Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Filesystem with Interactive Access Control for Linux

Bachelor Thesis

2024

Fedir Kovalov

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Filesystem with Interactive Access Control for Linux

Bachelor Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	RNDr. Jaroslav Janáček, PhD.

Bratislava, 2024
Fedir Kovalov

94283494

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Fedir Kovalov
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Filesystem with Interactive Access Control for Linux
Súborový systém s interaktívnym riadením prístupu pre Linux

Anotácia: Tradičné mechanizmy riadenia prístupu v operačných systémoch povoľujú rovnakú úroveň prístupu všetkým procesom bežiacim v mene toho istého používateľa. Toto typicky umožňuje škodlivým procesom čítať a/alebo modifikovať všetky údaje prístupné používateľovi, ktorý spustil zraniteľnú aplikáciu. Dá sa to riešiť použitím rôznych mechanizmov povinného riadenia prístupu, no tieto sú často náročné na konfiguráciu a zriedkavo sa používajú v bežných scenároch orientovaných na používateľa. Táto práca sa zameriava na návrh a implementáciu vrstvy súborového systému, ktorá rozhodnutie povoliť alebo zakázať prístup k objektu súborového systému konkrétnym procesom deleguje na používateľa.

Cieľ:
- analyzovať problém a navrhnúť riešenie
- implementovať riešenie využitím FUSE
- otestovať riešenie a demonštrovať jeho prínos

Vedúci: RNDr. Jaroslav Janáček, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 31.10.2024

Dátum schválenia: 31.10.2024 doc. RNDr. Dana Pardubská, CSc.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Fedir Kovalov
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Filesystem with Interactive Access Control for Linux

Annotation: Traditional access control mechanisms in operating systems allow the same level of access to all processes running on behalf of the same user. This typically enables malicious processes to read and/or modify all data accessible to the user running a vulnerable application. It can be dealt using various mandatory access control mechanisms, but these are often complicated to configure and are rarely used in common user oriented scenarios. This thesis focuses on design and implementation of a filesystem layer which delegates the decision to allow or deny access to a filesystem object by a specific process to the user.

Aim:
- analyse the problem and design a solution
- implement the solution using the FUSE framework
- test the solution and demonstrate its benefits

Supervisor:	RNDr. Jaroslav Janáček, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	31.10.2024

Approved: 31.10.2024 doc. RNDr. Dana Pardubská, CSc.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Abstrakt

Tradičné mechanizmy riadenia prístupu v operačných systémoch povoľujú rovnakú úroveň prístupu všetkým procesom bežiacim v mene toho istého používateľa. Toto typicky umožňuje škodlivým procesom čítať a/alebo modifikovať všetky údaje prístupné používateľovi, ktorý spustil zraniteľnú aplikáciu. Dá sa to riešiť použitím rôznych mechanizmov povinného riadenia prístupu, no tieto sú často náročné na konfiguráciu a zriedkavo sa používajú v bežných scenároch orientovaných na používateľa. Táto práca sa zameriava na návrh a implementáciu vrstvy súborového systému, ktorá rozhodnutie povoliť alebo zakázať prístup k objektu súborového systému konkrétnym procesom deleguje na používateľa.

Kľúčové slová: riadenie prístupu, súborové systémy, FUSE, súhlas používateľa, najmenšie oprávnenie, oprávnenia, udeľovanie oprávnení, riadenie prístupu riadené používateľom.

Abstract

Traditional access control mechanisms in operating systems allow the same level of access to all processes running on behalf of the same user. This typically enables malicious processes to read and/or modify all data accessible to the user running a vulnerable application. It can be dealt using various mandatory access control mechanisms, but these are often complicated to configure and are rarely used in common user oriented scenarios. This thesis focuses on design and implementation of a filesystem layer which delegates the decision to allow or deny access to a filesystem object by a specific process to the user.

Keywords: access control, filesystems, FUSE, user consent, least-privilege, permissions, permission granting, user-driven access control.

Contents

List of Figures

List of Tables

Todo list

Introduction

In modern operating systems, access control mechanisms are fundamental to ensuring the confidentiality, integrity, and availability of system resources. These mechanisms dictate how users and processes interact with system objects such as files, directories, and devices. However, traditional access control models, such as the discretionary access control (DAC) employed by Linux and other Unix-like systems, operate under the assumption that all processes running under the same user account should have the same level of access to system resources. While this simplifies user management and permissions, it can introduce significant security risks.

The problem arises when a process or application running under a user's account becomes compromised. In such cases, the malicious code or exploit can leverage the user's existing permissions to access or modify sensitive data, potentially leading to data breaches or other security incidents. This fundamental limitation of traditional access control mechanisms underscores the need for a more granular and dynamic approach to file system access control.

Over the years, various mandatory access control (MAC) mechanisms, such as SELinux (Security-Enhanced Linux) and AppArmor have been developed to address these limitations. These systems enforce access control policies at a more granular level, often based on labels or rules defined by system administrators. While these mechanisms are effective in certain scenarios, they are generally complex to configure and require significant expertise to maintain. As a result, they are rarely adopted in common user-oriented environments, where simplicity and ease of use are paramount.

In this thesis we introduce our approach to file system access control that empowers users to make real-time decisions about which processes or applications should have access to specific file system objects. By integrating an interactive decision-making layer into the file system, this solution aims to bridge the gap between the security benefits of MAC mechanisms and the simplicity required for widespread adoption. The proposed system delegates access control decisions to the user, enabling them to grant or deny access to individual processes or applications on a per-object basis. This approach not only enhances security but also maintains the flexibility and usability that are critical for user-oriented systems.

The rest of this thesis is organised as follows: Chapter 1 and chapter 2 provides a

review of existing access control mechanisms and their limitations. Chapter 3 outlines the design objectives, architecture, and the interactive component of the proposed file system layer. Chapter 4 describes the implementation process, including the tools and techniques used to develop the system. Finally, in chapter 5 we present experimental results, evaluate the performance, security benefits and limitations of the proposed solution, and discuss the potential for further development.

Chapter 1

Filesystem access control on Linux

1.1 Traditional UNIX Filesystem Access Control Policies

By default, UNIX-like operating systems only provide simplistic Discretionary Access Control (DAC) policies whose objects are files, and subjects are users.

The policy used by traditional UNIX systems is based on the concepts of *file owner*, *group of a file*, and *others*. For each file, the access rights for these three categories can be specified independently using a so-called access mode. The access mode is a bitmask which specifies whether the file owner, the group of the file, and others have read, write, or execute permissions.

Each process has it's own *Effective User ID* (EUID), that identifies the user that initiated it. When a process tries to access a file, the kernel checks the access mode of the file, and grants or denies access based on the following rules:

- If the process's effective user ID matches the file owner, the file owner's access mode is used.
- Otherwise if the process belongs to the group of the file, then the group's access mode is used.
- If neither condition holds, others' access modes are applied.

The access mode is stored in the file's inode, and is set by a process with the file owner's user ID using the `chmod` system call. The file owner is the user who created the file, and can be changed using the `chown` system call by a process with the effective user ID of a superuser. The group of a file is set to the group of the file owner when the file is created, and can also be changed using the `chown` system call by a process with the effective user ID of a superuser.

Later, a feature called Access Control Lists (ACL) was introduced to many UNIX-like operating systems and eventually included in the POSIX standard. ACLs provide the ability to control file permissions of specific users, rather than just owner, group and others. Similar to the classic UNIX access control policies, only processes running with the user ID that matches the owner user ID of a file can change its ACLs.

1.2 The Inherent Flaw of User-Centric Access Control

Although this kind of access control solutions has been proven to be helpful in multi-user environments, it is obviously insufficient to protect against an attack performed by a process initiated by the same user.

The fundamental weakness of the traditional UNIX DAC model, and even its extension with ACLs, lies in its reliance on user identity as the primary access control decision point. While effective at separating access between different users, it provides little to no protection within a user's own account. This deficiency is particularly problematic in modern computing environments where a user's processes are increasingly complex and often involve downloaded or third-party code.

This vulnerability stems from the "all or nothing" nature of user ownership. A process running with user's EUID inherits all of user's privileges, treating all files they own as equally accessible. There's no way to restrict a specific process, even one initiated by the user themselves, from accessing certain files or performing certain operations.

These limitations highlight the need for more sophisticated access control mechanisms that go beyond simple user identity and consider the context and trustworthiness of the process attempting to access a resource. Mandatory Access Control (MAC) and sandboxing technologies are emerging solutions aiming to address these shortcomings by introducing finer-grained control over process privileges and resource access. The following chapter will explore these alternatives in detail.

Draft note: Talk more about the threat model?

Chapter 2

Current solutions, and why they won't suffice

2.1 MAC mechanisms

Many Linux OS ship with additional Mandatory Access Control (MAC) mechanisms (e.g. AppArmor, SELinux) that allow to restrict the usage of file system objects by specific programs.

Draft note: Explain how exactly can they do that? (It seems irrelevant to the overall topic)

Unfortunately, these mechanisms require a considerable amount of knowledge and effort for the user to manage them, which makes them infeasible for most single-user environments.

2.2 FGACFS

In Lovyagin et. al. 2020 [1] authors propose and implement a so called FGACFS file system that extends traditional UNIX access control policies with far more sophisticated and granular system. This also includes the ability to restrict access on per-program basis. However, due to the sheer variety of options and configurable parameters, this approach still falls short when it comes to ease of use and user-friendliness.

Additionally, all the above solutions share a significant drawback: they necessitate user intervention to secure files, even when those files are never accessed. For instance, if access to a file system object is denied (allowed) for all programs by default and only allowed (denied) for specific ones, granting (revoking) access for new programs requires users to modify access permissions proactively.

While some solutions offer automatic inheritance or assignment of rules and access control policies, they still need extensive manual configuration. Even if inheriting all

access permissions from a default value were practical, installing new programs would always necessitate updating rules to adhere to the principle of least privilege.

Another problem of these solutions, is that their policies are granted forever and the user is never informed about the actual usage of those permissions, which makes them more vulnerable to attacks by proxy. For example, if the program `cat` is allowed to read contents of the file `~/secrets/text.txt`, malicious program may execute `cat ~/secrets/text.txt > ~/stolen-text.txt` command at any time, without any warning and regardless of whether the malicious program has access to `~/secrets/text.txt` or `~/stolen-text.txt`. If the user only granted read permissions to `cat` when they are actually using the program themselves, such attack could likely be avoided.

2.3 Containerisation

Another solution to consider, is using containerised software distribution, like Flatpak [2], Snapcraft [3] or AppImage [4]. Those types of package distribution systems either use Linux feature called *namespaces* or leverage MAC mechanisms to isolate software from the rest of the system. Aside from solving common dependency management problems, this approach also allows some capabilities of the distributed software to be restricted, like access to camera, hardware devices, but, most importantly, file system objects.

However, because the developer of the distributed software is responsible for defining the permissions that his own program needs, it often leads to programs having excessive privileges after installation[1] without any notification of the user.

Additionally, it is a responsibility of the software developer to choose the distribution method, and despite containerised software getting more and more popular, there are still plenty of programs that can only be installed using traditional methods, that do not offer any mechanisms for restricting file system access.

Furthermore, some software is impractical to sandbox. For example, because of the FlatPak's design, CLI software has to be run with `flatpak run` command and has to use often long and hard-to-remember package names, which may appear rather cumbersome for most users.

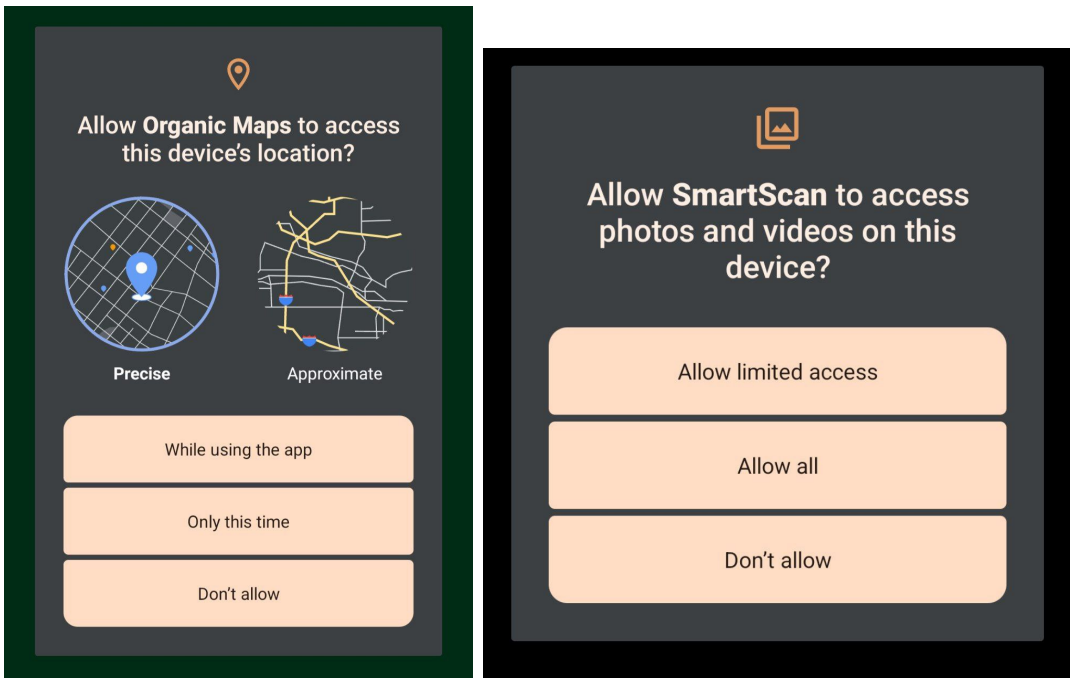
2.4 Android

Another, similar solution can be found in the Android operating system. Here, all apps are sandboxed by default. But Android does way more than Flatpak: it adds an

[1] It is important to mention, that although this flaw remains unmitigated, the analysis made by Dunlap et al. 2022 [5] shows that most package maintainers actively attempt to define least-privilege application policies.

interactive component to the access control.

When an app need permission to access the shared storage (part of the filesystem, common to all applications), an overlay is displayed, prompting the user for their decision on whether to allow or deny access to user's files. Notably, this approach avoids the problem of granting permissions up front, and always informs the user about the permissions that the app wishes to have.



(a) (b)

Figure 2.1: Permissions dialogues in Android 14: The location access permission dialogue (a) shows three options: "While using the app", "Only this time" and "Don't allow". The media access permission dialogue also has three, but different options: "Allow limited access", "Allow all" and "Don't allow"

Furthermore, starting in Android 11, whenever an app requests a permission related to location, microphone, or camera, the user-facing permissions dialogue contains an option called "Only this time". If the user selects this option in the dialogue, the app is granted a temporary one-time permission.[6]

Unfortunately, Android access control system is specific to Android. Also, it inherits the already mentioned drawbacks of containerisation, and only works through special API, thus requiring the software to be redesigned to work with such an access control system.

2.5 Ranacco

Finally, in McIntosh et al. 2021 [7] authors propose and implement software called *Ranacco*, which attempts to analyse various system environmental factors (e.g. latest mouse and keyboard activity) and file system operations to detect potentially malicious actions made by processes, in which case it delegates access control decision to the user. This approach avoids the shortcomings of other possible solutions, while remaining easy-to-use. Although this system is more advanced than the one we propose in this thesis, not only is it exclusive to Windows, but it also remains unavailable for the general public.

2.6 Requirements for the solution

Draft note: Negate the statements? (state what we want, not what we don't want)

The key issues with existent solutions, that our the system proposed in this thesis will try to address are as follows:

- Not all solutions assume processes to be malicious until proven (confirmed by the user to be) safe. Quite often access control permissions are either predefined, inferred or assumed.
- Some solutions can only enforce access policies on software that is distributed in a special way. This leaves the file system just as unprotected against all other software.
- Most solutions require passive action from the user besides initial installation (e.g. you have to reconfigure policies all the time). This adds further inconvenience to using such systems.
- Most solutions grant permissions forever, which significantly increases attack surface. Specifically, this opens up possibilities for attacks by proxy.
- Majority of solutions focus on preventing unwanted access by other users, which makes it unsuitable for single-user environments.
- Solutions are either overly complex and not user-friendly, or too simplistic to provide adequate granularity of permissions. This either leads to slower adoption of such systems, or makes them insufficient at protecting user data.

Chapter 3

Interactively Controlled File System

This chapter presents the solution developed for this thesis, the Interactively Controlled File System (ICFS), a user-centric filesystem layer designed to enhance access control through real-time user input.

ICFS provides users with direct control over filesystem access decisions. Unlike traditional systems relying on static policies, ICFS dynamically prompts users for authorization via a graphical interface, ensuring decisions align with immediate contextual needs.

Key Features:

- User-Friendly Design: Requires no prior configuration or specialized knowledge. The intuitive interface eliminates complex terminology, enabling seamless interaction.
- Dynamic Policy Enforcement: Permissions are established on-demand and stored for future reference, minimizing repetitive prompts.
- Granular Control: Policies apply at the process-file level, with options to generalize rules for broader categories, reducing user fatigue.
- Backward Compatibility: Implemented via the FUSE framework, ICFS intercepts system calls without altering existing software workflows.

3.1 Usage

To deploy ICFS, the user selects a target directory and executes:

```
icfs path/to/directory
```

This mounts ICFS over the specified directory, enforcing access control for all subsequent interactions. While the name includes "File System," ICFS operates as a

filesystem layer , intermediating between the physical filesystem (e.g., ext4) and user processes. It preserves the appearance of the original filesystem while enforcing its own access logic (implementation details in section 4.1).

3.2 Access Control Model

ICFS adopts a straightforward access control model:

- Subjects: Processes requesting access.
- Objects: Files or directories undergoing access attempts.

When a process requests access (e.g., open, modify, delete) to a file without pre-existing permissions, ICFS generates an access dialogue (see Figure 3.1).

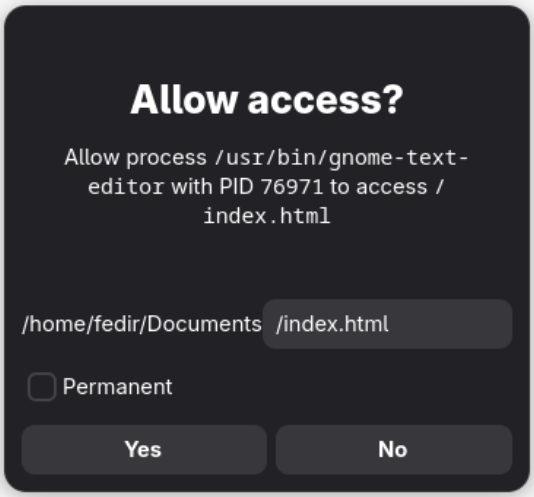


Figure 3.1: ICFS Access Dialogue: Displays the process executable name, PID, and target file path. Users may adjust the file scope, toggle permanent permissions, or grant/deny access.

The dialogue contains three functional elements:

- Access Grant Buttons.
  - Yes Button : Grants temporary access to the requested file to the requesting process only. If the user selects this option, the process is allowed to proceed with the requested access (e.g., read, write).

  - No Button : Denies access to the file for the current process. The filesystem returns an error (e.g., EACCES) to the requesting process, mimicking standard permission denial behavior.

- Permanent Permission Checkbox: A toggle labeled "Permanent" allows the user to persist the access decision beyond the current process. If checked , the permission rule (allow/deny) is stored in a local configuration database. The rule then applies to all future access attempts by processes (and any of their child processes) with an executable filename matching the requesting process. If unchecked , the decision applies only to the requesting process and it's child processes. That is, the process can actually access the file multiple times with this permission. Permissions granted with this box toggled on (off) will from now on be referred to as ”permanent“ (”temporary“).

- File Path Substitution Field: A text input field pre-filled with the absolute path of the requested file. Users may edit this field to modify scope of the permission (e.g., granting access to all files in the parent directory instead of a single file).

Behaviour changes slightly, if the operation is performed on a directory or a symbolic link: If the file is a directory, only changing the access mode and removal require permission from the user. With symbolic links, following is always permitted. If a process attempts to create a file, it is automatically granted permanent access to the file it has created.

This model addresses five key limitations of traditional systems:

- Reactive Configuration: No upfront setup required; permissions emerge organically.

- Temporary Permissions: Users may limit access to a single instance.

- Scalable Granularity: Policies adapt from specific files to broader categories.

The remaining two criteria are analysed in the next section.

3.3 Least Privilege vs. Usability

Balancing the principle of least privilege with usability posed the greatest design challenge. Strict enforcement—prompting for every access attempt—would minimize risk but overwhelm users.

To reduce friction, ICFS needs to keep the number of dialogues to minimum. This necessitates avoiding prompts for actions likely to be safe. However, we still aim to avoid granting excessive privileges by default.

When ICFS is initially started, no user decisions are known, and thus no processes have access to protected files. Each new access attempt triggers a privilege escalation request via the access dialogue.

Applying this rule strictly to all filesystem objects – including directories and symbolic links – with intelligent user decisions, would perfectly adhere to the principle of least privilege.

However, such strictness would render ICFS excessively cumbersome to use. To mitigate this, the rule has been relaxed to compromise user data as little as possible.

Firstly, Unlike POSIX, ICFS does not restrict directory visibility. While this exposes file structures, directory names rarely contain sensitive data.

Second, processes are permitted to create files without restriction. This decision is based on the observation that many programs create auxiliary and temporary files. For instance, the pdfLaTeX compiler creates 21 files in the source directory for this thesis, only 10 of which are human-editable; the remaining files are intermediary output of the compiler. Requiring the user to grant permissions for all these files would more than double the decision-making burden.

While this approach increases the potential for malicious processes to disrupt other processes, the risk is considered lower than the burden of constantly prompted permission requests. We discuss these limitations in section 5.3 of chapter 5.

Thirdly, all access permissions apply to the child processes too. Since only the parent process has control over starting its children, it is theoretically safe to presume that non-malicious processes won't spawn malicious child processes. Of course, this presumption is not necessarily true in reality: programs contain a plethora of bugs some of which might as well allow for arbitrary code execution, and thus starting of unwanted programs as its children.

However, we decided that the burden caused by having to allow access to all its children is way too high. For example, the Neovim text editor may spawn up to five additional child processes that analyse the opened file, such as code linters, formatters and debuggers. We discuss potential threats that relate to this rule in section 5.3 of chapter 5.

Chapter 4

Implementation

This chapter outlines the software design and architecture of ICFS, detailing how these elements address the challenge of fine-grained access control. Subsequent sections introduce the FUSE framework, methods for managing process-specific permissions, and the architectural strategies employed to mitigate unauthorised filesystem access.

4.1 FUSE framework

To regulate filesystem operations, ICFS employs the FUSE (Filesystem in Userspace) framework[8], which intercepts filesystem calls. FUSE enables the creation of custom filesystems or layers in user space, offering flexibility and ease of implementation. It provides an API for developers to define filesystem behavior. Once implemented (hereafter termed the FUSE application), the system mounts the custom filesystem at a specified location, substituting standard filesystem operations with methods defined by the API.

ICFS implements this API in C using the libfuse3 library [9]. It initializes the FUSE daemon via the `fuse_main()` function, which manages communication between the kernel and the FUSE application. Rather than directly overriding system calls, FUSE interacts with the kernel through `/dev/fuse`, a specialized device file that translates filesystem requests into API method invocations using a dedicated protocol.

ICFS does not have a backing store (a separate filesystem that contains actual data). Instead, it functions as a so-called passthrough filesystem, where system calls are forwarded to the original filesystem, if access control policies allow them.

To enforce access restrictions, ICFS mounts directly over the target directory, intercepting all access requests directed to it. As part of Linux's Virtual Filesystem (VFS) architecture, processes interacting with the protected directory are routed through ICFS. However, ICFS retains direct access to the underlying files by opening the directory with the `O_PATH` flag before mount. Subsequent operations are executed us-

ing "at"-suffixed system calls like `openat()`, performed directly at the file descriptor level[10], which bypasses ICFS's own layer.

4.2 Permission tables

To enforce an access control policy over time, filesystem needs to store user decisions in an appropriate data structure. As described in section 3.2, ICFS can give out two types of permissions: temporary and permanent. To accommodate this access control model, ICFS implements two data structures: a temporary permissions table, and a permanent permissions table, which we describe in detail in subsection 4.2.1 and subsection 4.2.2 respectively.

To pass permissions to child processes, both tables use procfs. When a permission check for the requesting process yields no results, recursive checks are performed on parent processes by traversing the process tree.

4.2.1 Temporary permissions

To function, temporary permissions storage should contain all information needed to identify the process, and associate the files to which the access is denied or allowed with it. We chose to keep track of processes by comparing the following characteristics:

- Process ID: Number that uniquely identifies a process on Linux systems.
- Start time: The time the process started after system boot. The value is expressed in clock ticks.

The process is considered the same if and only if both characteristics match.

At first, it might seem that factoring in start time is excessive. However, only using PID as the only identifying property of a process is problematic: PID is only unique among the currently running processes, not across the entire uptime of the system. Processes can not only acquire the PID of another, already finished process, but also attempt to request a specific PID. The start time is looked up in procfs by PID, which is provided by libfuse.

The temporary permissions table consists of tuples $(pid, starttime, allowed, denied)$, where *allowed* and *denied* are sequences of files, that the process is allowed or denied to access respectively.

In our implementation, entries are organised in a hash map, with PIDs as keys. This provides quick lookup of entries much needed for filesystem operations. ICFS uses the hash map implementation from the Convenient Containers library [11], that is well-tested and has an intuitive interface, which has helped to simplify the development.

One disadvantage of such a data structure, is that there isn't any inherent mechanism to remove entries that are no longer valid (e.g. permissions of a process that is already finished).

Unfortunately, we haven't found an efficient way to remove expired entries in the temporary permission table. On Linux, a process can't be notified of other processes' end unless they are child processes or the tracking process is being run with superuser permissions [12]. Hence, we had to resort to cleaning out expired entries using the garbage collection technique: an independent thread periodically checks validity of every entry in the table. If an entry is invalid, it is erased. We discuss its effect on performance in the chapter 5.

4.2.2 Permanent permissions

Since permanent permissions are granted to all processes' with the same executable, only it's filename is needed for identification. Since the permissions have to persist after filesystem restart, the table needs to be stored on the disk. Hence, we chose SQLite [13] as the backend for the permanent permissions table. It is well-tested and lightweight, making it an ideal choice for a program like ICFS.

Due to specifics of relational databases, the permissions are stored as a relation $(executable, filename, type)$, where $executable$ is the filename of the executable, $filename$ is a filename of the file that the permission targets and $type$ is a boolean value indicating whether the permission allows or prohibits access to the target file.

The database is stored in a file on the disk that the user chooses during startup. The database file is protected from outside access using standard POSIX permissions: during installation, a special user is created for ICFS, the owner UID of the executable is set to the UID of the new user, and the setuid bit is set, to allow other users to launch ICFS as a special user. On startup, database file is created as the special user, and the access mode is set to prohibit access by any other user. After the database is opened, UID of ICFS process (effective UID) is switched to the UID of the user (real UID) that originally started it using the `setuid` system call. The database remains open for the rest of the runtime of ICFS.

Unfortunately, in the current version of ICFS there is no way to edit the permanent permission table. We address this limitation in more detail in chapter 5.

Chapter 5

Evaluation

Draft note: This chapter is *very* incomplete, and only contains a brief and informal talk about the issues I am facing right now. This is not the actual thesis-worthy text. All issues discussed were relevant as of 11.04.2025

In this chapter presents the method of evaluating the solution is presented, and the found qualities of the solution are discussed.

Specifically will include:

- „Does the solution actually solve the problem?"
- Interoperability with other software: does using this fs break other programs, like whether it interferes with programs using auxiliary files, usability of terminal programs (`grep` is a particularly nasty one for this specific project).
- Performance and overhead.
- Security considerations.

5.1 Usability

While it is difficult to put an objective score on the usability of any system,

"does using this fs break other programs?"

~~Mostly - no. The biggest issue right now are (ironically) file trackers like tracker-miners. Those are programs that scan the filesystem (e.g. to make file search more efficient). The problem is that the current version of the software does not allow changing the scope of permissions you are giving (e.g. you can't just give permission to access the entire filesystem). The solution would be to give an ability to adjust the scope inside the permission dialogue. Everything on the "backend" side is ready for this change, but... since I am using zenity it does not easily give me the ability to just add another~~

~~element to the dialog. Probably a custom dialogue program has to be written, or multiple dialogues would have to be shown(e.g. standard one with Allow/Allow this time/Deny with a "more options" button, and then a second zenity dialogue with more detailed configurations) to solve the issue.~~

Now it does not.

"..like whether it interferes with programs using auxiliary files.."

No, this issue is solved. Because the programs that use such files are typically the ones that create them, they automatically get the permissions to access them.

In fact, I am writing this thesis inside of a folder managed by ICFS, and even despite TeX's notorious love for auxiliary files it works just fine.

"..usability of terminal programs (grep is a particularly nasty one for this specific project)"

~~Yes, this still is an issue. The problem of grep is the same as with file trackers, so I will skip it.~~

This is mostly solved in my experience. It wasn't as annoying to use the terminal programs as I initially expected.

As for terminal programs, I see these possible ~~solutions~~ improvements:

- Use SID and TTY to identify a shell session (like `sudo` does).

5.2 Performance

Performance of ICFS is terrible. ~~Unfortunately, I was unable to make perf work with it for some reason, so I don't really know what is slowing operations down. So those are my speculations for what *may* be the bottleneck. A lot of it is caused by it's design. For example, ICFS needs to look through procfs to get process creation time, and there is no way of going around this it seems.~~

I managed to get the `perf` to work: it showed that almost all time was consumed by libfuse, not my program. My code used something like 0.0001% of runtime on a pretty heavy test. I am not quite sure what to do, since libfuse is already optimised to smithereens. I guess I will just write how it is and not touch the performance ever again.

5.3 Limitations

It can only be used in a single-user environment.

Conclusion

Draft note: This chapter is *very* incomplete.

In conclusion, the overall value and benefits of the solution is discussed(reiterated :)).

Bibliography

[1] N. Y. Lovyagin, G. A. Chernishev, K. K. Smirnov, and R. Y. Dayneko, "Fgacfs: A fine-grained access control for *nix userspace file system," *Computers & Security*, vol. 88, p. 101 632, 2020, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2019.101632. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404819301798.

[2] "Flatpak - the future of application distribution on linux.," Flatpak Team. (2025), [Online]. Available: https://flatpak.org/.

[3] "Snapcraft - snaps are universal linux packages.," Canonical Ltd. (2025), [Online]. Available: https://snapcraft.io/.

[4] S. Peter. "Appimage | linux apps that run anywhere." (2019), [Online]. Available: https://appimage.org/.

[5] T. Dunlap, W. Enck, and B. Reaves, "A study of application sandbox policies in linux," in *Proceedings of the 27th ACM on Symposium on Access Control Models and Technologies*, ser. SACMAT '22, New York, NY, USA: Association for Computing Machinery, 2022, pp. 19–30, ISBN: 9781450393577. DOI: 10.1145/3532105.3535016. [Online]. Available: https://doi.org/10.1145/3532105.3535016.

[6] "Permissions updates in android 11." (2025), [Online]. Available: https://developer.android.com/about/versions/11/privacy/permissions.

[7] T. McIntosh, A. Kayes, Y.-P. P. Chen, A. Ng, and P. Watters, "Dynamic user-centric access control for detection of ransomware attacks," *Computers & Security*, vol. 111, p. 102 461, 2021, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2021.102461. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404821002856.

[8] "Fuse — the linux kernel documentation." (), [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[9] "libfuse." version 3.17.2. (2025), [Online]. Available: https://github.com/libfuse/libfuse.

[10] *Open(2) system calls manual*, 6.9.1, Free Software Foundation, May 2024.

[11] J. L. Allan. "Github - jacksonallan/cc: A small, usability-oriented generic container library." (2025), [Online]. Available: `https://github.com/JacksonAllan/CC`.

[12] "C++ - how to get notified when a process ends under linux? - stack overflow." (2016), [Online]. Available: `https://stackoverflow.com/questions/34800568/how-to-get-notified-when-a-process-ends-under-linux`.

[13] R. D. Hipp. "SQLite." version 3.47.2. (2024), [Online]. Available: `https://www.sqlite.org/index.html`.