Journal Pre-proof

FGACFS: a Fine-Grained Access Control for *nix Userspace File System

Nikita Yu. Lovyagin, George A. Chernishev, Kirill K. Smirnov, Roman Yu. Dayneko

PII: S0167-4048(19)30179-8
DOI: https://doi.org/10.1016/j.cose.2019.101632
Reference: COSE 101632

To appear in: *Computers & Security*

Received date: 11 December 2018
Revised date: 21 August 2019
Accepted date: 1 October 2019

Please cite this article as: Nikita Yu. Lovyagin, George A. Chernishev, Kirill K. Smirnov, Roman Yu. Dayneko, FGACFS: a Fine-Grained Access Control for *nix Userspace File System, *Computers & Security* (2019), doi: https://doi.org/10.1016/j.cose.2019.101632

FGACFS: a Fine-Grained Access Control for *nix Userspace File System

Nikita Yu. Lovyagin[a,*], George A. Chernishev[a,1,*], Kirill K. Smirnov[a,*], Roman Yu. Dayneko[a]

[a]*Saint Petersburg State University, 7–9 Universitetskaya emb., Saint Petersburg, Russia, 199034*
[b]*National Research University Higher School of Economics, 20 Myasnitskaya St., Moscow, Russia, 101000*

Abstract

In this paper we present FGACFS — a fine-grained access control file system designed for creating and administering directories with shared access in the *nix operating system family. The proposed access control model extends POSIX ACLs. Its essential features are: 1) an extensive list of enforceable permissions, 2) separating file and directory permissions, 3) two different mechanisms of permission inheritance — one for classic inheritance and one for copying permissions for newly-created objects. In overall, there are 19 file and 29 directory permission types. These permissions are designed to be implemented in a single tool and to allow control of both system users and programs simultaneously.

To evaluate our approach, we have developed a software implementation based on this model. FGACFS is a userspace file system that was created by implementing the FUSE interface. Our file system is independent of underlying network and on-disk file systems. In our experiments we have evaluated two different approaches for storing permissions and a single permission caching scheme that we have developed to speed up operations.

The conducted performance tests show the efficiency of our approach and demonstrate that our solution is ready to be deployed and used at least in small workgroups.

Keywords: access control, folder sharing, ACL, filesystems, FUSE, userspace filesystem

1. Introduction

Maintaining shared access for directories on multiuser and multi-terminal access systems is a considerably common task. It requires controlling individual user access rights to shared files and directories in order to restrict undesirable data access and modification.

The POSIX standard for File Access Permissions [1] specifies three types of access rights: read, write and execute for a file or search for a directory (rwx). These can be set for three possible user types — owner, owner group, and all others. Unfortunately, this approach does not allow to specify individual permissions for each system user.

POSIX Access Control Lists (ACLs) [2] address this shortcoming by allowing to control individual user permissions. However, they are confined to the same three operations (rwx). This leads to an inability to specify detailed access rights such as allowing a user to create files but not subdirectories, or allowing a user to create but not to delete files. Another example is giving a non-root user the right to change file ownership.

NTFS File and Folder advanced permissions [3] provide most of this functionality, but this file system access control is only available on Windows hosts. Some access control tools like SELinux [4], AppArmor [5] and others also provide some of the required functionality. However, due to being based on policy and not on access attributes, they require more effort during setup and administration phases, compared, for example, to NTFS.

Currently, several approaches exist for this task. We discuss them in detail and describe their disadvantages in the Related Work section. Afterwards, we provide a justification for our approach.

In this paper we present and evaluate an alternative solution for this problem. Our technique offers a more fine-grained model for managing permissions that can be defined not only for users, but also for applications (executable files). This approach is based on administering individual operations with files and directories.

Unlike many existing approaches, our solution allows direct setting of access rights for file system objects [1] with the possibility of right inheritance. This inheritance is propagated across the directory tree and affects both file and directory operations differently. More importantly, our solution does not require global user roles and policies to be defined. Finally, to the best of our knowledge, our solution implements such functionality in userspace for the first time.

We named our file system FGACFS, which stands for Fine-Grained Access Control File System. It is implemented using FUSE [6], which is known to provide suf-

*Corresponding authors
Email addresses: n.lovyagin@spbu.ru (Nikita Yu. Lovyagin), chernishev@gmail.com (George A. Chernishev), k.k.smirnov@spbu.ru (Kirill K. Smirnov)

[1] Throughout this study we will employ the term "file system object" to denote either a directory or a file.

ficient performance for small servers [7]. The main goal of implementing our system in userspace is allowing a non-privileged user to create and manage shared folders for small workgroups.

The contributions of this paper are the following:

1. A novel model for fine-grained access control which allows to manage file or directory permissions for individual users, user groups, and processes. The extended access control list includes 19 file permissions and 29 directory permissions. The model offers two types of permission inheritance mechanisms, each handling file and directory permissions separately.
2. An implementation of this model inside a file system in userspace, and a discussion of its limitations.
3. An experimental evaluation of our approach which involves a comparison with other file systems, a study of two permission storage approaches, and an examination of the proposed permission caching scheme.

2. Motivation

This section describes several use-cases which either could not be solved or could be hardly worked around in existing file systems.

1. Consider a shared folder with selective access for different users. In particular, a case where common users need to be allowed to create files in some directories, but not subdirectories, as well as edit and delete only their own files. Further, moderators need to be allowed to edit and delete any file, but not to create and delete subdirectories or modify access rights. Only administrators (who are not necessarily system root users or members of a system root group) have full control. This scenario is useful for collecting information (reports, work results and so on) that should be structured and distributed into subdirectories in a specific way. However, it can be implemented only in NTFS and NFSv4, not in POSIX ACL.
2. On the contrary, consider a shared folder with unrestricted access. In particular, a folder on a removable storage device. Turning off the access control check in file systems that support it is not possible in their native OS (Windows for NTFS, linux for ext4 and btrfs, and so on). Consequently, a variation of the FAT file system is often used on removable devices. This leads to known problems with reliability (FAT is non-journalised) and restrictions on volume and file size. The problem of using file systems with ACLs on removable devices arises due to identifiers of the same user varying between computers. Thus, the owner of a file on one machine will not be its owner on another. In some cases, only the machine administrator would be able to read and modify files saved on a different machine. This problem can be worked around by allowing access for all users. However, some applications could restrict access to the files they create. This can be found out after the removable device is transferred, resulting in an access problem. Similar problems could arise in a multi-user shared folder on a single computer. Our file system allows to automatically set all files and subdirectories as owned by the same user (even root or a user that does not exist on a particular machine). This and setting correct permissions would allow full access to the shared folder for all users without any possibility of restricting it.
3. The same issue exists in Android OS, where every user application is executed as being run by an individual POSIX user. To save user data, which is shared between applications, i.e. between all POSIX users, a separate FAT-formatted volume is used. Any application either has full or no access to the volume. Separating access in a way that:
   (a) some applications have access to all folders,
   (b) some applications have access to selected folders,
   (c) some folders being accessed by a number of applications

   is not possible using FAT, and in POSIX ACL and NFSv4 this will lead to issues like applications revoking some of the permissions, effectively disrupting shared access. Our fine-tuned model with the ability to "unify" the owner of all files and folders created could solve the problem.
4. We should mention the usefulness of allowing data access only for a specific program (e.g., one that was specifically approved for a particular purpose). This can be done via MAC, but not ACLs of existing file systems.
5. Several other use-cases:
   - A permission that allows to modify contents of a file, but not its size. In NFSv4 and NTFS there are separate permissions to write and to append to file, but they do not contain a separate permission to truncate file.
   - The inconvenience of changing permissions for a subdirectory tree in case when there is no permission inheritance. Unlike NTFS, NFSv4 inheritance is in fact permission copy-on-create, so a change in parent directory permissions would not result in change for child permissions. On the other hand, recursive changing is not a perfect solution, since there is no easy way to mark and exclude objects with own, not inherited permissions.
   - The inconvenience of permission configuring in systems which do not separate directory and file permissions. For example, in NFSv4 the "w" directory permission allows file creation,

but being inherited by the created file, it becomes a permission to modify file content. This behavior is not suitable in a case where a user should be allowed only to create a file and to save data without changing it afterwards.

3. Related work

The authors of study [8] show that UNIX-like systems employ a very unsophisticated mechanism for managing file access permissions: even POSIX ACL is inferior to ACL Windows and NFSv4 [9]. NFSv4 is the extension of ACL Windows that offers the capability to specify permissions for almost every individual operation (16 in total). Furthermore, it supports permission inheritance using the directory tree. NFSv4 provides a powerful access control model but requires running a server even on a local machine. That study proposes a solution: RichACLs. It is essentially a kernel-level implementation of NFSv4 that is based on ext4. In this implementation, ACLs are stored as extended attributes.

In its current state, the RichACLs module is not included into the Linux kernel, and it cannot be easily ported to other file systems. Moreover, there are no performance tests provided in the original paper. Finally, RichACLs as well as NFSv4, only partially separate types of actions that can be performed on file system objects. For example, creating a file and a directory are actions controlled by separate permissions, while creating a regular file and a symbolic link are not.

Other attempts at developing file systems with access control have been made. In study [10], the authors describe a decentralized file system which employs encryption to provide access control and security. There are two file types in this file system: mutable and immutable. Users are granted permissions to read, read and write, and check integrity of mutable files. Considering immutable files, users can only read them and check their integrity. Permissions can be delegated to other users by sharing.

The DAFS [11] system is a modification of the NFSv4 protocol that aims to improve its performance.

The paper [12] describes an algorithm for fast permission checking of file access operations in a model that supports inheritance. The access rights policy is described using XACML. A prototype is implemented in Python as an extension for the Samba protocol. It offers functionality to control basic file operations. However, its types of admissible access control permissions are the same as in Samba.

A different approach is adopted in study [13]. Here, access control is performed using a special cryptographic system developed by the authors. File attributes are used to control which text (i.e. file) the user can decipher.

In the aforementioned studies, access control is based on attributes. There are other approaches that are based, for example, on security policy, user roles, and addressed problems. This way, mandatory access control (MAC) allows to regulate permissions for processes and users to perform certain actions on files. In this model the list of allowed actions depends on the user roles and file labels. The designation of access permissions is performed on a system-wide level via adopting a security policy.

There is no standard MAC for *nix systems. Different Linux distributions use different, often incompatible, tools, e.g. SElinux [4] and AppArmor [5]. Another example is the study [14] where the authors describe development of MAC for Android that is based on SELinux.

This variety of implementations makes the MAC approach poorly portable. Nevertheless, security policy based access control is widely used in practice.

In study [15], requirements for access control in inter-organizational workflows are described. The authors study internal policies in an organization and policies that are based on tasks.

As we have already stated, such approaches are very popular and widely employed. However, an approach that employs access control lists to define permissions for individual file system objects is still useful. For example, it is more convenient to set up and maintain in case of small working groups where each member operates on a local subset of files and directories.

However, only MAC tools allow to define permissions not only for users, but also for executable files. In our case, it would be very useful to support both approaches. Thus, there is no such file system tool that provides adequate flexibility of user access rights management for *nix systems. Consequently, creating a tool that allows fine tuning of user access rights is an unsolved and actual problem.

4. Extended Access Rights List

The core of our approach is the Extended Access Rights List. It is a specialized permission list that defines admissible operations for all file system objects. Designing this list, we aimed to control access to all common file system operations on an individual basis.

Files and directories have different operational semantics, e.g. directory operations allow file and subdirectory creation, whereas file operations allow reading symlink content. Thus, we have designed separate permission lists, which are presented in Tables 1 and 2.

Permissions can be applied to one or several of the following entities: all users, owner user, owner group, all other users, individual user, individual group, and process executable. Further on we will refer to such entities as FGACFS permission categories.

In our approach, each file system object contains a list of records describing its permissions. Each record contains the permission category and two bit masks — one for allow and another for deny permissions. Available permission types are described in Tables 1–2. If the permission category is either “individual user”, “individual group”, or

Table 1: File permission attributes

Name	rwx	Description
FRD	r	read file content (open for reading)
FRA	r	read file attributes (stat call)
FRP	r	read file permissions (fine-grained ones)
FXA	r	read file extended attributes (getxattr call)
FSL	r	read symlink (if the file is a symlink)
FRW	w	rewrite file (modify content, no append)
FAP	w	append to file (write to the end)
FTR	w	truncate (reduce file size)
FCA	w	change attributes (affects utime call only)
FCP	w	change fine-grained file permissions (affects chmod also)
FCI	w	change permission inheritance of the file
FCO	w	change file owner (chown call)
FCG	w	change owner group (chown call)
FSP	w	chown and chgroup to parent dir owner:group (restricted chown call)
FRM	w	remove file (unlink call)
FMV	w	rename file (rename call)
FMX	w	Modify file extended attributes (setxattr and removexattr call)
FSX	w	(Un)set file execution for bit for current user (restricted chmod call)
FEX	x	execute file (affects stat call result only)

Table 2: Directory permission attributes

Name	rwx	Description
DRD	r	read directory context (opendir call)
DRA	r	read directory attributes (stat call)
DRP	r	read directory permissions (fine-grained ones)
DXA	r	read directory extended attributes
DAF	w	create (add) files with the same owner as of the parent directory
DAD	w	create (add) subdirectory with the same owner as of the parent directory
DAL	w	create (add) symbolic link in the directory with the same owner as of the parent directory
DAC	w	create (add) symbolic device with the same owner as of the parent directory
DAB	w	create (add) block device with the same owner as of the parent directory
DAS	w	create (add) socket with the same owner as of the parent directory
DAP	w	create (add) fifo with the same owner as of the parent directory
DOF	w	create own files
DOD	w	create own subdirectory
DOL	w	create own symbolic link in the directory
DOC	w	create own symbolic device
DOB	w	create own block device
DOS	w	create own socket
DOP	w	create own fifo
DCA	w	change attributes (utime call)
DCP	w	change directory fine-grained permissions
DCI	w	change permission inheritance rules of the directory
DCT	w	change permission transfer rules of the directory
DCO	w	chown directory
DCG	w	change owner group
DSP	w	chown and chgroup to parent dir owner:group
DRM	w	remove directory
DMV	w	rename directory
DMX	w	modify directory extended attributes
DEX	x	browse directory

"process executable", then an additional identifier (ID) is supplied.

Essentially, an entry in the list can be viewed as a (category, id) pair that is mapped to a set of allow and deny permissions. In case of an unapplicable category, we consider ID as empty.

Note that there are no mandatory records in this list. It can be even empty due to inheritance.

The categories are used to grant permissions in the following way:

- Permissions belonging to the "all users" category affect all users, regardless of their uid.
- Permissions belonging to the "owner user" affect the user whose uid corresponds to the uid of object owner.
- Permissions belonging to the "owner group" affect users whose gid corresponds to the gid of object owner.
- Permissions belonging to the "individual user" and "individual group" categories affect users with a specific uid and gid respectively (ID stores either uid or gid). These categories allow to define a custom permission list for each particular uid (gid).

  In POSIX, each running process has a user identifier (process uid) and a group identifier (process gid). These identifiers correspond to the user (and its group) which started this process. The ownership of file system objects is organized in a similar manner — each of them has a user owner (object uid) and a group of the user owner (object gid). The same is true for FGACFS, where permission classes for owner user and group refer to permissions for object owner and object owner group. For these, if object uid or gid change, the permissions are transferred to the new owner and group. In contrast to that, individual permissions (individual user, individual group) always refer to a specific user uid and user gid.

  It is worth mentioning that the "individual group" category affects all users that belong to a certain group. Therefore, to check permissions for a given user it is necessary to check permissions of all groups that they are a member of.
- Permissions belonging to "all other users" category are checked if other permission categories are unapplicable to a given (uid, gid) combination, i.e.:
  - the effective user of a calling process (i.e. a process that requested a file system object operation) is not the object owner and process gid is not the same as the object gid;
  - individual permissions are set neither for process uid nor for process gid;
  - individual permissions of all groups that a given user is a member of are not set.
- The "process executable" category describes permissions for applications — executable files of the operating system (the ID stores a full path to the executable file). FGACFS can check permissions not only for process uid and gid, but also for an executable file name that has started the process. Note it is not always possible to perform this check (see Section 9), so it is turned off automatically if unavailable. In other case it could be turned off via the corresponding mount option.

In our approach, similarly to NTFS, each permission is described by a combination of two different flags — allow and deny. Denied permissions have higher priority than the allowed ones, i.e. the operation is allowed if and only if it is allowed and not denied. This also concerns the situation where the same user action is controlled by different permission records.

For example, consider a case where a specific permission for a specific user (the "individual user" category) is set to "allow" for a directory, but is set to "deny" for its subdirectory or a file that inherits permissions from its parent directory. Here, the permission will be denied altogether for this user. Another example: a user can be a member of several groups (the "individual group" category with different gids). If an action is allowed for one of the groups and denied for another, it will be denied for this user. Or suppose that an action is allowed for all users (the "all users" category), but denied for a specific group ("individual group" category). Therefore, this action would be denied for all users that are members of this group.

It is unclear how to set up fine-grained permissions for newly created file system entries, since this information is not passed from POSIX system calls. Furthermore, configuring permissions for new files universally is quite complicated. Therefore, similarly to NTFS, we have implemented permission inheritance. Since file permissions can be inherited only from a directory, we have decided to add file permissions into the directory permission list. These additional permissions do not affect the directory itself, only its contents. Permission inheritance for files and for directories is regulated differently.

FGACFS supports two different mechanisms for passing permissions to child file system objects — we call this permission transfer. The first one — plain inheritance — describes the situation when permissions of parent directory (or another ancestor) are shared by the child object. The second one is copying the parent directory permissions to a newly created object. The copying happens only once, when the object is created.

These two mechanisms have different semantics. In the first case, changing ancestor permissions leads to changes in descendant permissions. In the second, descendant permissions remain unaffected since they belong to the object itself.

In FGACFS, plain permission inheritance is implemented in such a way that it allows keeping inherited permissions if the inheritance from the parent directory is turned off. If this happens, the object's inherited permissions are transformed into its intrinsic permissions as if they were copied from the parent object.

Thus, FGACFS supports two types of permission inheritance flags: flags regulating permission inheritance from parent and flags regulating copying of permissions to newly created children objects. In the latter case, two distinct operations are performed: inheritance flags for newly created objects are set and their permissions are initialized via copying.

Flags regulating permission inheritance involve plain inheritance flags (FPI and DPI) and flags to store inherited permissions in case of plain inheritance deactivation (FPK and DPK). Flags regulating permission transfer contain copy inherited permission flags (FPC and DPC) and flags that regulate setting of the inheritance flags for newly created objects (SFI for FPI, SFK for FPK, SDK for DPK, SFC for FPC, SDC for DPC).

Flags that start with "D" stand for directory permissions and flags that start with "F" stand for file permissions. Flags that regulate setting of the inheritance flags are recursive: they are set not only for a single object but also for its child objects. This way, we ensure full propagation of permissions. Inheritance flags are presented in Table 3.

For safety and ease of use, permissions for changing inheritance flags and transfer flags are differentiated (FCI/DCI and DCT correspondingly). This lets us, for example, to disallow a user to perform a specific operation on directory files, but to allow creation of subdirectories with full access.

Thus, checking the operation permission for a given object is done by a bottom-up traversal of the directory tree. This traversal starts if and only if FPI or DPI (depending on the type of object) flag is set. It is performed recursively in a bottom-up manner, and it continues until it encounters a directory which has the corresponding (FPI or DPI) flag unset.

The complexity of the described algorithm for fine-grained permission checking is linear with respect to the directory tree nesting level and to the size of user group lists. The impact of file system size and the number of permission records depends on the specific implementation.

5. Prototype Implementation

In order to evaluate the performance of our approach, we have developed a prototype. All source code is written in C, and it is available on our website [2]. We use a directory on the host file system to store the following:

1. files, directories and their respective contents;
2. file attributes, their time-related properties;
3. original extended attributes supported by the file system;
4. fine-grained permissions.

In this study we have tried two different representations for storing fine-grained permissions. These approaches will be described below.

For safety reasons, the directory and all contained files should be owned and be accessible only by a file system owner (not necessarily root). The FGACFS FUSE driver run by file system owner mounts host file system directory to a VFS mount point, which becomes the shared folder itself. Note that FUSE should be configured to allow multi-user access.

The implementation consists of 4 parts:

1. `libfgacfs`, which provides library routines for all FGACFS operations on the host directory;
2. `fgacfs` which is the FUSE module itself;
3. the `mkfgacfs` tool that creates a file system (i.e. host directory);
4. the `fgacfsctl` tool that manages extended ACLs.

The last tool is required since no POSIX syscall or shell tool provides this functionality. The `fgacfsctl` utility should have access to the host file system directory due to the fact that this tool could be called by any FGACFS user.

Currently, we have resolved this by setting setuid as root. However, this approach is considered unsafe, since running code with root privileges puts the whole system at risk. Nevertheless, this approach is still widely used (e.g. passwd uses it to obtain permissions for writing to password hash storage). Another possible solution is communicating with the FUSE module mount process from userspace. However, IPC routines do not allow controlling the uid of the sender process (the one that notifies mounting module of permission alteration) easily and securely. To illustrate security issues, consider the following situation: a process with a given pid and uid sends a permission alteration message. However, before the message is received, the sender process is destroyed and a new process with the same pid, but a different uid appears.

In the future, we plan to implement a safe solution that will employ a shared pipe between mount and control processes. At the moment, this pipe is used to command the mount process to drop the cache of altered entries.

We have devised two different approaches to storing extended permissions and owner for a FGACFS object. The first one is a centralized storage using an embedded database (SQLite) placed in the host directory — the "db" approach. The second one is to use the host file system extended attributes [16] — the "xattr" approach. The permissions storage approach is selected during the `mkfgacfs` call and it is automatically picked up by other tools.

The overall scheme of communication between implementation modules, user, and system is shown in the Fig. 1.

[2] https://github.com/lovyagin/fgacfs

Table 3: Inheritance flags. The first flag in the cell is the inheritance flag itself. The second is a flag that is used to set the former on the newly-created child object. To support recursive initialization the second flag also sets itself on the same object. Note that only FPI and FPK flags could be set on files.

Permission type	Inherit from parent	Copy from parent on inh. drop	Copy to child
file permissions	FPI / SFI	FPK / SFK	FPC / SFC
directory permissions	DPI / SDI	DPK / SDK	DPC / SDC

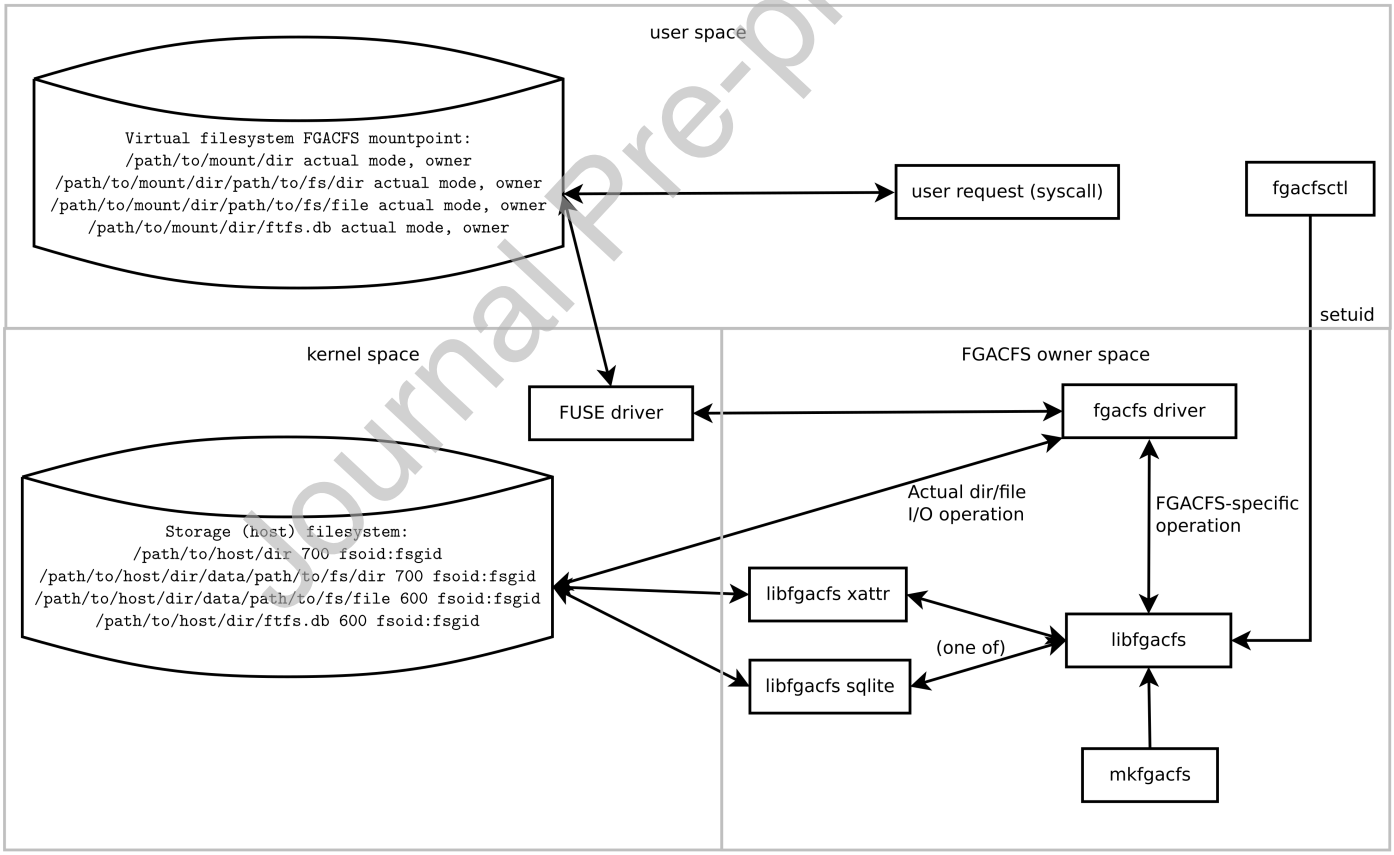

Figure 1: General overview of communication between FGACFS modules, user, and system.

6. Interoperability with POSIX syscalls

Now let us examine when and how fine-grained permissions are converted and checked. During the execution of file system operations, the FUSE-module functions have access to owner, group, and id of any process that is going to perform a given operation. Using this data, it is possible to check whether the requested operation is allowed. In the case of the process being run by a privileged user, it is possible to learn the name of the executable file by checking /proc. Otherwise, permission checking for applications is not performed.

Only a small number of operations, such as *readlink*, *unlink*/*rmdir*, can be validated using a simple permission check for a particular action.

The *open* call requires to check read and write (append and modify content) permissions depending on the opening flag. At the same time, *read* and *close* calls do not need permission checking at all, since they can not be invoked without executing *open* first.

At the same time, it is necessary to check the rewrite and append permissions during each *write* call, since the outcome depends on what is actually done. In other words, it is necessary to compare the file size and file write position. Just checking the `O_APPEND` flag on *open* is not suitable for two reasons. First, common file actions like copying or saving are implemented in most applications without using `O_APPEND`, even though application writes are performed at the end of an initially-empty file. Checking `O_APPEND` solely on *open* in case of allowed append but not rewrite would disable such actions, even if they do not change any file content. Second, according to POSIX (but not linux), a file opened with the `O_APPEND` flag still could be rewritten by a *pwrite* call.

Similarly to the previous case, for each *truncate* call, a permission to truncate or to append should be checked, because such a call can either result in reduction or in enlargement of file size. The file size can even stay unchanged: for example, truncating to zero is often performed right after new file creation. The latter should be always allowed.

It should be mentioned that the absence of a delete permission does not imply prohibiting modification of file content in case of allowed rewriting or truncation. This could be useful, for example, in a case when removing and re-creating a file will result in changing the file owner and/or file ACLs, while truncating to zero and rewriting would keep this data untouched. At the same time, if a user is not allowed to delete, truncate and rewrite a file, then they would not be able to remove existing file content (even if it was appended by this user).

Creating file system nodes (files, directories, FIFOs, etc) requires checking two permission classes — permissions for creation and permissions for addition of the corresponding nodes. If creation is permitted, then the new node will be owned by user, if only addition is permitted the new node will have the same owner as a parent directory.

The operation of owner change is regulated by the following permissions:

1. permission to change owner to any uid;
2. permission to change group to any gid;
3. permission to change owner and group to parent directory owner and group.

The *rename* call can refer to two different operations. The first one is renaming a file or a directory, and the second one is moving it from one directory to another. The DMV and FMV FGACFS permissions refer to the former case. The latter case is regulated by permissions for creating the corresponding object in the destination directory and permissions for reading and deleting the source object. If any of these permissions are denied, the operation is not carried out.

In the *stat* call, it is expected that the usual POSIX *rwx* permissions would be returned, thus the `fgacfs` module has to convert FGACFS permissions to POSIX. This is done as follows:

1. For every user or group the POSIX Read, Write or eXecute (*rwx*) permissions are set in accordance with the FGACFS read, write and execution permissions.
2. If node and process owner coincide:
   - POSIX user *rwx* are assigned according to the FGACFS permissions of process uid and gid.
   - POSIX group *rwx* are assigned according to the FGACFS permission set for process gid.
   - POSIX other *rwx* are assigned according to the FGACFS “all other users” permissions.
3. If node and process owner do not coincide, but node group and process group do, then:
   - POSIX user *rwx* are assigned according to the FGACFS permission set for the node owner.
   - POSIX group *rwx* are assigned according to the FGACFS permission set for process uid and gid.
   - POSIX other *rwx* are assigned according to the FGACFS “all other users” permissions.
4. If neither node owner nor node group coincide with the respective process counterparts, then:
   - POSIX user *rwx* are assigned according to the FGACFS permission set for node owner.
   - POSIX group *rwx* are assigned according to the FGACFS permission set for node group.
   - POSIX other *rwx* are assigned according to the FGACFS permission set for process uid and gid permissions.

The described algorithm ensures an interpretation of FGACFS execution permission that is correct from the standpoint of a POSIX application. FGACFS is unable

to perform an execution permission check. It is done outside of the file system, using the "x" bit for owner, group and others. The outcome depends on the relationships between owner and group of a file. The algorithm sets the "x" bit in such a way that the result of an outside check complies with FGACFS execution permission given for the user executing the process.

We employ a similar approach to handle both read and write bits. However, in each of these bits we have to fit a collection of different permissions of the same kind (see Tables 1, 2). To handle this problem, we adopt the following approach: the *stat* call will indicate that a read operation is permitted if at least one of its kind is permitted. Consequently, denied status will be returned if all read operations are not allowed. The write bit is treated similarly.

Out of all system calls, the *chmod* call is the hardest one to impart even remotely POSIX-compatible behaviour. Thus, the general idea of our approach is to determine which permission is being changed and the targeted user. The latter is performed using the technique that is inverse to *stat* call processing. The permission change is processed as follows:

1. if r, w or x operations are being allowed, clear all respective FGACFS class deny permissions and allow all respective FGACFS class permissions;
2. if these operations are being disallowed, deny all respective FGACFS class permissions.

This way the *stat* call after the *chmod* would produce consistent result (rwx bits would be set or unset as defined).

7. Caching

Preliminary evaluation has demonstrated unacceptable performance of our prototype. A detailed investigation revealed that the performance bottleneck was related to the retrieval of extended permissions. This was true for both database and extended attributes access.

We have used Sysprof [17] to study the performance bottleneck in FGACFS. We have found that libsqlite contributes more than 60% of the overall processing time in the db version, while kernel calls contribute around 30%. The share of FGACFS itself is less than 3%. The xattr version was different: 80% were kernel calls and 5% were FGACFS operations. Therefore, it was not the FGACFS algorithms that created the bottleneck, but permission storage and retrieval.

In order to address this, we have implemented caching of permission information using the LRU algorithm. The ACL list for each individual file may be very large, since it can contain permissions and restrictions for a large number of users and groups. Therefore, caching the whole permission list is not reasonable. Instead, we have decided to keep only accessed permissions for every file.

We have decided to store not only permissions, but the results of conversion of FGACFS permissions to POSIX (that are displayed by the *stat* call) as well. Moreover, we cache DEX permissions (an analogue of the POSIX "x" permission for a directory) for all ancestors up to the FGACFS root. Checking this permission involves a very costly procedure that requires to check the execution permission of each directory in the whole tree. Essentially, it requires a recursive bottom-up traversal. All this information is requested every time any user browses a directory and, therefore, it is reasonable to cache it.

However, this part of the cache frequently requires invalidation: any change in the directory permission list or changing the directory owner or group invalidates cache data of all descendants. Other parts of cache are updated in real-time and can be invalidated only due to `fgacfsctl` actions.

The following data structures were used to store permission record information:

1. Hash table that uses the file system-relative path of an object as a key. This path is the only file system object identifier provided by FUSE. The value is a pointer to the position in a dequeue which stores cache entries.
2. Dequeue of cache entries. Each entry contains the following:
   - a pointer to the "owning" hash table entry. In case of cache overflow, this pointer is used to delete excess cache entries from the table.
   - owner, group, and inheritance data for the corresponding file.
   - a red-black tree for storing permission record information. The tree is organized as follows: a combination of the FGACFS permission category and uid, gid or executable name is used as key and status as value. Note that for some permission categories (e.g. "for all") the second element of the pair is empty. The status lists permissions from Tables 1 and 2 that are represented by two bit masks (one for allow and one for deny).
   - a red-black tree for storing:
     (a) POSIX permissions (the ones exported from FGACFS via the *stat* call)
     (b) whether all ancestors of a given object (residing inside FGACFS) have a DEX permisson.

     Each tree is organized in the following way: a (uid, gid) combination is a key and a list of uid secondary groups, a search permission, and POSIX permissions are values. If there is a mismatch between the current and cached user group list (i.e. the uid membership was altered since the last caching), then the record becomes invalid. It is worth mentioning that

we use these trees only in cases when checking executable permission is not required, which is always true for directory browsing.

These structures are used to create a classic implementation of an LRU cache. It operates as follows:

- if there is no element in dequeue that corresponds to the file system object, then the required information is retrieved from the primary storage (either database or extended attributes). Next, a new element is placed in the beginning of dequeue. If the number of stored entries exceeds the threshold — maximum cache size — the last entry of dequeue is deleted.
- if the requested object is found in the cache it is moved to the beginning of the dequeue.
- Having obtained the dequeue entry, the corresponding red-black tree is probed to obtain either FGACFS permissions or both search and POSIX permissions. If the required record is not found, it is read from the primary storage and inserted into the corresponding tree. Otherwise, they are returned.

Any caching scheme results in additional memory footprint. In our case the major part of memory is spent on storing the file system-relative path of objects. In overall, we expect the cache to require 1KiB per object on average while not using access control for applications. Our estimates are around 5-10 KiB for the case with several tracked applications. These calculations lead to the following conclusion: storing thousands of file system entries would require a cache of megabytes or few dozens of megabytes. These numbers are acceptable for most modern desktop systems.

8. Evaluation

In order to examine the efficiency of our prototype and assess its suitability for practical application, we have performed an experimental evaluation. We have compared the FGACFS prototype and several popular file systems (ext4, NTFS-3G, NFS). The following hardware and software configuration was used:

- CPU AMD Turion(tm) 64 X2 Mobile Technology TL-58 1.9GHz
- SMARTBUY 2.5" Leap 128 GiB SATA III MLC SB128GB-LP-25SAT3
- 2GiB RAM
- GNU/linux (Fedora 28), kernel 4.17.7-200.fc28.i686
- SQLite 3.22.0
- ntfs-3g 2017.3.23 integrated FUSE 28
- nfs-utils 2.3.2
- EXT2FS Library 1.44.2
- FUSE 2.9.7

In our implementation, the *stat* call is performed very often and it is very computationally expensive. Thus, first of all, we measured run times of *ls -l* executed inside mounted FGACFS file system directory containing 8192 files for three different scenarios:

1. The files are contained in the root directory of the file system without any permission inheritance.
2. The files are contained in a directory which is the lowest level of ten-level nesting directory hierarchy. These files inherit a single permission record from the root directory.
3. Similarly to the previous case, but files inherit 17 permission records (11 for users, 5 for groups, 1 for all users) from the root directory.

All scenarios were evaluated in three cases — without caching, with cache size of 2048 entries (partial) and with cache size of 10240 entries (full). We have compared them with the native ext4, ntfs-3g FUSE module, and with NFSv4 mount with and without attribute caching.

The following evaluation procedure was employed. In our calculations we used the data from 5 runs. However, we had carefully selected data for the *stat* call experiment — we had to exclude the first call, because in this case it is performed on uncached data.

The first run was evaluated separately, also using data from 5 trials. To ensure clean caches we have unmounted and mounted the file system for each run.

For both cases, we have computed an average and 95% confidence intervals, assuming normal distribution of measurement errors. The results are presented in Table 4. In this table, the first column specifies the examined file system, the second and the third — the employed inheritance model and caching policy respectively. The last two columns describe the performance of the resulting configuration (measured in entries per second) for uncached and cached runs. If the configuration does not employ caching, then we put "N/A" in the "first run" column.

The second group of experiments was an input-output speed test conducted using Bonnie++ [18]. We have decided to employ the third scenario (nested directory with multiple access control record inheritance), which we deemed to be most underperforming one.

We have compared two FUSE I/O modes: with direct I/O and without. Turning direct I/O off allows the kernel to cache data and access file content bypassing FGACFS. It is possible not to employ direct I/O for FGACFS, since file sizes are known in advance and files located in the directory on the host file system (containing FGACFS content) should not be modified bypassing the driver. We have compared "xattr" and "db" versions to ext4, NTFS-3g and NFSv4.

We think there are no reasons for FGACFS ACL inheritance and the quantity of permission records to affect the I/O speed, since it is redirected to the underlying file system (ext4).

For this series of tests, we have used samples of 20MB for byte-oriented I/O and samples of 4GB (set to be two times larger than available RAM) for block-oriented I/O testing. The results of our experiments are presented in Table 5.

We can draw the following conclusions from this data:

1. The database version is significantly slower than the extended attributes version.
2. Using even a partial cache significantly speeds up permission checking. Surprisingly, caching helps even during the first directory browsing query (ls).
3. FGACFS appears to be much slower than ext4 without ACLs and NFS, despite the absence of networking. However, it demonstrates performance of the same order as NTFS-3g. These facts suggest that the main source of performance drop is the overhead of the userspace implementation, rather than the permission model itself. It is essential to mention that the NTFS-3g driver does not check NTFS access permissions. Instead, file system objects are mapped as if they belong to a single POSIX user. POSIX permissions are defined using the mount parameter, similarly to the umask mechanism. Nevertheless, NTFS access permissions are set and inherited automatically during file and directory creation, according to the Windows rules for the corresponding operations.
4. Disabling direct I/O greatly increases read speed and significantly slows down writes. Therefore, the decision whether to turn it on or not should be delegated to user.
5. The table demonstrates that FGACFS with full cache can successfully compete with NTFS-3g — a file system used in practice. FGACFS shows good I/O speed and ability to read a directory containing over 100 entries in less than 200 ms. Thus, we deem that FGACFS demonstrated sufficient performance to be used practically as well (at least the extended-attributes version). In case of full cache, the main obstacle to using the database-based version is the first call — it may take too much time since it is always uncached.

9. Known Issues

Our tool is not and could not be made fully POSIX compatible. For example, setting and unsetting the same permission via *chmod* invocation would not revert the file system object to its original state. However, while constructing it, we aimed to achieve file system behavior that is similar to POSIX at least in basic use cases.

Note that copying a file even within a single FGACFS instance will lead to loss of permissions. This will happen due to the nature of our permission model: copying a file is basically a creation of a new file. Therefore, new files will inherit destination directory permissions. However, in a typical case where a user does not have any opportunity to manage permissions and inheritance rules, this behavior will not lead to any issues.

In other situations, the `fgacfsctl` tool can be used to dump own and inherited permissions of objects separately in the `fgacfsctl` command line format. Consequently, it becomes easy to save and restore (and thereby copy) permissions automatically. Another way to preserve permissions is to copy extended attributes with standard tools. The `fgacfs` FUSE driver lists, sets and gets own (not inherited) permissions of each mounted file system object as extended attributes (not related to extended attributes used to store permissions in a host file system). Copying an entire subtree with `cp` or `rsync` in xattr-preserving mode while also starting from a directory with no inherited permissions preserves FGACFS ACLs if the user is permitted to change inheritance and permission attributes for the destination directory. In case of copying from the middle of the permission inheritance tree, the use of `fgacfsctl` is required to preserve inherited permissions.

This driver restriction comes from the fact that copying is performed in an object-by-object manner (and FUSE does not provide any information regarding the root directory of the directory tree being copied). Therefore, listing inherited permissions for any given object will lead to them being copied, forfeiting inheritance.

It is necessary to mention that copying MAC labels and NFSv4 permissions requires additional tricks when using `cp` or `rsync`. For example, `cp` was specifically adapted to copy SElinux content labels, while `rsync` still has some problems in transferring NFSv4 ACLs and therefore requires tweaking [19].

However, *rename* call is free from this flaw when used to move objects from one FGACFS directory to another. It can correctly transfer permissions if a user is allowed to change them in the destination folder.

A more serious issue may arise due to the behaviour of several file editors (e.g. Midnight Commander Edit, Open Office and its forks). These editors create a temporary file in the directory of the edited file. Consequently, allowing to write (modify) a file, but not to create new ones would lead to an unexpected error on an attempt to edit a file.

Finally, process executables are obtained by parsing the `/proc` file system. This can be done if and only if FGACFS is executed as root.

Usability is another reason for concern: a large number of permission flags requires significant cognitive effort from an administrator. These flags are based on system calls, and it is necessary to map them into user actions while simultaneously deciding whether to allow or deny each of them. Although in theory most of FGACFS permissions are independent, their common combinations have to be used in practice. Examples of such combinations are shown in Table 6. Users are able to write shell scripts to

Table 4: Evaluating stat performance (in entries per second): running ls for 8192 files.

File system	inheritance	cache	first run	subsequent runs
FGACFS (xattr)	no	no	N/A	725 ± 1
FGACFS (xattr)	no	partial	2080 ± 40	2090 ± 5
FGACFS (xattr)	no	full	2036 ± 29	3388 ± 20
FGACFS (xattr)	single record	no	N/A	29 ± 1
FGACFS (xattr)	single record	partial	562 ± 15	539 ± 2
FGACFS (xattr)	single record	full	542 ± 30	2178 ± 7
FGACFS (xattr)	multiple record	no	N/A	29 ± 1
FGACFS (xattr)	multiple record	partial	542 ± 15	506 ± 2
FGACFS (xattr)	multiple record	full	552 ± 8	2139 ± 7
FGACFS (db)	no	no	N/A	256 ± 1
FGACFS (db)	no	partial	934 ± 5	943 ± 1
FGACFS (db)	no	full	925 ± 7	3368 ± 13
FGACFS (db)	single record	no	N/A	7.5 ± 0.1
FGACFS (db)	single record	partial	422 ± 7	445 ± 1
FGACFS (db)	single record	full	428 ± 14	2310 ± 9
FGACFS (db)	multiple record	no	N/A	7.3 ± 0.1
FGACFS (db)	multiple record	partial	408 ± 14	443 ± 1
FGACFS (db)	multiple record	full	425 ± 21	2180 ± 9
ext4	n/a	default	N/A	56174 ± 1873
NFSv4	no	default	N/A	53931 ± 1338
NFSv4 (w/o attr. cache)	no	no	N/A	723 ± 3
NTFS-3g	no ACL	default	N/A	5219 ± 68

Table 5: I/O performance evaluation

operation	write		rewrite	write		seek
type	byte	block	block	byte	block	random
unit	KiB/sec	MiB/sec	MiB/sec	KiB/sec	MiB/sec	ops
ext4	438 ± 5	147 ± 7	61 ± 3	849 ± 38	104 ± 2	6737 ± 156
NTFS-3g	6 ± 1	21 ± 5	28 ± 2	805 ± 36	91 ± 5	2363 ± 322
NFSv4	320 ± 120	57 ± 14	48 ± 8	482 ± 197	120 ± 35	2739 ± 521
FGACFS (db)	5 ± 0.1	25 ± 0.1	21 ± 0.2	630 ± 27	101 ± 0.4	2378 ± 12
FGACFS (xattr)	4 ± 0.1	24 ± 0.1	20 ± 0.1	678 ± 13	101 ± 0.2	2369 ± 5
FGACFS (db), dio	16 ± 0.1	73 ± 1	26 ± 0.8	17 ± 0.1	87 ± 1	3174 ± 18
FGACFS (xattr), dio	16 ± 0.1	71 ± 1	24 ± 0.8	17 ± 0.1	86 ± 1	3221 ± 15

set them.

10. Conclusion

In this paper we have presented fine-grained access control for the *nix file system. We started by motivating the need for such a tool and surveying the state of the art. Then, we have described our extension of POSIX ACLs — the Extended Access Rights list. This concept formed the basis of the software implementation that we have constructed to evaluate the feasibility of our approach.

The proposed access control model offers significantly more functionality to manage permissions than models implemented by all contemporary tools. First of all, it provides a broader list of enforceable access rights that is implemented in a single tool, thus, allowing to control both system users and programs simultaneously. Next, it offers an extended inheritance model that separates inheritance of file and directory permissions. Finally, our inheritance model allows not only to inherit, but to copy permissions during file or directory creation. The list of FGACFS capabilities compared to other common file systems is shown in Table 7. It is necessary to mention that solutions described in the Related Work section are either MAC-related (and therefore not analyzed here) or employ one of the considered models (POSIX ACL, NFSv4, NTFS).

For the first time such an extension of the ACL model is implemented in userspace — an approach that simplifies administration procedures and offers increased portability.

Our system is implemented as a FUSE module with two different approaches to storing extended permissions — using an embedded database and using file system extended attributes. Initial tests have demonstrated unacceptable performance, which forced us to design a permission caching scheme. We have successfully resolved the performance bottleneck by using it.

Experiments have demonstrated that the database version is slower than the one based on extended attributes. However, the former is more portable. Furthermore, the database version could be adapted to be web-server accessible (for example, using PHP code to provide `libfgacfs`-compatible functionality).

We believe that the performance of at least the extended attributes-based version is sufficient to be used in practice at least in small user groups.

Availability of data and materials. All source code is available at

`https://github.com/lovyagin/fgacfs`.

Competing interests. The authors declare that they have no competing interests.

Funding. This work is partially supported by Russian Foundation for Basic Research (RFBR), grant 19-01-00470 A.

Author contributions. Nikita Lovyagin has formulated the addressed problem, proposed an extension the of access control model and improved the initial implementation of the file system; Roman Daineko has created the initial implementation of the file system (without caching and inheritance); Kirill Smirnov has perfomed testing and worked on quality of code; George Chernishev has performed the evaluation.

Acknowledgements. This work is partially supported by Russian Foundation for Basic Research (RFBR), grant 19-01-00470 A.

References

[1] Posix.1-2008, the open group, 2016. URL: `http://pubs.opengroup.org/onlinepubs/9699919799.2016edition`.

[2] Posix acl, 2016. URL: `http://pubs.opengroup.org/onlinepubs/9668899/chap7.htm`.

[3] Ntfs, 1993. URL: `http://www.ntfs.com/ntfs-permissions-file-advanced.htm`.

[4] S. Smalley, C. Vance, W. Salamon, Implementing selinux as a linux security module, NAI Labs Report 1 (2001) 139.

[5] Apparmor, 2013. URL: `https://help.ubuntu.com/lts/serverguide/apparmor.html`.

[6] M. szeredi, file system in user space, 2016. URL: `https://github.com/libfuse/libfuse/releases`.

[7] A. Rajgarhia, A. Gehani, Performance and extension of user space file systems, in: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, ACM, New York, NY, USA, 2010, pp. 206–213. doi:`10.1145/1774088.1774130`.

[8] K. A. Kumar, A. Grünbacher, G. Banks, Implementing an advanced access control model on linux, in: Linux Symposium, Ottawa: Citeseer, 2010, pp. 19–32.

[9] Nfsv4, 2000. URL: `https://tools.ietf.org/html/rfc3010`.

[10] Z. Wilcox-O'Hearn, B. Warner, Tahoe: The least-authority filesystem, in: Proceedings of the 4th ACM International Workshop on Storage Security and Survivability, StorageSS '08, ACM, New York, NY, USA, 2008, pp. 21–26. doi:`10.1145/1456469.1456474`.

[11] S. Kleiman, N. Appliance, N. Storage, Dafs: A new high-performance networked file system., ACM Queue 1 (2003) 70–79.

[12] Z. Xiong, T. Guo, C. Zhu, W. Cai, L. Cai, Enterprise file-sharing system with lightweight attribute-based access control, University Politehnica of Bucharest scientific bulletin series C—electrical engineering and computer science 80 (2018) 15–26.

[13] V. Goyal, O. Pandey, A. Sahai, B. Waters, Attribute-based encryption for fine-grained access control of encrypted data, in: Proceedings of the 13th ACM conference on Computer and communications security, Acm, 2006, pp. 89–98.

[14] S. Bugiel, S. Heuser, A.-R. Sadeghi, Flexible and fine-grained mandatory access control on android for diverse security and privacy policies, in: Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13), 2013, pp. 131–146.

[15] M. H. Kang, J. S. Park, J. N. Froscher, Access control mechanisms for inter-organizational workflow, in: Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies, SACMAT '01, ACM, New York, NY, USA, 2001, pp. 66–74. doi:`10.1145/373256.373266`.

[16] Extended attributes, 2016. URL: `http://man7.org/linux/man-pages/man7/xattr.7.html`.

[17] Sysprof, 2015. URL: `http://www.sysprof.com`.

[18] Bonnie++, 1999. URL: `http://www.coker.com.au/bonnie++`.

[19] How to preserve nfs v4 acls via extended attributes when copying file, 2018. URL: `https://access.redhat.com/solutions/3628891`.

Table 6: Examples of some permission combinations

File access type

Permission	FRA	FXA	FRD	FSL	FRW	FAP	FTR	FCA
Read attributes only	+	+						
Read content	+	+	+	+				
Read and modify, fixed size	+	+	+	+	+			+
Read and append, no modify	+	+	+	+		+		+
Read and modify	+	+	+	+	+	+	+	+

Directory access type

Permission	DRD	DRA	DXA	DEX	DAF	DAD	DAL	DOF	DOD	DOL	DCA	FSP	DSP
Read directory content	+	+	+	+									
Add files (same owner)	+	+	+	+	+		+				+		
Add files and directories	+	+	+	+	+	+	+				+		
Create own files	+	+	+	+				+		+	+		
Create files and directories	+	+	+	+				+	+	+	+		
Create and add files	+	+	+	+				+		+	+	+	
Create and add files and directories	+	+	+	+				+	+	+	+	+	+

Parent directory configuration for its permission inheritance

Transmission flag	SFI	SDI	SFK	SDK	DPC	FPC	SDC	SFC
Permissions should be inherited by all subobjects	+	+						
Permissions should be inherited and copied on inheritance remove	+	+	+	+				
Permissions should be copied to all subobjects					+	+	+	+

Table 7: Key FGACFS capabilities compared to other common file systems

Capability	POSIX ACL	NFSv4	NTFS	FGACFS
Separate file and directory permission lists (applicable only for systems supporting inheritance)	non-applicable	NO	NO	YES
Separate file and directory creation permissions	NO	YES	YES	YES
Separate permissions for creation of different file types	NO	NO	non-applicable	YES
Automatic unification of all filesystem object owner	NO	NO	NO	YES
Permission inheritance	NO	NO	YES	YES
Automatic copy of permissions to child objects	NO	YES	partial	YES
Separate permissions for file rewrite and for append	NO	YES	YES	YES
Separate permission to file rewrite and file truncate	NO	NO	NO	YES
ACLs for applications	NO	NO	NO	YES

Nikita Lovyagin is an associate professor in the Saint Petersburg University, Saint Petersburg, Russia. He received the M.Sc. and Ph.D. degrees from Saint Petersburg University, Russia in 2008 and 2012 re- spectively. His research interests include numerical simulation, parallel programming, operating systems. In the university he teaches operating systems and *NIX courses. He has over 10 papers to his name. E-mail: n.lovyagin@spbu.ru.

George Chernishev is an assistant professor in the Saint Petersburg University, Saint Petersburg, Russia and senior lecturer at National Research University Higher School of Economics, Saint Petersburg, Russia. He received the M.Sc. degree from Saint Petersburg University, Russia in 2007. His research interests include physical design, query processing in databases and operating systems. He is a PC member of a number of database-focused conferences and a member of the Association for Computing Machinery. He has over 50 papers to his name. E-mail: chernishev@gmail.com.

Kirill Smirnov is an engineer at the Saint Petersburg University, Saint Petersburg, Russia and a software developer at Lanit-Tercom. He received the M.Sc. degree from Saint Petersburg University, Russia in 2007. His research interests include databases, networking and embedded operating systems. Kirill has more than ten years of industrial *nix software development experience. Also, he is a PC member of SEIM conference series for young researchers. He has over 25 papers to his name. E-mail: k.k.smirnov@spbu.ru.

Roman Dayneko studied software engineering at the Saint Petersburg University, Russia, and obtained his B.Sc. degree under the supervision of Professor Lovyagin in 2016. E-mail: dayneko3000@gmail.com.