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

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 [7] 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[5] or AppImage[1]. 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 interactive component to the access control.

[1]It is important to mention, that although this flaw remains unmitigated, the analysis made by Dunlap et al. 2022 [6] shows that most package maintainers actively attempt to define least-privilege application policies.

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.

Picture of an Android permissions dialogue.

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.[4]

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 [8] 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

In this section we present the solution developed as a part of this thesis, named *Interactively Controlled File System* (or ICFS for short).

ICFS is a filesystem layer that gives user direct command over its access control. Instead of relying on static policies or rules, it prompts the user for the access control decision via graphical interface.

ICFS is user-friendly and trivially easy to use. It does not introduce any new terminology or complex access control management strategies. The graphical interface is intuitive and self-explanatory. ICFS is configured on the fly: as programs request access, the user's decisions are recorded and later reused. There is no need for any configuration besides installation and choosing a directory to control. It operates on the level of individual processes and files, ensuring high granularity.

It is backwards compatible: ICFS overrides the regular system call interface using FUSE framework, which means that any software that wishes to use the files ICFS protects has to respect it's policies. Its interactivity combined with the ability to only grant permissions for the lifetime of a specific process makes proxy attacks very difficult to go unnoticed.

3.1 Usage

To use ICFS, the user only needs to select a directory that they wish to protect, and run:

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

Upon running this command, ICFS will be mounted over the chosen directory, and access to every file in it will from now on be controlled by ICFS.

Although it has the words "File System" in the name, ICFS is not an alternative to real filesystems like ext4 or btrfs. Instead, it is a *filesystem layer*, that sits in between

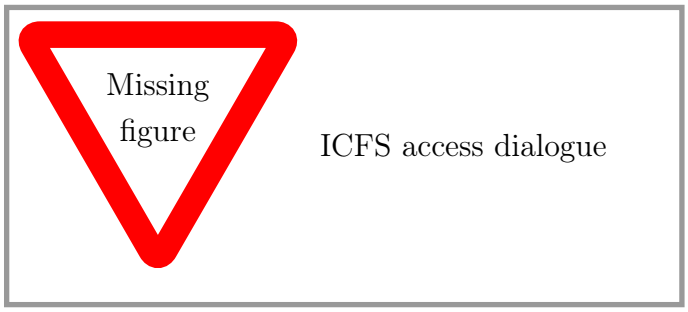
the actual on-disk filesystem and the virtual file system presented to the processes. It acts as an intermediary that intercepts the filesystem calls and enforces it's own logic. The exact details of the implementation of such mechanism will be discussed in the section 4.1.

What is important for this section however, is that from the standpoint of the user processes it looks just like the underlying filesystem would. Processes would just see the same folder that once was in ICFS's place.

3.2 Access Control Model

As promised, the access control model of ICFS is trivially simple. It features processes as it's subjects, and files as objects. Whenever a process attempts to open, remove or change the access mode of an existing file and no previous decisions were made regarding process's access to the file, window with a dialogue is displayed with three options:

- *Allow*, that will allow this process and any other process that is started with the same executable to access the file.
- *Allow this time*, that will allow the requesting process to access the file.
- *Deny*, that will deny all access to the file.



We will later refer to these windows as *access dialogues*.

Decisions made in the dialogues are recorded and later reused when processes access files again.

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 obviously mitigates at least four of the six previously laid out problems of existent solutions:

- Usage of such an access control is entirely reactive. The user does not need to configure anything until the filesystem access is actually needed.
- User can choose to only grant permissions temporarily, using the "Allow this time" option.
- The solution is designed specifically to work with single-user environments.
- Provides a very fine granularity of permissions. ICFS can deny or allow access of a process to a specific file, which is way finer than

The two remaining criteria will be discussed in more detail in the next section.

3.3 Least privilege and user-friendliness

Following the principle of the least privilege and being user-friendly at the same time, is the single biggest challenge of this work. Unfortunately, those features are mutually exclusive most of the time.

In order to stay user-friendly, ICFS needs to keep the number of dialogues to minimum. This means we have to avoid prompting for actions that are likely safe. However, we still want to follow the principle of the least privilege, and avoid allowing any kind of privileges to all software by default.

When ICFS is started for the first time, no user decisions are known yet. At this point, no processes have access to any file controlled by ICFS. When a new attempt to access a filesystem object is made, a kind of privilege escalation is attempted with the access dialogue. Assuming the user's decisions are intelligent, the principle of the least privilege would be followed perfectly in this case.

However, if the above rules were followed literally, the ICFS would be *very* tiring to use. To mitigate this issue, the rule had to be somewhat relaxed in way that compromises the least amount of user data.

Firstly, unlike POSIX access control policies, ICFS does not give an ability to hide the directory structure from processes. The only thing that a malicious process can gain from this relaxation of the original restrictions is the ability to see the directory structure, which is unlikely to contain any kind of sensitive data.

One more relaxation we decided to make was allowing processes to create files without restriction. This decision comes from an observation that most programs often open auxiliary and temporary files to functions. For example, one program that is particularly notorious for such behaviour is the pdfLaTeX compiler. Out of 21 files in the source directory for this thesis, only 10 files are actually human-editable and are not intermediary output of the compiler. The amount of decision-making required for

the user to give all appropriate permissions in this case is more than double than the amount of files where the actual data is stored.

Because of the overwhelming amount of additional files that programs need to create to function, we decided that the risk of letting the program create arbitrary files is less important. Indeed, if the process creates a new file, than it can't possibly extract any additional data from the system.

However, this decision actually opens up some possibilities for the malicious processes to disrupt functioning of the other processes. We discuss those in the section 5.3 of the chapter 5.

Chapter 4

Implementation

This chapter describes the software design and architecture, and the way that they help to solve the problem. The following sections describe the FUSE framework, different methods used to store access permissions of processes and the way the chosen architecture is designed to resist unauthorised access to the filesystem.

4.1 FUSE framework

In order to control the filesystem operations performed by the processes, ICFS uses FUSE (Filesystem in Userspace) framework[3] to override the filesystem call interface. FUSE allows to implement custom filesystems or layers in userspace, which makes it very flexible and easy to use. FUSE defines an API that can be implemented by the developers of the filesystems. After the implementation (which will be referred to as *FUSE application*) is launched, it mounts its filesystem at the specified location, and replaces the filesystem calls with its own methods, according to the FUSE API.

ICFS implements the API in C, using libfuse3 library. It then launches the FUSE daemon through fuse_main() function, which sets up the filesystem, and performs all the communication between the kernel and the FUSE application. FUSE does not directly replace syscalls, but instead communicates with the kernel via a special device in /dev directory, called /dev/fuse. This device uses a special protocol to communicate with the kernel and translate filesystem calls into FUSE API method invocations.

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

4.1.1 Hiding the underlying directory

Draft note: Write how the source directory is protected

4.2 Permission tables

Draft note: Write how permission tables scheme was chosen

4.2.1 Temporary permissions

Draft note: Write how temporary permission tables work. Specifically, how temporary permissions work with the cc library and start time process identification

4.2.2 Permanent permissions

Draft note: Write how permanent permission tables work. Specifically, how permanent permissions work with sqlite3.

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 Interoperability

"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.

"..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.

As for regular terminal programs, I see these possible solutions:

- 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.

But a lot can also be improved. For example,

- sqlite queries can be pre-"compiled"
- (I think) more paralellism can be achieved.

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] Appimage | linux apps that run anywhere.

[2] Flatpak - the future of application distribution on linux.

[3] Fuse — the linux kernel documentation.

[4] Permissions updates in android 11.

[5] Snapcraft - snaps are universal linux packages.

[6] Trevor Dunlap, William Enck, and Bradley Reaves. A study of application sandbox policies in linux. In *Proceedings of the 27th ACM on Symposium on Access Control Models and Technologies*, SACMAT '22, page 19–30, New York, NY, USA, 2022. Association for Computing Machinery.

[7] Nikita Yu. Lovyagin, George A. Chernishev, Kirill K. Smirnov, and Roman Yu. Dayneko. Fgacfs: A fine-grained access control for *nix userspace file system. *Computers & Security*, 88:101632, 2020.

[8] Timothy McIntosh, A.S.M. Kayes, Yi-Ping Phoebe Chen, Alex Ng, and Paul Watters. Dynamic user-centric access control for detection of ransomware attacks. *Computers & Security*, 111:102461, 2021.