# POSIX Access Control Lists on Linux

Andreas Grünbacher
*SuSE Labs, SuSE Linux AG*
*Nuremberg, Germany*
agruen@suse.de

## Abstract

This paper discusses file system Access Control Lists as implemented in several UNIX-like operating systems. After recapitulating the concepts of these Access Control Lists that never formally became a POSIX standard, we focus on the different aspects of implementation and use on Linux.

## 1 Introduction

Traditionally, systems that support the POSIX (Portable Operating System Interface) family of standards [2, 11] share a simple yet powerful file system permission model: Every file system object is associated with three sets of permissions that define access for the owner, the owning group, and for others. Each set may contain Read (r), Write (w), and Execute (x) permissions. This scheme is implemented using only nine bits for each object. In addition to these nine bits, the Set User Id, Set Group Id, and Sticky bits are used for a number of special cases. Many introductory and advanced texts on the UNIX operating system describe this model [19].

Although the traditional model is extremely simple, it is sufficient for implementing the permission scenarios that usually occur on UNIX systems. System administrators have also found several workarounds for the model's limitations. Some of these workarounds require nonobvious group setups that may not reflect organizational structures. Only the root user can create groups or change group membership. Set-user-ID root utilities may allow ordinary users to perform some administrative tasks, but bugs in such utilities can easily lead to compromised systems. Some applications like FTP daemons implement their own extensions to the file system permission model [15]. The price of playing tricks with permissions is an increase in complexity of the system configuration. Understanding and maintaining the integrity of systems becomes more difficult.

Engineers have long recognized the deficiencies of the traditional permission model and have started to think about alternatives. This has eventually resulted in a number of Access Control List (ACL) implementations on UNIX, which are only compatible among each other to a limited degree.

This paper gives an overview of the most successful ACL scheme for UNIX-like systems that has resulted from the POSIX 1003.1e/1003.2c working group.

After briefly describing the concepts, some examples of how these are used are given for better understanding. Following that, the paper discusses Extended Attributes, the abstraction layer upon which ACLs are based on Linux. The rest of the paper deals with implementation, performance, interoperability, application support, and system maintenance aspects of ACLs.

The author was involved in the design and implementation of extended attributes and ACLs on Linux, which covered the user space tools and the kernel implementation for Ext2 and Ext3, Linux's most prominent file systems. Parts of the design of the system call interface are attributed to Silicon Graphics's Linux XFS project, particularly to Nathan Scott.

## 2 The POSIX 1003.1e/1003.2c Working Group

A need for standardizing other security relevant areas in addition to just ACLs was also perceived, so eventually a working group was formed to define security extensions within the POSIX 1003.1 family of standards. The document numbers 1003.1e (System Application Programming Interface) and 1003.2c (Shell and Utilities) were assigned for the working group's specifications. These documents are referred to as POSIX.1e in the remainder of this paper. The working group was focusing on the following extensions to POSIX.1: Access Control Lists (ACL), Audit, Capability, Mandatory Access Control (MAC), and Information Labeling.

Unfortunately, it eventually turned out that standardizing all these diverse areas was too ambitious a goal. In January 1998, sponsorship for 1003.1e and 1003.2c was withdrawn. While some parts of the documents produced by the working group until then were already of high quality, the overall works were not ready for publication as standards. It was decided that draft 17, the last version of the documents the working group had produced, should be made available to the public. Today, these documents can be found at Winfried Trümper's Web site [27].

Several UNIX system vendors have implemented various parts of the security extensions, augmented by vendor-specific extensions. The resulting versions of their operating systems have often been labeled “trusted” operating systems, e.g., Trusted Solaris, Trusted Irix, Trusted AIX. Some of these “trusted” features have later been incorporated into the vendors’ main operating systems.

ACLs are supported on different file system types on almost all UNIX-like systems nowadays. Some of these implementations are compatible with draft 17 of the specification, while others are based on older drafts. Unfortunately, this has resulted in a number of subtle differences among the different implementations.

The TrustedBSD project (http://www.trustedbsd.org/) lead by Robert Watson has implemented versions of the ACL, Capabilities, MAC, and Audit parts of POSIX.1e for FreeBSD. The ACL and MAC implementations appear in FreeBSD-RELEASE as of January, 2003. The MAC implementation is still considered experimental.

## 3 Status of ACLs on Linux

Patches that implement POSIX 1003.1e draft 17 ACLs have been available for various versions of Linux for several years now. They were added to version 2.5.46 of the Linux kernel in November 2002. Current Linux distributions are still based on the 2.4.x stable kernels series. SuSE and the United Linux consortium have integrated the 2.4 kernel ACL patches earlier than others, so their current products offer the most complete ACL support available for Linux to date. Other vendors apparently are still reluctant to make that important change, but experimental versions are expected to be available later this year.

The Linux *getfacl* and *setfacl* command line utilities do not strictly follow POSIX 1003.2c draft 17, which shows mostly in the way they handle default ACLs. See section 6.

At the time of this writing, ACL support on Linux is available for the Ext2, Ext3, IBM JFS, ReiserFS, and SGI XFS file systems. Solaris-compatible ACL support for NFS version 3 exists since March 3, 2003.

## 4 How ACLs Work

The traditional POSIX file system object permission model defines three classes of users called owner, group, and other. Each of these classes is associated with a set of permissions. The permissions defined are read (r), write (w), and execute (x). In this model, the *owner class* permissions define the access privileges of the file owner, the *group class* permissions define the access privileges of the owning group, and the *other class* permissions define the access privileges of all users that are not in one of these two classes. The *ls -l* command displays the owner, group, and other class permissions in the first column of its output (e.g., “`-rw-r-----`” for a regular file with read and write access for the owner class, read access for the group class, and no access for others).

| **Entry type** | **Text form** |
|---|---|
| Owner | `user::`*`rwx`* |
| Named user | `user:`*`name`*`:`*`rwx`* |
| Owning group | `group::`*`rwx`* |
| Named group | `group:`*`name`*`:`*`rwx`* |
| Mask | `mask::`*`rwx`* |
| Others | `other::`*`rwx`* |

Table 1: Types of ACL Entries

An ACL consists of a set of entries. The permissions of each file system object have an ACL representation, even in the minimal, POSIX.1-only case. Each of the three classes of users is represented by an ACL entry. Permissions for additional users or groups occupy additional ACL entries.

Table 1 shows the defined entry types and their text forms. Each of these entries consists of a type, a qualifier that specifies to which user or group the entry applies, and a set of permissions. The qualifier is undefined for entries that require no qualification.

ACLs equivalent with the file mode permission bits are called *minimal* ACLs. They have three ACL entries. ACLs with more than the three entries are called *extended* ACLs. Extended ACLs also contain a mask entry and may contain any number of named user and named group entries.

These named group and named user entries are assigned to the *group class*, which already contains the owning group entry. Different from the POSIX.1 permission model, the group class may now contain ACL entries with different permission sets, so the group class permissions alone are no longer sufficient to represent all the detailed permissions of all ACL entries it contains. Therefore, the meaning of the group class permissions is redefined: under their new semantics, they represent an upper bound of the permissions that any entry in the group class will grant.

This upper bound property ensures that POSIX.1 applications that are unaware of ACLs will not suddenly and unexpectedly start to grant additional permissions once ACLs are supported.

In minimal ACLs, the group class permissions are identical to the owning group permissions. In extended ACLs, the group class may contain entries for additional users or groups. This results in a problem: some of these additional entries may contain permissions that are not contained in the owning group entry, so the owning group entry permissions may differ from the group class permissions.

This problem is solved by the virtue of the mask entry.

With minimal ACLs, the group class permissions map to the owning group entry permissions. With extended ACLs, the group class permissions map to the mask entry permissions, whereas the owning group entry still defines the owning group permissions. The mapping of the group class permissions is no longer constant. Figure 1 shows these two cases.

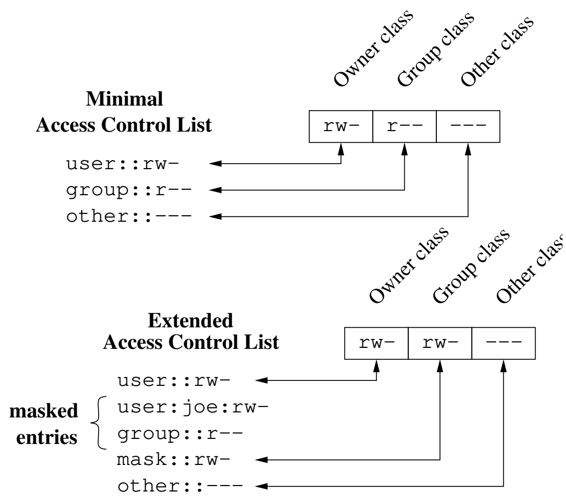


Figure 1: Mapping between ACL Entries and File Mode Permission Bits

When an application changes any of the owner, group, or other class permissions (e.g., via the *chmod* command), the corresponding ACL entry changes as well. Likewise, when an application changes the permissions of an ACL entry that maps to one of the user classes, the permissions of the class change.

The group class permissions represent the upper bound of the permissions granted by any entry in the group class. With minimal ACLs this is trivially the case. With extended ACLs, this is implemented by masking permissions (hence the name of the mask entry): permissions in entries that are a member of the group class which are also present in the mask entry are effective. Permissions that are absent in the mask entry are masked and thus do not take effect. See Table 2.

| Entry type | Text form | Permissions |
|---|---|---|
| Named user | `user:joe:r-x` | `r-x` |
| Mask | `mask::rw-` | `rw-` |
| **Effective permissions** | | `r--` |

Table 2: Masking of Permissions

The owner and other entries are not in the group class. Their permissions are always effective and never masked.

So far we have only looked at ACLs that define the current access permissions of file system objects. This type is called *access* ACL. A second type called *default* ACL is also defined. They define the permissions a file system object inherits from its parent directory at the time of its creation. Only directories can be associated with default ACLs. Default ACLs for non-directories would be of no use, because no other file system objects can be created inside non-directories. Default ACLs play no direct role in access checks.

When a directory is created inside a directory that has a default ACL, the new directory inherits the parent directory's default ACL both as its access ACL and default ACL. Objects that are not directories inherit the default ACL of the parent directory as their access ACL only.

The permissions of inherited access ACLs are further modified by the *mode* parameter that each system call creating file system objects has. The *mode* parameter contains nine permission bits that stand for the permissions of the owner, group, and other class permissions. The effective permissions of each class are set to the intersection of the permissions defined for this class in the ACL and specified in the *mode* parameter.

If the parent directory has no default ACL, the permissions of the new file are determined as defined in POSIX.1. The effective permissions are set to the permissions defined in the *mode* parameter, minus the permissions set in the current *umask*.

The umask has no effect if a default ACL exists.

## 4.1 Access Check Algorithm

A process requests access to a file system object. Two steps are performed. Step one selects the ACL entry that most closely matches the requesting process. The ACL entries are looked at in the following order: owner, named users, (owning or named) groups, others. Only a single entry determines access. Step two checks if the matching entry contains sufficient permissions.

A process can be a member in more than one group, so more than one group entry can match. If any of these matching group entries contain the requested permissions, one that contains the requested permissions is picked (the result is the same no matter which entry is picked). If none of the matching group entries contains the requested permissions, access will be denied no matter which entry is picked.

The access check algorithm can be described in pseudo-code as follows.

**If** the user ID of the process is the owner, the owner entry determines access

**else if** the user ID of the process matches the qualifier in one of the named user entries, this entry determines access

**else if** one of the group IDs of the process matches the owning group and the owning group entry contains the requested permissions, this entry determines access

**else if** one of the group IDs of the process matches the qualifier of one of the named group entries and this entry contains the requested permissions, this entry determines access

**else if** one of the group IDs of the process matches the owning group or any of the named group entries, but neither the owning group entry nor any of the matching named group entries contains the requested permissions, this determines that access is denied

**else** the other entry determines access.

**If** the matching entry resulting from this selection is the owner or other entry and it contains the requested permissions, access is granted

**else if** the matching entry is a named user, owning group, or named group entry and this entry contains the requested permissions and the mask entry also contains the requested permissions (or there is no mask entry), access is granted

**else** access is denied.

## 5 Access ACL Example

Let us start by creating a directory and checking its permissions. The *umask* determines which permissions will be masked off when the directory is created. A umask of 027 (octal) disables write access for the owning group and read, write, and execute access for others.

```
$ umask 027
$ mkdir dir
$ ls -dl dir
drwxr-x--- ... agruen suse ... dir
```

The first character *ls* prints represents the file type (`d` for directory). The string "`rwx` `r-x` `---`" represents the resulting permissions for the new directory: read, write, and execute access for the owner and read and execute access for the owning group. The dots in the output of *ls* stand for text that is not relevant here and has been removed.

These base permissions have an equivalent representation as an ACL. ACLs are displayed using the *getfacl* command.

```
$ getfacl dir
# file: dir
# owner: agruen
# group: suse
user::rwx
group::r-x
other::---
```

The first three lines of output contain the file name, owner, and owning group of the file as comments. Each of the following lines contains an ACL entry for one of the three classes of users: owner, group, and other.

The next example grants read, write, and execute access to user Joe in addition to the existing permissions. For that, the *-m* (modify) argument of *setfacl* is used. The resulting ACL is again shown using the *getfacl* command. The *–omit-header* option to *getfacl* suppresses the three-line comment header containing the file name, owner, and owning group to shorten the examples shown.

```
$ setfacl -m user:joe:rwx dir
$ getfacl --omit-header dir
user::rwx
user:joe:rwx
group::r-x
mask::rwx
other::---
```

Two additional entries have been added to the ACL: one is for user Joe and the other is the mask entry. The mask entry is automatically created when needed but not provided. Its permissions are set to the union of the permissions of all entries that are in the group class, so the mask entry does not mask any permissions.

The mask entry now maps to the group class permissions. The output of *ls* changes as shown next.

```
$ ls -dl dir
drwxrwx---+ ... agruen suse ... dir
```

An additional "+" character is displayed after the permissions of all files that have extended ACLs. This seems like an odd change, but in fact POSIX.1 allocates this character position to the optional alternate access method flag, which happens to default to a space character if no alternate access methods are in use.

The permissions of the group class permissions include write access. Traditionally such file permission bits would indicate write access for the owning group. With ACLs, the effective permissions of the owning group are defined as the intersection of the permissions of the owning group and mask entries. The effective permissions of the owning group in the example are still `r-x`, the same permissions as before creating additional ACL entries with *setfacl*.

The group class permissions can be modified using the *setfacl* or *chmod* command. If no mask entry exists, *chmod* modifies the permissions of the owning group entry as it does traditionally. The next example removes write access from the group class and checks what happens.

```
$ chmod g-w dir
$ ls -dl dir
drwxr-x---+ ... agruen suse ... dir
```

```
$ getfacl --omit-header dir
user::rwx
user:joe:rwx            #effective:r-x
group::r-x
mask::r-x
other::---
```

As shown, if an ACL entry contains permissions that are disabled by the mask entry, *getfacl* adds a comment that shows the effective set of permissions granted by that entry. Had the owning group entry had write access, there would have been a similar comment for that entry. Now see what happens if write access is given to the group class again.

```
$ chmod g+w dir
$ ls -dl dir
drwxrwx---+ ... agruen suse ... dir
$ getfacl --omit-header dir
user::rwx
user:joe:rwx
group::r-x
mask::rwx
other::---
```

After adding the write permission to the group class, the ACL defines the same permissions as before taking the permission away. The *chmod* command has a nondestructive effect on the access permissions. This preservation of permissions is an important feature of POSIX.1e ACLs.

## 6 Default ACL Example

In the following example, we add a default ACL to the directory. Then we check what *getfacl* shows.

```
$ setfacl -d -m group:toolies:r-x dir
$ getfacl --omit-header dir
user::rwx
user:joe:rwx
group::r-x
mask::rwx
other::---
default:user::rwx
default:group::r-x
default:group:toolies:r-x
default:mask::r-x
default:other::---
```

Following the access ACL, the default ACL is printed with each entry prefixed with "`default:`". This output format is an extension to POSIX.1e that is found on Solaris and Linux. A strict implementation of POSIX 1003.2c would only show the access ACL. The default ACL would be shown with the *-d* option to *getfacl*.

We have only specified an ACL entry for the *toolies* group in the *setfacl* command. The other entries required for a complete ACL have automatically been copied from the access ACL to the default ACL. This is a Linux-specific extension; on other systems all entries may need to be specified explicitly.

The default ACL contains no entry for Joe, so Joe will not have access (except possibly through group membership or the other class permissions).

A subdirectory inherits ACLs as shown next. Unless otherwise specified, the *mkdir* command uses a value of 0777 as the *mode* parameter to the mkdir system call, which it uses for creating the new directory. Observe that both the access and the default ACL contain the same entries.

```
$ mkdir dir/subdir
$ getfacl --omit-header dir/subdir
user::rwx
group::r-x
group:toolies:r-x
mask::r-x
other::---
default:user::rwx
default:group::r-x
default:group:toolies:r-x
default:mask::r-x
default:other::---
```

Files created inside *dir* inherit their permissions as shown next. The *touch* command passes a *mode* value of 0666 to the kernel for creating the file.

All permissions not included in the *mode* parameter are removed from the corresponding ACL entries. The same has happened in the previous example, but there was no noticeable effect because the value 0777 used for the *mode* parameter represents a full set of permissions.

```
$ touch dir/file
$ ls -l dir/file
-rw-r-----+ ... agruen suse ... dir/file
$ getfacl --omit-header dir/file
user::rw-
group::r-x              #effective:r--
group:toolies:r-x       #effective:r--
mask::r--
other::---
```

No permissions have been removed from ACL entries in the group class; instead they are merely masked and thus made ineffective. This ensures that traditional tools like compilers will interact well with ACLs. They can create files with restricted permissions and mark the files executable later. The mask mechanism will cause the right users and groups to end up with the expected permissions.

## 7 Extended Attributes

In this section we begin detailing the implementation of ACLs in Linux.

ACLs are pieces of information of variable length that are associated with file system objects. Dedicated strategies for storing ACLs on file systems might be devised, as Solaris does on the UFS file system [13]. Each inode on a UFS file system has a field called *i_shadow*. If an inode has an ACL, this field points to a *shadow inode*. On the file system, shadow inodes are used like regular files. Each shadow inode stores an ACL in its data blocks. Multiple files with the same ACL may point to the same shadow inode.

Because other kernel and user space extensions in addition to ACLs benefit from being able to associate pieces of information with files, Linux and most other UNIX-like operating systems implement a more general mechanism called Extended Attributes (EAs). On these systems, ACLs are implemented as EAs.

Extended attributes are name and value pairs associated permanently with file system objects, similar to the environment variables of a process. The EA system calls used as the interface between user space and the kernel copy the attribute names and values between the user and kernel address spaces. The Linux attr(5) manual page contains a more complete description of EAs as found on Linux. A paper by Robert Watson discussing supporting infrastructure for security extensions in FreeBSD contains a comparison of different EA implementations on different systems [25].

Other operating systems, such as Sun Solaris, Apple MacOS, and Microsoft Windows, allow multiple streams (or forks) of information to be associated with a single file. These streams support the usual file semantics. After obtaining a handle on the stream, it is possible to access the streams' contents using ordinary file operations like read and write. Confusingly, on Solaris these streams are called extended attributes as well. The EAs on Linux and several other UNIX-like operating systems have nothing to do with these streams. The more limited EA interface offers several advantages. They are easier to implement, EA operations are inherently atomic, and the stateless interface does not suffer from overheads caused by obtaining and releasing file handles. Efficiency is important for frequently accessed objects like ACLs.

At the file system level, the obvious and straightforward approach to implement EAs is to create an additional directory for each file system object that has EAs and to create one file for each extended attribute that has the attribute's name and contains the attribute's value. Because on most file systems allocating an additional directory plus one or more files requires several disk blocks, such a simple implementation would consume a lot of space, and it would not perform very well because of the time needed to access all these disk blocks. Therefore, most file systems use different mechanisms for storing EAs.

### 7.1 Ext2 and Ext3

As described in the Linux kernel sources, each inode has a field that is called *i_file_acl* for historic reasons. If this field is not zero, it contains the number of the file system block on which the EAs associated with this inode are stored. This block contains both the names and values of all EAs associated with the inode. All EAs of an inode must fit on the same EA block.

For improved efficiency, multiple inodes with identical sets of EAs may point to the same EA block. The number of inodes referring to an EA block are tracked by a reference count in the EA block. EA block sharing is transparent for the user: Ext3 keeps an LRU list of recently accessed EA blocks and a table that has two indices (implemented as hash tables of double linked lists). One index is by block number. The other is by a checksum of the block's contents. Blocks that contain the same data with which a new inode shall be associated are reused until the block's reference count reaches an upper limit of 1024. This limits the damage a single disk block failure may cause. When an inode refers to a shared EA block and that inode's EAs are changed, a copy-on-write mechanism is used, unless another cached EA block already contains the same set of attributes, in which case that block is used.

The current implementation requires all EAs of an inode to fit on a single disk block, which is 1, 2, or 4 KiB. This also determines the maximum size of individual attributes.

If the sets of EAs tend to be unique among inodes, no sharing is possible and the time spent checking for potential sharing is wasted. If each inode has a unique set of EAs, each of these sets will be stored on a separate disk block, which can lead to a lot of slack space. The extreme case is applications that need to store unique EAs for each inode. Fortunately for many common workloads, the EA sharing mechanism is highly effective.

Alternative designs with fewer limitations have been proposed [5], but it seems that they are not easy to actually implement. No alternatives to the existing scheme exist so far.

### 7.2 JFS

JFS stores all EAs of an inode in a consecutive range of blocks on the file system (i.e., in an extent) [3]. The extended attribute name and value pairs are stored consecutively in this extent. If the entire EAs are small enough, they are stored entirely within the inode to which they belong.

JFS does not implement an EA sharing mechanism. It does not have the one-disk-block limitation of Ext2 and Ext3. The size of individual attributes is limited to 64 KiB.

### 7.3 XFS

Of the file systems currently supported in Linux, XFS uses the most elaborate scheme for storing extended attributes [1]. Small sets of EAs are stored directly in inodes, medium-sized sets are stored on leaf blocks of B+ trees, and large sets of EAs are stored in full B+ trees. This results in performance characteristics similar to directories on XFS: although rarely needed, very large numbers of EAs can be stored efficiently.

XFS has a configurable inode size that is determined at file system create time. The minimum size is 256 bytes, which is also the default. The maximum size is one half of the file system block size. In the minimum case, the inodes are too small to hold ACLs, so they will be stored externally. If the inode size is increased, ACLs will fit directly in the inode. Since inodes and their ACLs are often accessed within a short period of time, this results in faster access checks, but also wastes more disk space.

XFS does not have an attribute sharing mechanism. The size of individual attributes is limited to 64 KiB.

### 7.4 ReiserFS

ReiserFS supports tail merging of files, which means that several files can share the same disk block for storing their data. This makes the file system very efficient for many small files. One potential drawback is that tail merging can consume a noticeable amount of CPU time.

Since ReiserFS is so good at handling small files, EAs can directly use this mechanism. For each file that has EAs, a directory with a name derived from a unique inode identifier is created inside a special directory. The special directory is usually hidden from the file system namespace. Inside the inode specific directory, each EA is stored as a separate file. The file name equals the attribute name. The file's contents are the attribute value.

ReiserFS does not implement an attribute sharing mechanism, but such an extension will possibly be implemented in the future. Sharing could even be implemented on a per-attribute bases, so the result would be a highly efficient and flexible solution. The size of individual attributes is limited to 64 KiB.

## 8 ACL Implementations

An interesting design decision is how ACLs should be passed between user space and the kernel, and inside the kernel, between the virtual file system (VFS) and the low-level file system layer. FreeBSD, Solaris, Irix, and HP-UX all have separate ACL system calls [9, 17, 21, 23].

Linux does not have ACL system calls. Instead, ACLs are passed between the kernel and user space as EAs. This reduces the number of system interfaces, but with the same number of end operations. While the ACL system calls provide a more explicit system interface, the EA interface is easier to adapted to future requirements, such as non-numerical identifiers for users and groups in ACL entries.

The rationale for using separate ACL system calls in FreeBSD was that some file systems support EAs but not ACLs, and some file systems support ACLs but not EAs, so EAs are treated as pure binary data. EAs and ACLs only become related inside a file system. [24, 26].

The rationale for the Linux design was to provide access to all meta data pertinent to a file system object through the same interface. Different classes of attributes that are recognized by name are reserved for system objects such as ACLs. The attribute names "system.posix_acl_access" and "system.posix_acl_default" are used for the access and default ACL of a file, respectively. The ACL attribute values are in a canonical, architecture-independent binary format. File systems that do not implement ACLs but do implement EAs, or ones that implement ACLs as something other than EAs, need to recognize the relevant attribute names.

While it is possible to manipulate ACLs directly as EAs, at the application level this is usually not done: since the EA system calls are Linux-specific, such applications would not be portable. Other systems support similar EA mechanisms, but with different system call interfaces. Applications that want to use POSIX.1 ACLs in a portable way are expected to use the *libacl* library, which implements the ACL-specific functions of POSIX.1e draft 17.

The access ACL of a file system object is accessed for every access decision that involves that object. Access checking is performed on the whole path from the namespace root to the file in question. It is important that ACL access checks are efficient. To avoid frequently looking up ACL attributes and converting them from the machine-independent attribute representation to a machine-specific representation, the Ext2, Ext3, JFS, and ReiserFS implementations cache the machine-specific ACL representations. This is done in addition to the normal file system caching mechanisms, which use either the page cache, the buffer cache, or both. XFS does not use this additional layer of caching.

Most UNIX-like systems that support ACLs limit the number of ACL entries allowed to some reasonable number. Table 3 shows the limits on Linux.

ACLs with a high number of ACL entries tend to become more difficult to manage. More than a handful of

| File system | Max. entries |
|---|---|
| XFS | 25 |
| Ext2, Ext3 | 32 |
| ReiserFS, JFS | 8191 |

Table 3: Maximum Number of Supported ACL Entries

ACL entries are usually an indication of bad application design. In most such cases, it makes more sense to make better use of groups instead of bloating ACLs.

The ReiserFS and JFS implementations define no limit on the number of ACL entries, so a limit is only imposed by the maximum size of EA values. The current EA size limit is 64 KiB, or 8191 ACL entries, which is too high for ACLs in practice: besides being impractical to work with, the time it would take to check access in such huge ACLs may be prohibitive.

## 9 Compatibility

An important aspect of introducing new file system features is how systems are upgraded, and how systems that do not support the new features are affected. File system formats evolve slowly. File systems are expected to continue to work with older versions of the kernel. In some situations, like in multiple boot environments or when booting from a rescue system, it may be necessary or preferable to use a kernel that does not have EA support.

All file systems that support ACLs on Linux are either inherently aware of EAs, or are upgraded to support EAs automatically, without user intervention. Depending on the file system, this is either done during mounting or when first using extended attributes.

On all file systems discussed in this paper, when using a kernel that does not support EAs on a file system with EAs, the EAs will be ignored. On ReiserFS, EA-aware kernels actively hide the system directory that contains the EAs, so in an EA-unaware kernel this directory becomes visible. It is still protected from ordinary users through file permissions.

Working with EA file systems with EA-unaware kernels will still lead to inconsistencies when files are deleted that have EAs. In that case, the EAs will not get removed and a disk space leak will result. At least on Ext2 and Ext3, such inconsistencies can be cleaned up later by running the file system checker.

ACL-unaware kernels will only see the traditional file permission bits and will not be able to check permissions defined in ACLs. The ACL inheritance algorithm will not work.

## 10 EA and ACL Performance

Since ACLs define a more sophisticated discretionary access control mechanism, they have an influence on all access decisions for file system objects. It is interesting to compare the time it takes to perform an access decision with and without ACLs.

Measurements were performed on a PC running SuSE Linux 8.2, with the SuSE 2.4.20 kernel. The machine has an AMD Athlon processor clocked at 1.1 GHz and 512 MiB of RAM. The disk used was a 30 GB IBM Ultra ATA 100 hard drive with 7200 RPM, an average seek time of 9.8 ms, and 2 MiB of on-disk cache. The Ext2, Ext3, Reiserfs, and JFS file systems were created with default options on an 8 GiB partition. On XFS, to compare EAs that are stored in inodes and EAs that are stored externally, file systems with inode sizes of 256 bytes and 512 bytes were used. These file systems are labeled *XFS-256* and *XFS-512*, respectively.

Table 4 compares the times required for the initial access check to a file with and without an ACL after system restart. To exclude the time for loading the file's inode into the cache, a *stat* system call was performed before checking access. The time taken for the *stat* system call is not shown. The first access to the access ACL of a file may require one or more disk accesses, which are several orders of magnitude slower than accessing the cache. The actual times these disk accesses take vary widely depend on the disk speed and on the relative locations of the disk blocks that are accessed. The function used for measuring time has a resolution of 1 microsecond. In the ACL case, the file that is checked has a five-entry access ACL.

| | Without ACL | With ACL |
|---|---|---|
| Ext2 | 9 | 1743 |
| Ext3 | 10 | 3804 |
| ReiserFS | 9 | 6165 |
| XFS-256 | 14 | 7531 |
| XFS-512 | 14 | 14 |
| JFS | 13 | 13 |

Table 4: Microseconds for Initially Accessing a File After System Restart, with and without ACLs

XFS with 512-byte (or larger) inodes and JFS store the ACLs directly in the inodes. Therefore, no additional disk accesses are needed for retrieving the ACLs.

After the first repetition, all information is fully cached. Figures 2−4 show micro-benchmarks of basic operations in this state. Each test repeats the same operation many times and averages the total time spent over the number of repetitions. In all configurations except XFS-256 with ACLs, the time per access check drops to around $1 - 2$ microseconds, as the leftmost measurements in Figures 3 and 4 show.

Figure 2 compares the speed of various system calls. The getpid system call is included to show the overhead of switching between the user and kernel address spaces. The *ls -l* command indicates in its output if a file has an extended ACL. Internally, it uses the acl_extended_file function from the *libacl* library. This function is almost as fast as the stat system call, which *ls -l* also calls for each file, so the additional overhead is small. For comparison, the acl_get_file measurement shows the time it takes to retrieve a five-entry ACL.

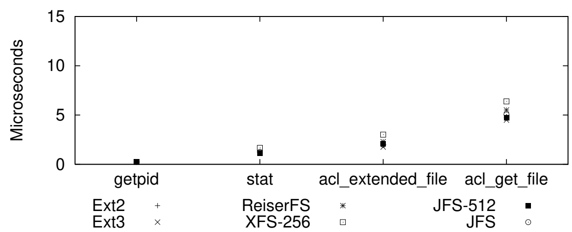


Figure 2: Various System Calls and Library Functions

Figure 3 shows the time taken for one access system call, depending on the number of directory levels in the pathname argument. Figure 4 shows the performance of the same operation with a five-entry access ACL on each directory. XFS's overhead for converting the ACL from its EA representation to its in-memory representation results in a noticeable difference, particularly with 256 byte inodes.

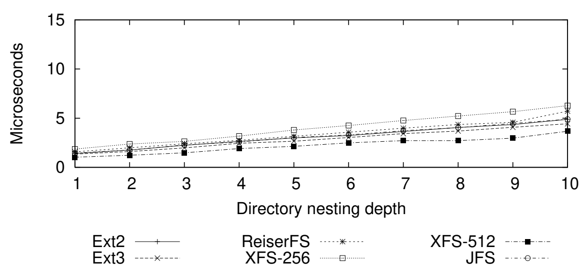


Figure 3: The Access System Call without ACLs

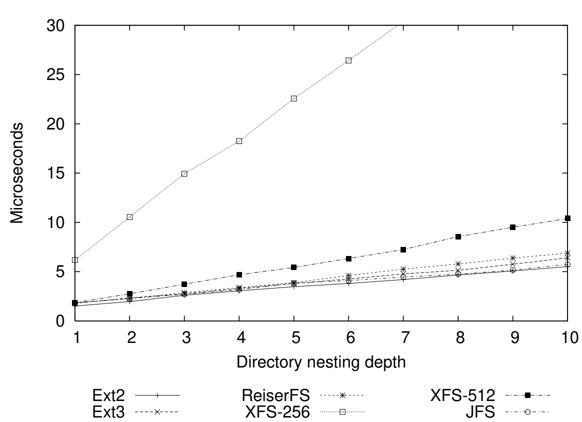


Figure 4: The Access System Call with ACLs

Tables 5 and 6 show the overhead involved when copying files from one file system to another. All the files that are copied have ACLs, but no additional EAs. Figure 5 shows the distribution of file sizes in the sample data sets. The tests show the time taken from starting the *cp* command to the return from the *sync* command, which we start immediately after the *cp* command. The *sync* command ensures that all files are immediately written. The first series uses a version of *cp* that does not support EAs or ACLs. The second series uses a version of *cp* that supports both EAs and ACLs. In both of the benchmarks, the source file system type is held constant, while the destination file system type is changed.

The sample size for the benchmark in Table 5 is small enough for all files to fit in main memory. The time for first reading the data into memory is not included in the results. These tests were repeated five times. The results show the median and the standard deviation of the results. The sample size for the benchmark shown in Table 6 is larger than either main memory or the file system journals. These tests were only executed once.

| **File system** | **Without EAs or ACLs** | | **With ACLs** | | **Overhead (%)** |
|---|---|---|---|---|---|
| Ext2 | 18.3 | 0.2 | 18.8 | 0.2 | +3 |
| Ext3 | 22.0 | 2.4 | 22.7 | 0.5 | +3 |
| ReiserFS | 9.0 | 0.1 | 12.8 | 0.1 | +42 |
| XFS-256 | 19.0 | 0.2 | 34.1 | 0.2 | +80 |
| XFS-512 | 20.1 | 0.4 | 21.4 | 0.2 | +7 |
| JFS | 38.2 | 0.6 | 36.5 | 0.2 | −4 |

Table 5: Seconds for Copying Files From Memory to a File System (11351 files, 608 directories, total file size = 137 MiB)

| **File system** | **Without EAs or ACLs** | **With ACLs** | **Overhead (%)** |
|---|---|---|---|
| Ext2 | 595 | 578 | −3 |
| Ext3 | 613 | 623 | +2 |
| ReiserFS | 518 | 538 | +4 |
| XFS-256 | 547 | 641 | +17 |
| XFS-512 | 549 | 566 | +3 |
| JFS | 654 | 590 | −11 |

Table 6: Seconds for Copying Files Between File Systems (96183 files, 6323 directories, total file size = 2.8 GiB)

Note that both benchmarks use ACLs excessively, which is a worst-case scenario. The overheads for real workloads should be much smaller.

It can be observed that the overhead of ACLs varies widely among the supported file systems. The differences show more when the I/O load is low, and get smaller as the I/O load rises. For ACLs, the Ext2 and Ext3 implementations have little overhead. ReiserFS EAs have a relatively high overhead. This may improve if attribute sharing is implemented. For XFS, increas-

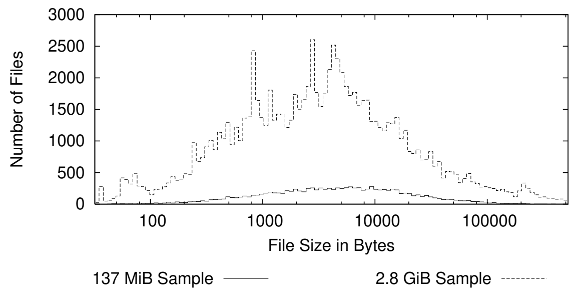


Figure 5: Distribution of File Sizes for Samples of Tables 5 and 6

ing the inode size so that ACLs can be stored directly in inodes makes a big difference.

It is unclear why JFS appears to be faster when copying ACLs than when not copying ACLs.

## 11 NFS and ACLs

Full ACL support over NFS requires two things: First, a mechanism so that all access decisions are performed in a way that honors the ACLs. Second, extensions to the NFS protocol for manipulating ACLs on remote mounted file systems.

The NFS protocol performs client-side caching to improve efficiency. In version 2 of the protocol, decisions as to who gets read access to locally cached data are performed on the client. These decisions are made under the assumption that the file mode permissions bits and the IDs of the owner and owning group are sufficient to do that. This assumption is obviously wrong if an extended permission scheme like POSIX ACLs is used on the server.

Because NFSv2 clients perform some access decisions locally, they will incorrectly grant read access to file and directory contents cached on the client to users who are a member in the owning group in two cases. First, if the group class permissions include read access, but the owning group does not have read access. Second, if the owning group does have read access, but a named user entry for that user exists that does not allow read access. Both situations are rare. Workarounds exist that reduce the permissions on the server side so that clients only see a safe subset of the real permissions [7, 10]. No anomalies exist for users who are not a member in the owning group.

There are two ways to solve this problem. One is to extend the access check algorithm used on the client. The other is to delegate access decisions to the server and possibly cache those decisions for a defined period of time on the client. The first solution would probably scale better to a high number of readers on the client side, as long as the server and all clients can agree on the access check algorithms use. Unfortunately, this approach falls apart as soon as servers implement different permission schemes.

Version 3 of the NFS protocol therefore defines a new remote procedure call (RPC) called ACCESS for delegating access decisions to the server. This RPC is similar to the access system call. NFSv3 clients are expected to use this RPC for determining to whom to grant access to cached contents.

The NFSv3 protocol unfortunately does not define mechanisms for transferring ACLs. As a consequence, different vendors have implemented proprietary protocol extensions that are incompatible with each other. Solaris implements an NFSv3 protocol extension called NFS ACL that supports ACLs only. Irix implements a more general protocol that supports EAs and passes ACLs as special EAs.

NFSv4 defines the structure and semantics of its own kind of ACLs, along with RPCs for transferring them between clients and servers. NFSv4 ACLs are similar to Microsoft Windows ACLs [14]. Unfortunately, the designers of NFSv4 have mostly ignored the existence of POSIX ACLs, so NFSv4 ACLs are not compatible with POSIX ACLs. Marius Aamodt Eriksen describes a one-way mapping between POSIX ACLs and NFSv4 ACLs [6], but this mapping is impractical. One of the central concepts in POSIX ACLs, which is needed to ensure compatibility with legacy POSIX.1 applications, is the mask entry. The NFSv4 ACL model could be extended by the mask concept. Although this would greatly improve interoperability with POSIX ACLs, proposals to extend the NFSv4 specification have so far been rejected.

Partial NFSv3 support has been added in the 2.2 Linux kernel series. The ACCESS RPC was added to the kernel NFS daemon in version 2.2.18, but the NFS client only correctly uses the ACCESS RPC in the 2.5 kernel series. A patch for older kernels exists since kernel version 2.4.19, which is included in the SuSE and UnitedLinux products.

Because the ACCESS RPC can lead to noticeable network overhead even on file systems that are known not to include any ACLs, the Linux NFSv3 client allows to mount file systems with the *noacl* mount option. Then the NFS client will use neither the ACCESS RPC nor the GETACL or SETACL RPCs. To ensure that no ACLs can be set on the server, the Ext2, Ext3, JFS, and ReiserFS file systems can be mounted on the server without ACL support by omitting the *acl* mount option.

Since March 3, 2003, an implementation of Sun's NFS ACL protocol for Linux (which is also included in SuSE Linux 8.2) is available at http://acl.bestbits.at/, with friendly permission from Sun to use it. The NFS ACL protocol was chosen because it is simple and supports POSIX 1003.1e draft 17 ACLs well enough. Solaris ACLs are based on an earlier draft of POSIX 1003.1e, so its handling of the mask ACL entry is slightly different than in draft 17 for ACLs with only four ACL entries. This is a corner case that occurs only rarely, so the semantic differences may not be noticeable.

## 12 Samba and ACLs

Microsoft Windows supports ACLs on its NTFS file system, and in its Common Internet File System (CIFS) protocol [20], which formerly has been known as the Server Message Block (SMB) protocol. CIFS is used to offer file and print services over a network. Samba is an Open Source implementation of CIFS. It is used to offer UNIX file and print services to Windows users. Samba allows POSIX ACLs to be manipulated from Windows. This feature adds a new quality of interoperability between UNIX and Windows.

The ACL model of Windows differs from the POSIX ACL model in a number of ways, so it is not possible to offer entirely seamless integration. The most significant differences between these two kinds of ACLs are:

- Windows ACLs support over ten different permissions for each entry in an ACL, including things such as append and delete, change permissions, take ownership, and change ownership. Current implementations of POSIX.1 ACLs only support read, write, and execute permissions.
- In the POSIX permission check algorithm, the most significant ACL entry defines the permissions a process is granted, so more detailed permissions are constructed by adding more closely matching ACL entries when needed. In the Windows ACL model, permissions are cumulative, so permissions that would otherwise be granted can only be restricted by DENY ACL entries.
- POSIX ACLs do not support ACL entries that deny permissions. A user can be denied permissions be creating an ACL entry that specifically matches the user.
- Windows ACLs have had an inheritance model that was similar to the POSIX ACL model. Since Windows 2000, Microsoft uses a dynamic inheritance model that allows permissions to propagate down the directory hierarchy when permissions of parent directories are modified. POSIX ACLs are inherited at file create time only.
- In the POSIX ACL model, access and default ACLs are orthogonal concepts. In the Windows ACL model, several different flags in each ACL entry control when and how this entry is inherited by container and non-container objects.
- Windows ACLs have different concepts of how permissions are defined for the file owner and owning group. The owning group concept has only been added with Windows 2000. This leads to different results if file ownership changes.
- POSIX ACLs have entries for the owner and the owning group both in the access ACL and in the default ACL. At the time of checking access to an object, these entries are associated with the current owner and the owning group of that object. Windows ACLs support two pseudo groups called Creator Owner and Creator Group that serve a similar purpose for inheritable permissions, but do not allow these pseudo groups for entries that define access. When an object inherits permissions, those abstract entries are converted to entries for a specific user and group.

Despite the semantic mismatch between these two ACL systems, POSIX ACLs are presented in the Windows ACL editor dialog box so that they resemble native Windows ACLs pretty closely. Occasional users are unlikely to realize the differences. Experienced administrators will nevertheless be able to detect a few differences. The mapping between POSIX and Windows ACLs described here is found in this form in the SuSE and the UnitedLinux products, while the official version of Samba has not yet integrated all the improvements recently made:

- The permissions in the POSIX access ACL are mapped to Windows access permissions. The permissions in the POSIX default ACL are mapped to Windows inheritable permissions.
- Minimal POSIX ACLs consist of three ACL entries defining the permissions for the owner, owning group, and others. These entries are required. Windows ACLs may contain any number of entries including zero. If one of the POSIX ACL entries contains no permissions and omitting the entry does not result in a loss of information, the entry is hidden from Windows clients. If a Windows client sets an ACL in which required entries are missing, the permissions of that entry are cleared in the corresponding POSIX ACL.
- The mask entry in POSIX ACLs has no correspondence in Windows ACLs. If permissions in a POSIX ACL are ineffective because they are masked and such an ACL is modified via CIFS, those masked permissions are removed from the ACL.
- Because Windows ACLs only support the Creator Owner and Creator Group pseudo groups for inheritable permissions, owner and owning group entries in a default ACL are mapped to those pseudo groups. For access ACLs, these entries are

mapped to named entries for the current owner and the current owning group (e.g., the POSIX ACL entry "`u::rw`" of a file owned by Joe is treated as "`u:joe:rw`").

If an access ACL contains named ACL entries for the owner or owning group (e.g., if one of Joe's files also has a "`u:joe:...`" entry), the permissions defined in such entries are not effective unless file ownership changes, so such named entries are ignored. When an ACL is set by Samba that contains Creator Owner or Creator Group entries, these entries are given precedence over named entries for the current owner and owning group, respectively.

- POSIX access ACL and default ACL entries that define the same permissions are mapped to a Windows ACL entry that is flagged as defining both access and inheritable permissions.

## 13 Backup and Restore

An important but easily overlooked aspect of introducing new features like EAs and ACLs is backup. Standard tools like GNU tar and GNU cpio do not include EA or ACL support. The *ustar* and *cpio* archive formats on which these tools are based do allow certain extensions, but no standards for storing EAs or ACLs have yet been defined.

The current version of POSIX.1 [11] defines a new utility called *pax*, which stands for *portable archive interchange*. The pax utility supports both the ustar and cpio archive formats in addition to the new *pax* archive format. This new format is based on ustar and is, to a large degree, compatible with ustar. Pax introduces a mechanism called *extended headers*. Extended headers consist of a list of attributes very similar to EAs. They are used to store things that cannot be represented in ustar headers, such as sub-second resolution file timestamps, or file sizes of 8 Gib or more.

The pax archive format is a good match for storing EAs and ACLs. As neither EAs nor ACLs are part of any POSIX standard, no specific format for EAs or ACLs to use in extended headers has been defined. The specification leaves room for vendor-specific attributes tagged with the vendor name, so even if no agreement can be reached soon on the EA or ACL formats to be used, the vendor-specific extensions can at least be distinguished and implementations may support more than one extension.

A benefit of the pax format is that most pax archives can also be restored with tar implementations. To tar, extended headers look like files of an unknown type. The information stored in the extended headers will be lost, but files and directories will be extracted correctly. This will not work for pax archives that contain files 8 GiB or more in size; this is the maximum file size in tar archives.

The following solutions exist for backing up (and later restoring) EAs and ACLs:

- Jörg Schilling's implementation of pax and tar called *star* includes support for POSIX.1e ACLs and a few others. The resulting archives are portable among systems that implement various versions of POSIX ACLs, including FreeBSD, Irix, HP-UX, Compaq/HP Tru64, Linux, Solaris. A patch that implements Linux EA support in star exists as well. Star is available from ftp://ftp.berlios.de/pub/star/.
- On the XFS file system, the *xfsdump* and *xfsrestore* utilities can be used. However, the backup format is file-system–specific, so this approach is not recommended.
- Finally, the *getfattr* and *getfacl* utilities can dump ACLs and EAs to text files, which the *setfattr* and *setfacl* utilities are able to restore. This works reasonably well for restoring complete backups, but it is impractical for restoring individual files.

## 14 Application Support for ACLs

Today, the most basic tools that are needed to operate a system with EAs and ACLs are available: there is EA and ACL support in the basic file utilities (*ls*, *cp*, and *mv*), there are utilities for manipulating EAs and ACLs from the command line, and there are solutions for backing up and restoring a system that uses those features. Still, there are many applications that are lacking support. Although for many of them this is irrelevant, there are some classes of applications for which this leads to problems. This includes file managers, editors, and file system synchronization tools.

File managers usually can copy and move files and allow permissions to be changed. UNIX kernels have no functions for copying files or for moving files across file system boundaries. Therefore, these operations are implemented by reading from the source file and copying the data to the destination file. The kernel has no way of telling which sequences of operations are file copies or moves and which are something else, so it cannot preserve EAs and ACLs automatically. This means that applications must take care of preserving EAs and ACLs themselves as needed.

Front-ends that allow manipulation of permissions usually allow manipulation of the standard POSIX.1 permissions, but none are known yet that allow manipulation of ACLs. It is quite likely that for the foreseeable future, ACL editing support will not become available except with the command line utilities.

Some editors suffer from the file copying problem as well. There are two ways of safely modifying a file. One is to create a copy of the original file and then to modify the original file. The other is to rename the original file and then to create a new file that includes the modifications in the original file's place. The latter is equivalent to copying files as far as EAs and ACLs are concerned. The option used also has additional consequences for files that are symlinks and for files with a link count greater than one. *Emacs* and *vi*, the most popular editors on UNIX-like systems, both can be configured to use either method.

When preserving permissions, it is important that as much information is preserved as possible. Although this seems obvious, correctly implementing this is not trivial. There are a number of complications that make that operation prone to implementation errors. This is especially true if the source and destination files reside on different file systems, only one of which has ACL support. To take this additional burden from programmers, the current version of *libacl* includes functions for copying EAs and ACLs between files [8].

It would also be nice to have EA and ACL support in popular utilities like *scp* and *rsync*. Unfortunately, utilities that transfer files between different systems have a much harder time handling incompatibilities. Only some UNIX-like systems support the POSIX.1e ACL library functions and other systems have their own operating system interfaces. Even worse, the semantics of ACLs differs widely among UNIX systems alone, not to speak of non-UNIX systems. The semantics and system interfaces for EAs unfortunately are different among various systems as well.

## 15 Conclusions and Future Work

Integrating support for EAs was an important step that is going to simplify the development of various sorts of applications, including system level security extensions, such as Capabilities and Mandatory Access Control schemes, Hierarchical Storage Management, and many user space solutions.

POSIX.1e ACLs are a consequent extension of the POSIX.1 permission model. They support more fine-grained and complex permission scenarios that are difficult or impossible to implement in the traditional model.

It is unfortunate that neither of these areas has been formally standardized. Already there is a wild mix of implementations with subtle differences and incompatibilities. As more implementations become available, future standardization is becoming more and more unlikely.

As for POSIX ACLs, although they are a substantial improvement, many restrictions remain:

- More find-grained permissions would be useful. For directories, the write permission includes the rights to add and remove files. For files, it allows overwriting of existing contents as well as appending. For directories, the sticky bit helps, but its applicability is limited. Ext2 and Ext3 support flags like *append* and *immutable* that can be set on a per-file basis. ACL permissions would be per-user or per-group.
- The creator of a file is also the initial file owner. There is no way to restrict the file owner from modifying permissions. It is impossible to implement upload directories securely at the file system level that don't allow to modify existing files, or that don't allow users to create files that other users can read.
- It is not possible to completely delegate administration of a directory to a regular user. It would be necessary to ensure that this locally-privileged user cannot be denied access to files below that directory and that this user can change permissions, and potentially also assign or take ownership of files.

On UNIX-like systems, it is easier to work around problems than on other popular systems, but these workarounds cause complexity and may contain bugs. It may be better to solve some of the existing problems at their root. All extensions must be designed carefully to simplify the integration with existing systems like Windows/CIFS and NFSv4.

The UNIX way of identifying users and groups by numeric IDs is a problem in large networks [18]. Like the whole POSIX.1 permission model, current implementations of POSIX.1e ACLs are based on these unique IDs. Maintaining central user and group databases becomes increasingly difficult with increasing network size. The CIFS and NFSv4 protocols solve this problem differently.

In CIFS, users and groups are identified by globally unique security identifers (SIDs). Processes have a number of SIDs, which determine their privileges. CIFS ACLs may contain SIDs from different domains.

In NFSv4, users and groups are identified by a string of the form "user@domain". Implementations of NFSv4 are expected to have internal representations for local users, such as unique user or group IDs. ACLs may contain local and non-local user or group identifiers.

Current implementations of POSIX ACLs only support numeric user or group identifiers within the local domain. Allowing non-local identifiers in ACLs seems possible but difficult. A consequent implementation would require substantial changes to the process model. At a minimum, in addition to non-local user and group identifiers in ACL entries, file ownership and group ownership for non-local users and groups would have to be supported.

## 16 Acknowledgments

I would like to thank SuSE for allowing me to continue working on ACLs. Much of what was achieved in recent months would not have been possible without the developers from Silicon Graphics's XFS for Linux project. Many thanks to Robert Watson and Erez Zadok and to the anonymous Usenix reviewers for their many suggestions that have lead to countless improvements in the text.

## References


[1] Curtis Anderson: *xFS Attribute Manager Design.* Technical Report, Silicon Graphics, October 1993. http://oss.sgi.com/projects/xfs/design_docs/xfsdocs93_pdf/attributes.pdf

[2] Austin Common Standards Revision Group. http://www.opengroup.org/austin/

[3] Steve Best, Dave Kleikamp: *How the Journaled File System handles the on-disk layout.* IBM developerWorks, May 2000. http://www-124.ibm.com/developerworks/oss/jfs/

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

[5] Andreas Dilger: *[RFC] new design for EA on-disk format.* Mailing list communication, July 10, 2002. http://acl.bestbits.at/pipermail/acl-devel/2002-July/001077.html

[6] Marius Aamodt Eriksen: *Mapping Between NFSv4 and Posix Draft ACLs.* Internet Draft, October 2002. http://www.citi.umich.edu/u/marius/draft-eriksen-nfsv4-acl-01.txt

[7] Andreas Grünbacher: *Known Problems and Bugs in the Linux EA and ACL implementations.* March 20, 2003. http://acl.bestbits.at/problems.html

[8] Andreas Grünbacher: *Preserving ACLs and EAs in editors and file managers.* February 18, 2003. http://www.suse.de/˜agruen/ea-acl-copy/ for a description.

[9] Hewlett-Packard: *acl(2): Set a file's Access Control List (ACL) information.* HP-UX Reference. http://docs.hp.com/

[10] Hewlett-Packard: *acl(4): Access control list.* Compaq Tru64 Reference Pages. http://www.hp.com/

[11] *IEEE Std 1003.1-2001 (Open Group Technical Standard, Issue 6), Standard for Information Technology—Portable Operating System Interface (POSIX) 2001.* ISBN 0-7381-3010-9. http://www.ieee.org/

[12] IEEE 1003.1e and 1003.2c: *Draft Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) and Part 2: Shell and Utilities, draft 17 (withdrawn).* October 1997. http://wt.xpilot.org/publications/posix.1e/

[13] Jim Mauro: *Controlling permissions with ACLs.* Describes internals of UFS's shadow inode concept. SunWorld Online, June 1998.

[14] Microsoft Platform SDK: *Access Control Lists.* http://msdn.microsoft.com/

[15] Mark Lowes: *Proftpd: A User's Guide* March 31, 2003. http://proftpd.linux.co.uk/

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

[17] Silicon Graphics: *acl(4): Access Control Lists.* Irix manual pages. http://techpubs.sgi.com/

[18] J. Spadavecchia, E. Zadok: *Enhancing NFS Cross-Administrative Domain Access.* Proceedings of the Annual USENIX Technical Conference, FreeNIX Track, Pages 181–194. Monterey, CA, June 2002.

[19] W. Richard Stevens: *Advanced Programming in the UNIX(R) Environment.* Addison-Wesley, June 1991 (ISBN 0-2015-6317-7).

[20] Storage Networking Industry Association: *Common Internet File System Technical Reference.* Technical Proposal, March 2002. http://www.snia.org/tech_activities/CIFS/

[21] Sun Microsystems: *acl(2): Get or set a file's Access Control List.* Solaris 8 Reference Manual Collection. http://docs.sun.com/

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

[23] Robert N. M. Watson: *acl(3): Introduction to the POSIX.1e ACL security API.* FreeBSD Library Functions Manual. http://www.FreeBSD.org/

[24] Robert N. M. Watson: *TrustedBSD: Adding Trusted Operating System Features to FreeBSD.* USENIX Technical Conference, Boston, MA, June 28, 2001. http://www.trustedbsd.org/docs.html

[25] Robert N. M. Watson: *Introducing Supporting Infrastructure for Trusted Operating System Support in FreeBSD.* BSDCon 2000, Monterey, CA, September 8, 2000. http://www.trustedbsd.org/docs.html

[26] Robert N. M. Watson: *Personal communication.* March 28, 2003.

[27] Winfried Trümper: *Summary about Posix.1e.* Publicly available copies of POSIX 1003.1e/1003.2c. February 28, 1999. http://wt.xpilot.org/publications/posix.1e/