache (files/sec)	ACL SFS w/o cache (files/sec)
create	59	40 (68%)	41 (70%)
read	206	120 (58%)	87 (42%)
delete	95	87 (92%)	72 (76%)

Table 5.1: LFS small file benchmark, with 1,000 files created, read, and deleted. The percentages are relative to the performance of the original SFS. The cache refers to both the ACL cache and the user permissions cache.

In the create phase, the performance penalty is mainly due to the extra NFS requests that are needed in order to write the ACL of each newly created file. As expected, the performance is about the same with or without the ACL cache.

On the other hand, the ACL cache greatly improves performance in the delete phase, since it allows the read/write server to avoid most of the extra NFS calls to locate and read the corresponding ACL.

In the read phase, the ACL cache does not seem to improve performance as much as one would have expected. This discrepancy can be explained by taking into account the access pattern imposed by the benchmark: initially, the ACL cache does not

contain any entries, and the files are subsequently read only once. Given this pattern of access, one could only expect a moderate increase in performance when using the cache. To be more precise:

In the read phase, each `read` request is preceded by a `lookup` and an `access` request. In ACL SFS, each `lookup` request generates an extra NFS call to obtain the current directory's ACL. Similarly, each `access` request generates two extra NFS calls in order to find out whether the file object pointed to is a file or a directory, and then read its ACL. If ACLs are cached, in the subsequent `read` call the SFS read/write server can avoid issuing the extra `read` request to get the target file's ACL. Likewise, the read/write server can avoid the `read` request to get the current directory's ACL in future `lookup` calls.

Table 5.2 below presents the effective cost (as # of disk operations) of reading a file during the read phase of the benchmark. The predicted relative performance agrees with the measurements shown in Table 5.1 to within 2%.

NFS request	Original SFS (disk operations)	ACL SFS w/ cache (disk operations)	ACL SFS w/o cache (disk operations)
lookup	1	1	2
access	1	3	3
read	1	1	2
Total	3	5	7
Predicted performance	100 %	60 %	43 %

Table 5.2: Effective cost (as # of disk operations) of reading a file during the read phase of the Sprite LFS small file benchmark.

Chapter 6

Concluding remarks

This thesis has demonstrated that the existing SFS infrastructure can be used to create a powerful, yet easy-to-use access control mechanism, which is well suited to a global, decentralized and egalitarian filesystem like SFS. We showed that by embedding ACLs inside the filenames, it becomes possible for a user-space program such as SFS to perform access control. Moreover, by having users' public keys double as user identifiers, we created a global user namespace that allows access control to extend to all SFS users without requiring the assistance of system administrators.

Below, we present some implementation-related limitations of our current ACL mechanisms, as well as suggestions for future work.

6.1 Limitations of ACLs

6.1.1 ACL length restriction

Because ACLs for files are stored in a fixed section of 512 bytes at the beginning of the file, the length of an ACL cannot exceed 512 bytes. In the future, this limitation can be eliminated by allowing indirect references to local files where the ACL can be continued, in a way vaguely reminiscent of i-node storage.

6.1.2 Unsupported types of NFS objects

Only files and directories can have ACLs associated with them. NFS objects of type NF3FIFO and NF3SOCK are implemented in terms of files, while the creation of objects of type NF3BLK and NF3CHR is not allowed. Links (NF3LNK) do not have ACLs associated with them, and all users are allowed to read the path that they point to.

6.1.3 Support for groups

The current version does not allow a user to give a self-certifying pathname pointing to a group, that is, a text file containing a list of public key hashes and (recursively) other groups.

6.2 Suggestions for future work

Adding support for groups is probably the most important improvement that needs to be made to the current version of ACL support. The present version lacks such support because ACL searching is done synchronously and so performance could be adversely affected by the latency of communication with other servers or the non-availability of a remote server. There is also the potential of deadlock (e.g. Alice can read Bob's group definitions if and only if Bob can read Alice's definitions). In the future, the task of testing membership in remote servers could be done asynchronously by a dedicated, asynchronous agent.

Bibliography

[1] David Mazières, "SFS 0.6 Manual", (`www.fs.net`)

[2] David Mazières, M. Frans Kaashoek, "Escaping the Evils of Centralized Control with self-certifying pathnames", *Proceedings of the 8th ACM SIGOPS European workshop: Support for composing distributed applications*, (Sintra, Portugal, September 1998)

[3] David Mazières, "Self-certifying File System", *Ph.D Thesis*, (Massachusetts Institute of Technology, May 2000)

[4] David Mazières, Michael Kaminsky, M. Frans Kaashoek, Emmett Witchel, "Separating key management from file system security", *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, (Kiawah Island, South Carolina, December 1999)

[5] National Institute of Standards and Technology (NIST), "Secure Hash Standard", *FIPS Publication 180*, (1993)

[6] B. Preneel, "Analysis and Design of Cryptographic Hash Functions", *Ph.D. Thesis*, (Katholieke University Leuven, 1993)

[7] M.J.B. Robshaw, "MD2, MD4, MD5, SHA and Other Hash Functions", *Technical Report TR-101, version 4.0*, (RSA Laboratories, 1995)

[8] Andrew Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart, "The Echo distributed file system", *Research Report 111*, Compaq Systems Research Center, Palo Alto, CA (September 1993).

[9] Edward Wobber, Martín Abadi, Mike Burrows, and Butler Lampson, "Authentication in the Taos Operating System", *ACM Transactions on Computer Systems, Vol. 12, No.1*, (February 1994)

[10] Andrew Birrell, Butler Lampson, Roger Needham and Michael Schroeder, "A Global Authentication Service Without Global Trust" (System Research Center, Digital Equipment Corporation, 1986)

[11] Matthew Fredette, "The SDSI 2.0 Library and Tools, Release 0.4.5", (MIT Laboratory for Computer Science, 9 July 1999)

[12] Matthew Fredette, "An Implementation of SDSI - The Simple Distributed Security Infrastructure", *Master's Thesis*, (Massachusetts Institute of Technology, May 1997)

[13] Ronald L. Rivest, Butler Lampson, "SDSI - A Simple Distributed Security Infrastructure", (MIT Laboratory for Computer Science and Microsoft Corporation, April 1996)

[14] Ronald L. Rivest, Dwaine Clarke et al., "Certificate Chain Discovery in SPKI/SDSI" (work in progress), (MIT Laboratory for Computer Science, Intel, BBN, November 22, 1999)

[15] Transarc Corporation, "The Andrew File System"

[16] Ronald L. Rivest, Adi Shamir, Len Adelman, "On Digital Signatures and Public Key Cryptosystems," MIT Laboratory for Computer Science Technical Memorandum 82 (April 1977).

[17] S. Shepler, "NFS Version 4 Design Considerations" (RFC 2624, Sun Microsystems, Inc., June 1999).

[18] Spencer Shepler, "NFS Version 4 Requirements" (Internet Draft), Internet Engineering Task Force Network Working Group (November 1998)

[19] J. G. Steiner, B. Clifford Neuman, and J.I. Schiller, "Kerberos: An Authentication Service for Open Network Systems", *Proceedings of the Winter 1988 Usenix Conference*, (February, 1988)

[20] Kohl, J. and B. Neuman, "The Kerberos Network Authentication Service (V5)" (RFC 1510, USC/Information Sciences Institute, September 1993.)

[21] Tung, B., et al., "Public Key Cryptography for Initial Authentication in Kerberos", (IETF Internet Draft)
(`http://www.ietf.org/proceedings/99jul/I-D/draft-ietf-cat-kerberos-pk-init-09.txt`)

[22] D. Davis, "Kerberos Plus RSA for World Wide Web Security", *In Proceedings of the USENIX workshop on Electronic Commerce* (July 1995).

[23] Sirbu, M.A. and J.C.-I. Chuang, "Distributed Authentication in Kerberos Using Public Key Cryptography", *in Symposium on Network and Distributed System Security*, (San Diego, California, 1997)

[24] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS version 3 design and implementation", 1994.

[25] Macklem, R., "The 4.4BSD NFS Implementation", Sun Microsystems Inc.

[26] M. Rosenblum and J. Ousterhout, "The design and implementation of a log-structured file system", *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, (Pacific Grove, CA, October 1991)