University of Padua

Department of Mathematics "Tullio Levi-Civita"

Master's Degree in Computer Science

Towards Secure Virtual Apps: Bringing Android Permission Model to Application Virtualization

Master's Thesis

Supervisor

Prof. Eleonora Losiouk
University of Padua

Candidate

Alberto Lazari
2089120

Academic Year 2023–2024

Acknowledgements

Many have supported me along the path to this achievement. I would like to share my deepest gratitude to each one of them, although I know that words alone will never be enough to express it completely. I hope someday I can return this kindness. In the meantime, let me try my best through these black letters on a white page.

To Professor Losiouk, for inspiring my interest in a new field and supervising the long process that led to this work with admirable patience and care.

To everyone at Corollario, for their warm welcome into a new adventure. Although it has been a short experience up until now, it filled the last days of my degree with inspiring harmony.

To everyone within the scouting world I had the privilege of meeting. So many embodied living examples of the kind of Christian, citizen, and person I aspire to become as a man.

To my closest friends, that I can always trust to be there for supporting and encouraging me. I am lucky to have their constant belief in me and my capabilities, even more than I have in myself.

To my family, that, no matter what, is always fully supporting my choices with their love and care. To my parents, for trusting my decisions and always guiding me toward my achievements. To my sister Anna, that always manages to bring a smile to my face and makes me proud of her achievements. To my brother Francesco, that always has been a great inspiration in so many aspects of my life.

To each one who brightened my days in Pauda. They are too many to list them all, but each should know they share a special place in my heart. Without them, I would not have come this far—and this is not a formality, I truly mean it. The excitement of small fun projects, the unnecessary knowledge shared during lunches, the support in overcoming exams, the bonds that last far beyond lectures and the narrow borders of Padua and Italy. These are just a handful of reasons that made me appreciate my time at university and this field of study, and that ultimately brought me here, to the end of this incredible journey.

Abstract

Android app-level virtualization allows apps to run in isolated environments within a host app. However, existing solutions lack dedicated virtual permission management, posing significant security concerns. When a host app requests a permission, it is shared across all virtual apps in it, bypassing individual checks and granting unintended access to sensitive resources, especially when multiple virtual apps share the host.

This work presents a custom permission management model, integrated into an existing virtualization framework. It redirects permission operations to enforce fine-grained controls for virtual apps, treating them as distinct entities. Evaluation proves its effectiveness in enforcing permissions for basic use cases and specific features.

The thesis also addresses challenges in scaling the solution for more complex interactions, highlighting limitations of Android's app-level virtualization architecture. Potential approaches are explored, emphasizing the need for further refinement.

Index

1

Introduction

As one of the most widely used mobile operating systems, Android is built around robust security principles. Fine-grained permission management is one of the fundamental ones, controlling access to sensitive resources like contacts, camera, and location data, it ensures that apps can only access what they explicitly request and receive user approval for. This permission model has evolved significantly over the years to address the growing complexity of mobile ecosystems and to meet the increasing demand for secure app interactions. However, this principle of precise, user-mediated permission enforcement breaks down in the context of app-level virtualization frameworks.

Android *app-level virtualization* is a technique that enables multiple apps to run within a single host app, each in its isolated virtual environment. While this approach can be used to enhance security through isolation, it fundamentally alters the way permissions are handled. In most app-level virtual environments, virtual apps do not independently request permissions. Instead, they inherit the permissions granted to the host app. This creates a critical vulnerability: a virtual app can access sensitive resources it does not need or should not have, simply because the host app has been granted those permissions.

This behavior undermines Android's permission model, particularly in scenarios where multiple virtual apps share the same host. The lack of fine-grained permission management in such environments not only compromises user privacy, but also exposes the limitations of app-level virtualization in following Android's security principles. It is also a critical security threat, enabling virtual apps to perform privilege escalation attacks.

The main goal of this thesis is to address this issue, by proposing and implementing a custom virtual permission model for Android app-level virtualization. This model aims to replicate and extend the native Android permission system, to ensure that virtual apps are subject to proper permission checks and that permissions are managed for individual apps. To realize this, the work focuses on extending an existing app-level virtualization framework, by integrating the custom model in a virtual permission management system that intercepts permission-related operations from virtual apps, ensuring fine-grained access control.

Before designing this custom permission model, Android's native permission model implementation is analyzed by exploring parts of the Android Open Source Project (AOSP) code and the

behavior of apps in recent Android versions. This step ensures that the virtual system closely resembles the system's original permission model, enabling smooth interaction and compatibility between the two. By understanding how Android's permission system operates, the virtual framework can easily replace the it in the virtual environment, safely managing permissions for virtual apps in a consistent way.

This work also details the virtual permission model's implementation and evaluates it through both controlled tests and a real-world application to prove its ability to enforce appropriate permission management in the virtual framework. Additionally, the challenges and limitations of scaling this model to handle more complex system-level interactions are discussed, particularly their relations to the constraints of app-level virtualization and Android's architecture.

Finally, potential future directions are explored, including research into OS-level virtualization as a promising solution to address the limitations of app-level approaches. Additionally, alternative solutions to overcome the limitations of the current virtual permission model are discussed, aiming to enhance its scalability and effectiveness in more complex scenarios.

2

Background

2.1 Virtualization

2.1.1 Overview

Virtualization is a technique used to create abstractions of real environments or resources, providing replicas that are more flexible, isolated, and scalable than their concrete counterparts. By simulating the functionality of hardware, software, or other resources, it allows multiple isolated environments to be created and run within a single system, often simultaneously. This approach can be used to enhance security or maximize resource efficiency of a system, as each virtual environment can be controlled and managed independently.

One of the earliest implementations were multiple-user operating systems for 1960s mainframes, where each user was given a virtual environment running on the same physical device, sharing its resources. This model was not only meant to let multiple operators access an individual mainframe concurrently, but to also give them a private, independent space, that would simulate having a computer of their own.

The concept evolved with time and started being used for the creation of *virtual machines* (VM), that allow for the virtualization of the entire software stack, leading to actual "computers within a computer". VMs are managed by a hypervisor, a software layer that coordinates and isolates them, making it possible to run multiple operating systems on the same hardware. This technology is frequently used to provide pre-configured machine images for reproducibility, ensure consistent environments, or create secure spaces that are kept isolated from the host system.

More recently, virtualization has extended its use-cases to OS-level environments, which come in the form of containers, created with tools such as Docker [1] and Podman [2]. They are OS instances that share the host kernel and are widely used for running multiple virtual servers on a same physical one, often by creating them from scratch on the fly from a previously specified configuration. Containers can be used for creating lightweight reproducible environments to run software tests, compilation tasks, or deploy modular and scalable software with pre-configured

dependencies. They are usually preferred to full VMs, because they require a smaller overhead on system resources and simpler configuration.

2.1.2 Android Virtualization

As virtualization technology has evolved, efforts to adapt its principles to Android have emerged. Recent works, like VPBox [3], are exploring OS-level virtualization for Android, by aiming to run isolated instances within containerized environments. However, these methods typically require custom Android images or modifications to the system, making them impractical for a general, end-user adoption.

Because of these limitations, app-level virtualization has become the most practical solution in the Android context. Unlike OS-level virtualization, which demands kernel-level access and extensive configuration, app-level virtualization handles the creation and management to a single application that operates entirely within the user-space layer, the *container app*, providing a convenient method for offering isolation without requiring system alterations. The creation and management of the virtual environments is handled entirely within a single application, the *container app*, which can host one or more *plugin apps*, the virtualized apps. This design enables features such as:

- App cloning: running multiple instances of the same app simultaneously within isolated contexts.
- Sandboxed environments: enabling apps to run in a more restricted and controlled environment for enhancing security and research.
- Dynamic patches: applying hotfixes or updates to a virtualized environment without modifying the main system [4].

2.2 Android Architecture

The Android architecture consists of many layered components that interact to provide essential features. Each layer has specific responsibilities, from user applications down to the low-level system components, contributing to Android's performance, scalability, and security.

Virtualization frameworks often need to interact with or emulate virtual versions of certain components. The following sections outline the purpose of each layer, providing an essential context understanding Android virtualization.

Application Layer. The Application layer consists of user-facing apps, that can be installed and managed by the user. Each app runs in its own sandboxed process, ensuring security and privacy by isolating apps from one another. The sandbox is guaranteed by three concepts:

1. UID model: a unique user ID (UID) is assigned to each app by the system, by creating a dedicated Unix user. This ensures that each application has its own private storage directories and files, which are kept isolated from other apps.

2. Process separation: each app runs as an independent OS process, which means that the underlying Linux process isolation applies by default. This ensures that memory and resources allocated to one process are not accessible by others unless explicitly shared.
3. Permission model: Android enforces a fine-grained permission model that controls access to specific system resources and data. Permissions are granted based on the app UID and GID.

Java API Framework. This layer provides a set of APIs that enables third-party applications to manage UI elements, handle application lifecycles, and interact with system services. It is implemented as an extensive codebase of Java and Kotlin classes, presented in the official documentation as the same framework used by system apps [5], thus providing the entire feature-set of the Android OS. Starting with Android 9, however, the framework introduced a separation between SDK and non-SDK interfaces via the hidden APIs list [6]. This created a gap between the capabilities of system applications and those available to third-party applications, restricting access to certain internal features.

System Layer. The System layer includes essential system applications and services that manage core functionalities of the OS, such as telephony, location and media. These components are granted elevated permissions and provide services that user apps rely on, but cannot directly access.

- Services: examples of key system services include the Location Manager, Telephony Manager, Notification Manager. They expose their functionalities using interfaces in the framework API, often granting controlled access to sensitive resources that are restricted behind specific permissions. The actual implementation resides in system components that live in their own elevated process.

  While service interfaces are typically defined in `android.*` packages, their implementations are placed under `com.android.server.*` packages. This separation is even more evident in the file system structure of the source files, where services have their source code organized in a dedicated `services/` directory tree, which puts emphasis on the fact that they are components of the system, and not part of the Android framework API directly.
- System apps: they are regular apps that come pre-installed with the system image and can be granted system permissions [7]. They are installed under a read-only directory, to avoid deletion and modification.

Binder Mechanism. The Binder mechanism is Android's core inter-process communication (IPC) system, used allow components running in different processes to communicate with each other. Acting as a bridge between the application layer and system services, it provides a way for apps to request and access services and resources managed by the system. It is used to ensure security in the system is maintained, by enforcing permissions and sandbox policies, or allowing them to be enforced by the services themselves.

While not an actual layer of the Android architecture, it is a crucial structural component. Its role is especially relevant in app-level virtualization and in the implementation of Android's permission model.

Native Libraries and HAL. Many components are implemented at a lower level using native C and C++ code and require native libraries providing basic system interaction, such as Libc, WebKit, and Media Framework. The hardware abstraction layer (HAL) is one of these components and defines standard interfaces for hardware components, allowing Android to interact with device hardware without requiring device-specific code at higher levels.

Android Runtime. Starting from Android 5, Android runtime (ART) is the execution environment for Android apps, each running its own instance of the runtime within its process. It compiles and executes the app's code in the Dalvik Executable format, a reduced bytecode format designed for minimal memory footprint on Android devices. It is able to leverage both Ahead-Of-Time (AOT) and Just-In-Time (JIT) compilation techniques, improving the balance between performance and memory usage. It also provides some runtime libraries to support most of the functionality of JVM-based languages, like the Java and Kotlin languages and Java 8 features.

Linux Kernel. At the foundation of the Android OS, a custom Linux kernel manages fundamental system tasks like memory management, process scheduling and control, and device I/O. It is configured with custom Security-Enhanced Linux (SELinux) policies, to enforce mandatory access control (MAC) over all processes [8].

2.3 VirtualXposed

2.3.1 Project Overview

The original Xposed framework is a powerful tool designed for rooted Android devices that allows users to modify and customize system behavior at a deep level. It was developed for older Android versions and works by injecting code directly into the Android runtime, allowing users to change how both system and user applications behave, without altering their APK files. The framework hooks into the Android runtime (ART, or Dalvik in older versions), intercepting specific method calls from apps and system services. This hooking process allows modifications (called modules) to replace or extend the behavior of the original functions. For example, a module could hook into the method responsible for determining app permissions, allowing users to override permission checks or alter granted permissions dynamically. Similarly, Xposed modules can also alter UI elements, bypass in-app restrictions, block ads, or change app functionality based on user preferences.

In the last years, Android's increasing security measures and restrictions on low-level access (more specifically SafetyNet) have made it more challenging to keep Xposed's full functionality on newer Android versions. Currently, Xposed has been discontinued and replaced by newer

implementations, such as LSPosed [9] and EdXposed [10]. Additionally, root access has been progressively limited and discouraged in the Android OS over the course of time.

VirtualXposed [11] is a project designed to bring this functionality to unrooted devices by creating a virtualized environment, recreating the Xposed experience within a sandboxed space. It uses VirtualApp [12] at its core, which is one of the most popular open-source app-level virtualization frameworks. It is used to create a virtual environment and run apps inside of it by injecting hooks at the application layer. VirtualXposed is thus limited to modifying the virtual app behavior rather than system-wide functions.

Over recent version updates, Android has introduced various security and architectural changes, which lead to many Xposed functionalities becoming partially broken or incompatible. Despite these limitations, VirtualXposed remains a valid project, providing a mature VirtualApp wrapper. This is particularly significant since VirtualApp has been closed-sourced by its author, meaning it is only actively maintained in its business version. However, within the scope of VirtualXposed, the framework has been maintained (to a certain degree) for compatibility with recent Android versions. This currently allows VirtualXposed to provide a functional, pre-configured application that comes with a fully integrated UI and launcher, removing the need for specific manual setup.

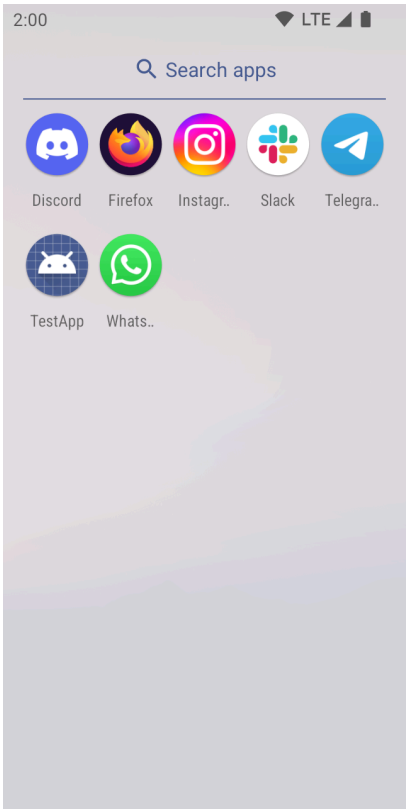

Figure 1: VirtualXposed integrated launcher.

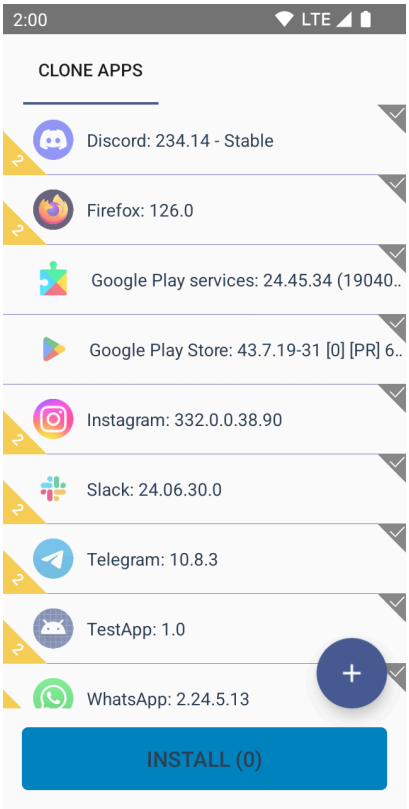

Figure 2: Application installer activity.

2.3.2 VirtualApp Architecture

By looking at the structure of VirtualXposed's project, the code is divided in three Gradle projects:

- `:app` is the UI part of the application, declaring activities and core components that interact with the virtualization framework.
- `:launcher` is a fork of a AOSP-like launcher, adapted to be included in the application.
- `:lib` is the actual VirtualApp framework, back-end of the application.

This section explores the high-level architecture of the `:lib` project, highlighting the purpose of components that are relevant for the following chapters.

Virtual Services. VirtualApp relies on services that emulate the behavior of Android original ones, providing the same functionalities to plugin apps, hence they are called *virtual services*. They are usually identified in the framework by the original service name, prefixed with "V". Acting as an intermediary layer, these services process requests from virtual apps, either handling them directly within the virtual environment or forwarding them to the OS when necessary.

One of the key services is the `VPackageManagerService`, which acts similarly to Android's `PackageManager` and maintains and manages the list of installed plugin apps and their resources. This service makes sure that when a virtual app requests package information, it gets responses adapted for the virtual environment, while still providing access to features from the real `PackageManager` service.

Custom App Process and Activity Handling. To manage virtualized applications, the framework makes use of multiple processes.

An engine process, that operates as a daemon service, is kept running in the background. It hosts server components of the framework, which include the core virtual services and managers that handle fundamental tasks for handling plugin apps.

Additionally, the framework defines a set of 50 fixed processes, named sequentially from `:p0` to `:p49` in its manifest file. Each of these is used to host stub activities, dialogs, and content providers, with their lifecycles managed by the `VActivityManagerService`. Virtual apps use these as a bridge between their virtual components and actual ones that the Android OS can see and manage.

By using this method, VirtualApp can ensure that each virtual app's components behave as if they were integrated directly into the OS, while still providing some fundamental level of isolation. The use of different processes, specifically, provides the inherent separation that comes with the OS' natural process isolation.

Dynamic Proxies. VirtualApp also uses Java's dynamic proxies to intercept system service calls made by plugin apps, which makes it possible to modify or simulate responses as needed. These proxies are created by defining a class that implements Java's `InvocationHandler` inter-

face, which allows it to wrap an object and intercept all method calls to it by implementing the `invoke` method. This approach lets VirtualApp intercept system service calls, forward them with modified parameters, or return custom responses to virtual apps.

Android apps typically communicate with system services through the Binder, with proxies automatically generated from Android Interface Definition Language (AIDL) interfaces. In the Android Framework, for example, the `ActivityManager` service provides its functionalities by redirecting method calls to the `IActivityManager` interface, which is actually implemented in the `ActivityManagerService`.

To ensure that plugin apps can work within the virtual environment, VirtualApp injects its dynamic proxies into plugin apps processes, replacing system implementations with the framework versions. This approach ensures that even apps that are only installed in the virtual environment can interact with Android's system services, with the flexibility to control their behavior or manage permissions through VirtualApp's own services.

Virtual Environment. Since plugin applications in VirtualApp share the same UID as the container app, from the point of view of the system, they face challenges that have to be addressed, in the context of a virtual environment. Typically, Android assigns a unique UID to each application, restricting access to its private data folder, ensuring isolation between apps. However, in VirtualApp's case, all plugin apps use the same UID as the container app, meaning they would have access to the same data folder. To address this issue, VirtualApp performs a redirection at a file system level and manages virtual UIDs.

For file system-related operations, it uses native hooks to intercept system calls to low-level functions, such as `open()` and `fcntl()`, and modifies file paths to redirect them to isolated storage locations, managed by `VEnvironment`. Each plugin app is assigned a unique private folder inside the container's data folder, ensuring they do not conflict or access each other's data.

The framework also includes a virtual UID system, handled in `UidSystem`. Upon installation, VirtualApp assigns a unique virtual UID to the app's package, much like the Android OS itself. As explained in later chapters though, this UID is shared between a same app's clones.

These mechanisms must be considered by virtual services, and this is typically handled inside virtual proxies implementations. For instance, when an app calls `getCallingUid()`, the proxy replaces the container's UID with the plugin app's virtual UID. Similarly, when forwarding requests to the system, the proxy ensures that the container's UID is properly used.

Mirror Classes. Supporting virtualization requires an extensive use of hidden APIs, which have been restricted to normal apps by Android's SDK policies, starting with Android 9. The limitations are bypassed by using a library that is able to disable the block and allow these APIs to be called via reflection. To avoid extensive use of reflection—which requires much boilerplate code—VirtualApp provides a set of mirror classes that replicate Android platform classes, exposing hidden APIs and fields accessible for the framework's functionality.

3

Related Work

Android virtualization has developed significantly over time. In the beginning, app-level virtualization tools were widely used because of their lightweight and flexible nature. However, these systems often struggled to properly replicate Android's sandboxing, which led to many privacy and security problems. Research at the time focused on addressing these issues, but the design of app-level solutions made achieving full security a challenging task.

To overcome these limitations, the field shifted towards OS-level virtualization, which provides stronger isolation and better control. Recent advancements, such as Google's introduction of pKVM and the Android Virtualization Framework in Android 13, highlight this trend.

The following sections provide an overview on key contributions and advancements in virtualization security, outlining both app-level and OS-level approaches and their impact on the ecosystem.

3.1 App-Level Virtualization Security Concerns

While providing a lightweight and flexible approach, app-level virtualization introduces multiple security challenges due to the lack of robust isolation between virtualized applications and the host system. These issues have been discussed in different research works [13, 14, 15], with the primary issues being outlined in the following subsections.

3.1.1 Same UID Across Apps

A fundamental issue with app-level virtualization is that plugin applications are executed as separate processes within the container application, inheriting the host's UID. This causes Android to treat them as the same app, leading to a lack of differentiation between the host and the virtualized apps.

Since the UID is a key element used by the system to enforce security policies and isolate apps, sharing it introduces several vulnerabilities. A significant risk is that plugin apps gain access to the internal storage of the entire container app, potentially allowing them to read sensitive data from other plugin apps, or even inject malicious code by tampering with the internal files of these apps. This last issue is not addressed in this thesis, as the VirtualApp framework already

includes mechanisms to mitigate these risks. However, their consistency and effectiveness should be verified.

3.1.2 Privilege Escalation

Another major security concern comes from the shared UID, which gives plugin apps access to all the permissions requested by the host application. This creates opportunities for several types of attacks.

For instance, a plugin app can gain access to resources not explicitly declared in its manifest, but for which the container app has access, since the system only enforces restrictions on the container app. Furthermore, if the container app is granted a dangerous permission for a plugin app, it is automatically extended to all other virtual apps, including potentially malicious ones. This unintended privilege escalation can lead to serious security breaches, allowing apps to perform actions beyond their intended scope, often without users noticing any anomalous behavior.

This issue is the main focus of this thesis.

3.2 Boxify

Introduced in 2015 as one of the first attempts at app-level virtualization for Android, Boxify [16] aimed to provide full-fledged app sandboxing on stock Android devices. Its core concept was based on app virtualization and process-based privilege separation, which isolates untrusted apps within a controlled environment without requiring modifications to the underlying Android firmware or root access.

The primary contribution of Boxify is its ability to enforce security policies at the application layer, creating a unique approach for sandboxed apps to communicate with the Android framework. It used Android's isolated processes, which are granted zero permissions by default. The container app needed to explicitly grant them to plugin apps, allowing for a complete control on which kind of resources apps were allowed to access, making it possible to implement a fine-grained permission control system.

Despite its promising concept, Boxify was never released for public use, and its implementation details are somewhat limited in the paper: replicating the system would be challenging due to the lack of practical explanations. However, the concept remains influential in discussions around Android app security and sandboxing techniques.

3.3 OS-Level Virtualization

OS-level virtualization frameworks provide a stronger alternative to app-level virtualization by offering enhanced isolation and security guarantees. Key projects in this domain include:

- VPDroid [17] and Cells [18]: these are older, pioneering virtualization solutions, focused on isolated execution environments to ensure app separation and control over resource access. They laid the ground for many of the concepts used in modern Android virtualization.

- VPBox [3]: a recent solution to address some core challenges observed in app-level virtualization, particularly around stealth and transparency. The framework is capable of customizing device attributes and simulating multiple virtual devices on a single hardware system, all while maintaining native performance.

3.3.1 Android Virtualization Framework (AVF)

Introduced on selected devices with the release of Android 13 [19], the Android Virtualization Framework provides official OS-level support for virtualization, marking a significant step towards secure, isolated execution environments.

This framework is designed for specific devices and allows small payloads to run in a virtualized secure environment. It is backed by a private Kernel-based Virtual Machine (pKVM), a hypervisor built on top of Linux's KVM, and relies on a minimalistic environment known as Microdroid, a stripped-down OS that lacks any graphical support. This makes it unsuitable for general-purpose use cases, since it is not intended to run full applications or entire operating systems. Instead, Microdroid is optimized for running small, isolated workloads that require secure environments, such as certain system tasks or trusted applications.

Overall, AVF is a more specialized tool, compared to broader frameworks that allow full app virtualization or the execution of entire guest operating systems.

4

Android Permission Model

The Android permission model is designed to protect user privacy and security, preventing apps from freely accessing sensitive information about the user or the device. It controls access to resources and features that extend beyond the app's sandbox, such as:

- Hardware device capabilities (e.g., internet connection, location, microphone).
- User's private data (e.g., calendar, media content, SMS).
- System and device settings.

This system of permissions gives apps a mechanism to follow the least privilege principle and increase user awareness, creating a barrier that requires—either explicit or implicit—user or system consent before sensitive information or powerful features can be accessed.

Every app must declare all required permissions in its `AndroidManifest.xml` file. The manifest can be seen as a contract, listing permissions the app requests and will probably use at some point. If an app tries using additional permissions, that were not declared in the manifest, it cannot request or use them at runtime. By requiring these declarations, Android ensures transparency, since users are presented and can review the requested permissions before installation.

4.1 Protection Levels

Permissions are assigned a *protection level* at the time they are defined. It reflects how sensitive the property protected by a permission is, and determines which steps an application needs to perform to obtain it.

Protection levels categorize permissions as follows:

1. Normal permissions: they grant access to simple resources that pose a low risk to user privacy and other apps' security, such as setting alarms, checking the network state, or controlling the flashlight status.
2. Dangerous permissions: they provide access to sensitive data or resources, such as device's location, camera, or contacts, which have a higher impact on user privacy.

3. Signature permissions: they are only granted to apps that are signed with the same certificate as the app defining the permission. These are typically used when two or more apps from the same developer need to share data or functionality. They can also be system permissions that are signed with the OS signature, meaning that only system and privileged apps can use them.

Prior to Android 6 (Marshmallow), both normal and dangerous permissions were automatically granted at install-time, where users had to either accept all requested permissions or cancel the installation. Since Android 6, apps are required to request individual dangerous permissions before using them, prompting users to approve each permission request. This further divides permissions into two categories, based on their protection level:

1. Install-time permissions: they are granted once during installation, and cannot be revoked. They include both normal permissions, which are always granted automatically, and signature permissions, which depend on the requesting app's certificate.
2. Runtime permissions: they have a dangerous protection level and must be requested as the app runs, with explicit user approval through a permission dialog.

4.2 Runtime Permissions

By involving an interaction with the user, *runtime permissions* are the most complex type in the Android permission model. These permissions require that prompts be designed to be quick and simple, understand user intentions, and be accessible to a general user to understand them. This complexity has led to multiple updates and adjustments throughout different Android versions, with changes made at various levels of the permission system to simplify the user experience and improve security.

4.2.1 Permission Dialog

The permission dialog is the primary interface through which users interact with Android's runtime permissions, and is presented every time apps request an unset permission. To keep interactions simple, the dialog hides complex permission logic mechanisms by condensing it into just two or three buttons, balancing user control with simplicity of understanding. Most of the times, the buttons are as shown in Figure 3:

- Allow/While using the app: it's usually the top button and permanently grants the permission for foreground access, marking it as a fixed setting that remains unless the user actively changes it.
- Don't allow: it's the button at the bottom of the dialog and rejects the permission request, restricting the app's access to the requested resource for the current session.

  In older Android versions, in order to permanently deny the permission, users needed to check an option to remember the choice. More recently, since Android 11, that checkbox is no longer provided and, instead, the choice is automatically made permanent by rejecting the same permission twice in a row.

- Only this time: for certain permissions that also involve a background access—like camera or location permissions—a third button may appear in the middle. This "Allow once" option grants access only for the current app session. For example, the camera permission applies to foreground access only, while a separate `BACKGROUND_CAMERA` permission can provide background usage. In such cases, selecting "Only this time" grants the permission temporarily, revoking it when the app is closed.

This approach simplifies complex permission management for users, allowing Android to communicate security options without overwhelming users with technical details.

It is also not the only way for users to manage runtime permissions. System settings provide a similar interface for setting permission statuses for each installed app, illustrated in Figure 4. Users are required to use settings to change permissions that had been already set, since the dialog will not be prompted anymore.

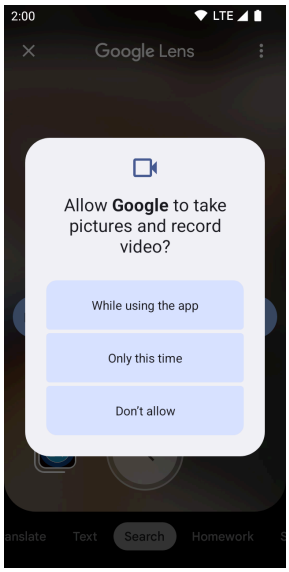

Figure 3: Permission dialog to request the camera permission.

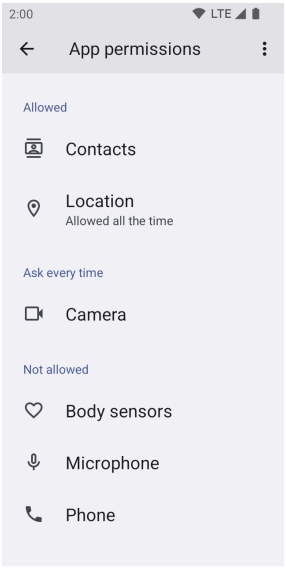

Figure 4: App permissions settings interface.

4.2.2 Permission Groups

In order to avoid repetitive requests of similar permissions, Android organizes them into *permission groups*, based on the type of data or resource they protect.

When accepting a permission request, Android saves the choice for the permission's group, so that the next time a request for the same group is performed, the user is not prompted the request dialog and the permission is automatically granted or rejected, based on its group status.

For example, when granting an app the "Read contacts" permission, the entire "Contacts" permission group is granted. This means that "Write contacts" (another permission in the "Contacts" group) is not granted yet, but it will automatically be at the app's request, without

user intervention. The reverse is also true: if the user denies a permission, every other permission in its group is immediately denied.

At a practical level, the behavior of permission groups makes request dialogs more closely tied to the group itself rather than to individual permissions. To reflect this, dialogs display the permission group's icon and description.

Starting from Android 11, static information about platform permission groups is no longer provided. Platform permissions are statically defined with an `UNDEFINED` group, with the actual one being set by the system at runtime. While it is still possible to determine which group a platform permission belongs to, this information must now be queried dynamically using the method `getGroupOfPlatformPermission()`. This change reflects a move towards more granular control over permissions and reduces reliance on predefined groupings.

4.2.3 Edge Cases

Since request dialogs hide some complexity of the underlying permission model, certain permissions and state combinations arise some peculiarities:

- `shouldShowRequestPermissionRationale`: when users deny a permission, they might inadvertently block a feature without fully understanding its importance. Android provides developers with the `shouldShowRequestPermissionRationale()` method to address this, allowing the app to display a rationale for the permission if the user rejected a request for it. This method returns `true` only if the user has denied the permission once, but not permanently.
- Dismissing the dialog: users can dismiss the permission dialog by tapping outside it, which leaves the permission unset without explicitly indicating user intent. In this scenario, Android neither marks the permission as "denied once" nor permanently denied, meaning that dismissing the dialog repeatedly will not lead to a fixed denial. As a result, `shouldShowRequestPermissionRationale()` returns `false`, which might mislead the app into interpreting this as if the permission dialog was never shown in the first place.
- Detecting fixed denials: the combination of `checkPermission()`—method that determines whether a permission is granted—and `shouldShowRequestPermissionRationale()` is the only way for third-party developers to infer permission statuses, because internal permission APIs are not accessible to normal apps. However, these methods alone are insufficient for deducing a permission's exact state, particularly for fixed denials: where a user has permanently denied the permission. This is a useful information, that an app might want to use to inform the user that it cannot request the permission anymore. In such cases, `checkPermission()` returns `PERMISSION_DENIED`, while `shouldShowRequestPermissionRationale()` returns `false`. Unfortunately, this combination can also occur if a user dismissed the dialog without explicitly denying the permission, leaving the app unable to differentiate between a permanent denial and a dialog dismiss.

This is a very specific case, but it creates an issue that cannot be solved completely without an official support from the OS. Developers exploit creative solutions to tamper it, but are only able to partially address the issue.

- Location group: permissions belonging to the "Location" permission group present some specific dialog layouts, introduced on various versions:
  - Android 10 introduced the tristate permission dialog [20], where users are given the choice to grant location permissions for foreground access only, or for background access too.
  - Android 12 introduced the possibility for users to choose between granting coarse or fine location access. As shown in Figure 5, it presents a different dialog based on the current state and permissions that are being requested, allowing to request the coarse location and then upgrade to the more precise version, or request both at the same time and let the user decide.
- Background permissions: with the introduction of the tristate location permissions on Android 10, background permissions appeared for the first time. They are permissions linked to their foreground counterpart, and are currently available for camera, microphone, and location permissions. They can be requested only after obtaining access to the foreground permissions. When requested they do not prompt a permission dialog. Instead, they redirect to a system settings activity (Figure 6), where users need to manually allow the permission.

Analyzing these behaviors is essential for understanding how Android's permission model affects app interactions with permissions. This analysis was also useful for developing a consistent behavior for the virtual permission model, that is presented in later chapters.

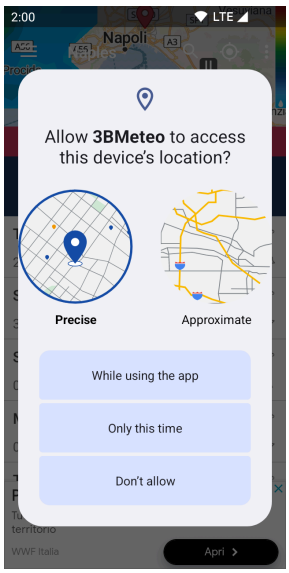

Figure 5: The new location permissions request dialog.

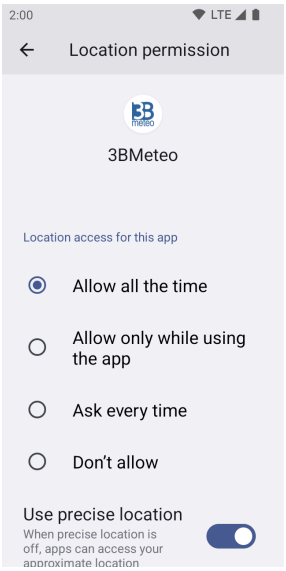

Figure 6: Allowing location access in the background from settings.

4.3 Implementation

As discussed in previous sections, the Android permission model is inherently complex—not only due to the intricate edge cases it has to address—but also because it needs to provide robust, system-level security features to protect sensitive resources. Additionally, with many updates and refinements over time, the model's implementation has grown into a large, continually evolving architecture, spread between multiple components and deep layers.

The following analysis is focused on understanding the model architecture at a higher level, by identifying the main components that have an active role in the logic behind permission checking, how runtime permissions are requested, and storing and managing the status of permissions. This architecture is taken as an inspiration for the virtual permission model, described in later chapters.

4.3.1 Main Classes

Until Android 6, permission handling was managed directly by the `PackageManager` service. Since permissions were only granted at install-time, a single, centralized manager was sufficient. With the introduction of runtime permissions, however, permission management became more complex, requiring it to be split between multiple dedicated components.

The following subsections describe the main classes involved in the current Android permission system, as shown in Figure 7. Each class is analyzed with respect to its responsibilities and interactions with other components, providing a comprehensive—although simplified—view of how the system operates.

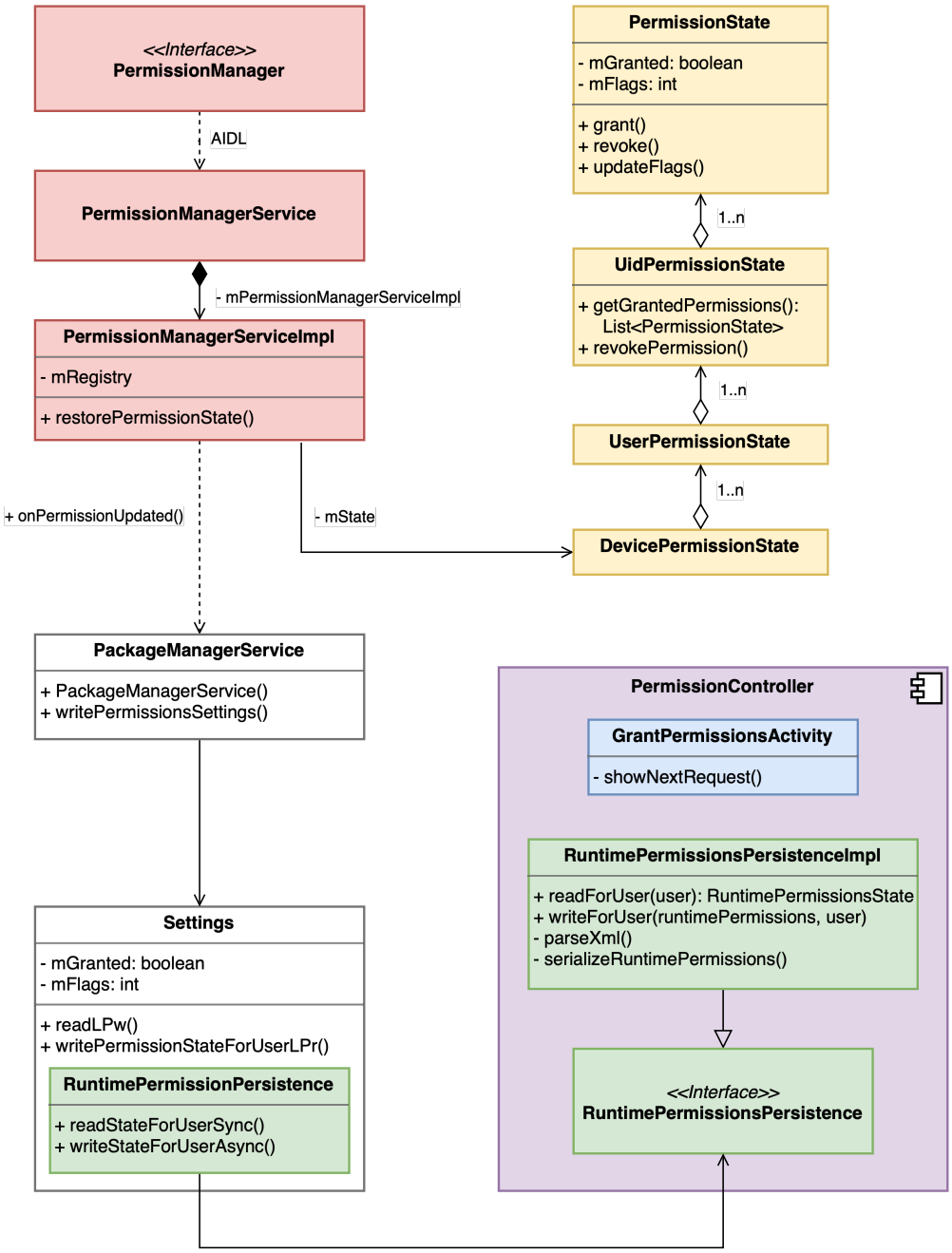

Figure 7: Simplified architecture of the permission model's classes.

`PermissionManager`. It is the main service interface for permission management. As a central access point, it provides high-level methods for managing permissions and is the entry point for applications and other services needing permission information or status updates. While normal apps cannot directly access most of its APIs, all permission-related operations are handled by this service at some point in the call stack, sometimes called to provide control over the permission

state to publicly accessible methods in `Context` or `Activity`, such as `checkSelfPermission()` and `requestPermissions()`.

`PermissionManager` was created to support the more complex needs of runtime permissions, shifting permission handling from the `PackageManager` service. As all Android services, it is implemented in a separate `PermissionManagerService` class, which allows modular implementations by relying on a specialized `PermissionManagerServiceInterface` interface, stored in the `mPermissionManagerServiceImpl` private field.

`PermissionManagerServiceImpl`. This is the underlying class implementing the detailed permissions logic exposed in `PermissionManagerService`. It has direct access to the current internal state of all permissions in its `mState` field. The state is modeled with multiple hierarchical classes:

- `PermissionState`: stores the current grant status and flags associated with a specific permission. It also provides methods to perform direct operations on the permission, such as `grant()`, `revoke()`, and `updateFlags()`.
- `UidPermissionState`: groups the state of permissions associated with a specific UID, and provides methods to interact with it, such as `getGrantedPermissions()` and `revokePermission()`.
- `UserPermissionState`: organizes the `UidPermissionState` instances for all applications installed under a specific user.
- `DevicePermissionState`: tracks the `UserPermissionState` associated with each user on the system.

The `DevicePermissionState` owned by `PermissionManagerServiceImpl` is initialized during boot via the `restorePermissionState()` method, loading stored permission data.

This class also owns an internal connection with user and package managers, to handle permissions in multi-user environments, and retrieve information about installed packages and their declared permissions.

Additionally, its `mRegistry` field manages an internal storage of information about all known permissions in the system and their related settings.

`RuntimePermissionsPersistenceImpl`. It is the latest implementation responsible for managing runtime permission data persistence. It is part of the `PermissionController` Android Pony EXpress (APEX) module that focuses exclusively on permission management, and is responsible for reading and writing the state of a user's permissions in its `runtime-permissions.xml` file.

While legacy code is still present across the framework, where permissions were handled by internal components using hidden APIs, this newer approach moves the functionality into a specialized module that can be upgraded independently from the system [21]. `RuntimePermissionsPersistenceImpl` specifically takes care of the actual parsing and serialization of the XML files that store permissions data for each user.

The permissions files group every permission requested by apps installed by a user, specifying the grant status and additional flags. They not only group runtime permissions, as the file name implies, but also install-time ones. Listing 1 provides an example where the Internet and NFC permissions are stored, which are normal permissions.

```
<!-- /data/misc_de/$userId/apexdata/com.android.permission/runtime-permissions.xml -->
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<runtime-permissions version="10" >
  <package name="com.example.testapp" >
    <permission name="android.permission.ACCESS_FINE_LOCATION" granted="false" flags="300" />
    <permission name="android.permission.BODY_SENSORS" granted="false" flags="301" />
    <permission name="android.permission.INTERNET" granted="true" flags="0" />
    <permission name="android.permission.ACCESS_COARSE_LOCATION" granted="true" flags="80301" />
    <permission name="android.permission.CALL_PHONE" granted="false" flags="300" />
    <permission name="android.permission.WRITE_CONTACTS" granted="false" flags="300" />
    <permission name="android.permission.NFC" granted="true" flags="0" />
    <permission name="android.permission.CAMERA" granted="true" flags="301" />
    <permission name="android.permission.RECORD_AUDIO" granted="false" flags="300" />
    <permission name="android.permission.READ_CONTACTS" granted="true" flags="301" />
  </package>
  <!-- Other packages... -->
</runtime-permissions>
```

Listing 1: Example of a permission XML file.

`RuntimePermissionsPersistenceImpl` exposes its features in two public methods:

1. `RuntimePermissionsState readForUser(UserHandle user)`.
2. `void writeForUser(RuntimePermisionsState runtimePermissions, UserHandle user)`.

`Settings`. The class dedicated to store system dynamic settings also acts as a link between the `PermissionController` module and system framework APIs. It does so by storing a reference of the `RuntimePermissionsPersistence` and wrapping it in an internal `RuntimePermissionPersistence` (note the missing 's'). This internal persistence manager defines APIs to interact with the actual `PermissionController` persistence, most notably the `readStateForUserSync()` and `writeStateForUserAsync()`. These are exposed in the `Settings` class in the methods:

- `readLPw()`: reads the entire settings for each user, including their permissions state. It is called once by `PackageManagerService` in its constructor, meaning that the file seems to be read once during the services initialization phase. Once the permissions state is loaded into memory it is managed there and, eventually, it will be written back to file as updates occur.
- `writePermissionStateForUserLPr()`: writes the permissions state of a specific user. It is called by the `PackageManagerService` method `writePermissionSettings()`, which, in turn, is called from `PermissionManagerServiceImpl` via its `onPermissionUpdated()` callback whenever a permission is modified.

`PermissionController`. This is the module dedicated to managing permissions-related UI interactions and system logic. It is the main component addressing user-centric tasks, regarding the granting process and permission policies in general. Its main responsibilities are:

- Managing permission requests: this is done mainly in the `GrantPermissionsActivity`, which is the one creating the dialogs presented to users when applications request runtime permissions. Its purpose is to bridge the interaction between apps and the permission model, handling user inputs and interact with the model accordingly.
- Permission granting and group logic: it handles the granting logic, especially for runtime permissions within groups. When handling a permission request, it checks whether a permission in the same group is already granted. If not, it manages the change in the permission state.
- Group revoking: a specific case to manage for permission groups is the possibility for them to be revoked. When revoking a permission, the module has to extend the operation to all other permissions in the group. It is also possible to revoke a group directly from the settings. The grant status of all permissions inside of it has to be updated and managed correctly.
- Auto-revoke mechanism: it also implements the auto-revoke of permissions, for apps that were not being used for an extended period of time.

4.3.2 Functional Components

The architecture described in Section 4.3.1, even while being a simplification, contains several details that may be complex to keep in mind. It may be useful to categorize the individual classes into higher-level logical components, based on the different functional roles that can be found in the model.

Figure 8 shows the components described in the following subsections.

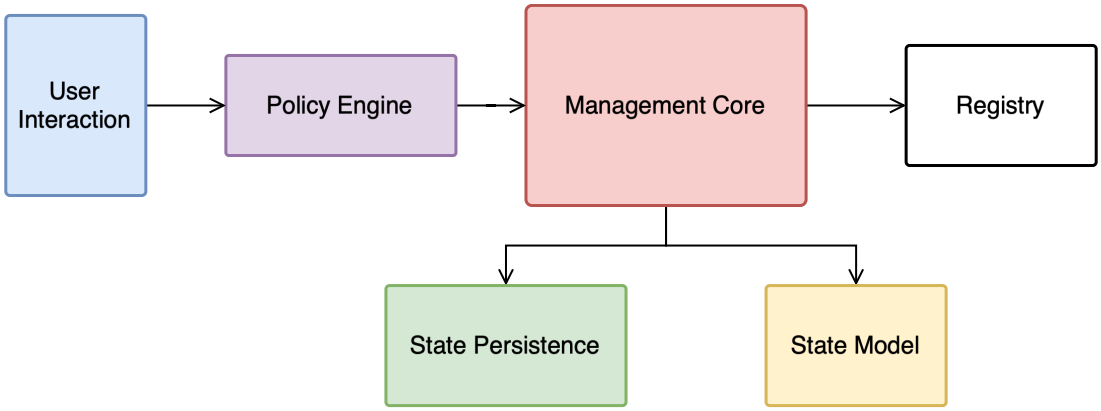

Figure 8: The functional components and their interactions.

Management Core. It provides a centralized control for querying and modifying the state of permissions. It serves as the primary interface for all permission-related operations and acts as the entry point for other system services and applications to interact with the permission model. Typical operations it should implement are:

- Checking permission status for specific permissions, UIDs, or users.
- Granting or revoking permissions and manage their flags.

- Retrieving metadata about permissions or permission groups.

It is implemented in:
- `PermissionManager`.
- `PermissionManagerService`.
- `PermissionManagerServiceImpl`.

State Model. It defines the in-memory representation of permission data. It provides the foundation for managing and manipulating permission states. The data structures in this component reflect the hierarchy and relationships within the permission system, and should:
- Track individual permissions, including their grant status and flags.
- Manage the state of permissions associated with specific UIDs.
- Aggregate UID-level states for each user in the system.
- Support multi-user environments by handling permission data for each user.

It is implemented in:
- `PermissionState`.
- `RuntimePermissionState`.
- `UidPermissionState`.
- `UserPermissionState`.
- `DevicePermissionState`.

State Persistence. It maintains a persistent record of permission states across system reboots by:
- Storing permission states in user-specific files.
- Loading permission data during system initialization.
- Writing updates to the storage layer whenever permissions are updated.

It is implemented in:
- `RuntimePermissionsPersistence`.
- `RuntimePermissionsPersistenceImpl`.
- `RuntimePermissionPersistenceImpl`.
- The `runtime-permissions.xml` file.

Policy Engine. It is the decision-making layer, enforcing rules and constraints on permission operations, and ensuring that all actions are aligned with system policies and security requirements.

It needs to implement the logic closer to user interactions, such as group-based constraints for granting and revoking permissions.

It is implemented in the `PermissionController`, more specifically in its service.

User Interaction Layer. This layer fills the gap between the permission system and the user. It handles user-facing operations, ensuring that the system is able to communicate permission requirements and decisions clearly.

Its main use-case is presenting permission request dialogs to users, following a group-based permission logic, and managing their input.

It is implemented in the `GrantPermissionsActivity` of the `PermissionController` module.

Registry. It maintains a catalog of all known permissions and their attributes, providing metadata about each permission and supporting querying operations needed by other components.

This component is not explored in detail, since the virtual permission model is able to re-use Android's implementation, so a deep understanding is not needed.

5

Securing the Virtual Apps

5.1 Preliminary Work

Before introducing the virtual permission model, some preliminary work had to be addressed. VirtualXposed has been currently updated to support Android versions up to Android 12 in its official open-source repository, with a release called “Initial support for Android 12”. Simultaneously, a developer working on a personal fork of VirtualXposed also independently updated it to Android 12. Although both of these VirtualXposed versions had compatibility issues, they were complementary in some areas. A merge of these two took the best of both worlds, applying fixes from one version to the other. This created a starting point, where VirtualXposed had a sufficient Android 12 support, compared to older versions. At that time, however, the latest version was Android 14, so further work had to be done, in order be able to compare the virtual environment with the native one on a current version.

5.1.1 Android 14 Support

In its initial form, VirtualXposed was incompatible with the latest version, not even being able to install, because of multiple errors detected in the manifest. This happened because the app was originally developed for older Android versions, and updates had introduced mandatory changes.

Required changes in the app included:

- Explicit exported components: Android 12 and later versions require that components like activities or receivers explicitly declare their export status, when they have intent filters.

```
<activity android:name=".sys.ShareBridgeActivity"
    android:exported="true"
    android:label="@string/shared_to_vxp"
    android:taskAffinity="${applicationId}.share"
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="*/*" />
    </intent-filter>
</activity>
```

Listing 2: Activity that required an explicit export status.

- Service type declaration: starting with Android 14, foreground services must specify a service type. VirtualApp's daemon service had to be updated to include this requirement.

```
<uses-permission android:name="android.permission.FOREGROUND_SERVICE_SPECIAL_USE"
    android:minSdkVersion="34" />
<application>
    <service android:name="com.lody.virtual.client.stub.DaemonService"
        android:process="@string/engine_process_name"
        android:foregroundServiceType="specialUse" />
    ...
</application>
```

Listing 3: Updating the daemon service.

- Service notification: similarly, since Android 8, services running in the background are required to display a notification. The app was not doing it properly, so the feature was not working as expected.

```
getSystemService(NotificationManager.class)
    .createNotificationChannel(new NotificationChannel(
        CHANNEL_ID,
        "Daemon service notification",
        NotificationManager.IMPORTANCE_DEFAULT
    ));
Notification notification = new Notification.Builder(this, CHANNEL_ID)
    .setContentTitle("Daemon service")
    .setSmallIcon(android.R.drawable.ic_dialog_info)
    .build();
```

Listing 4: The right way of creating the notification.

- Use additional permissions: additional permissions were introduced over various system updates, and were necessary to provide specific features to plugin apps and the container app itself. These permissions include `QUERY_ALL_PACKAGES` and `ACCESS_BACKGROUND_LOCATION`.

Once these fixes were applied, installation could proceed successfully, but the app still manifested many issues. Initially, it would crash on startup, and even after fixing those errors, many exceptions still occurred, especially when starting virtual apps. Many adjustments had to be done in the virtualization framework, in order to fix all these issues.

Most of these problems were caused by changes in the Android API, especially in non-SDK interfaces (hidden APIs). Android actually introduced them in the first place to allow Android's developers to make structural changes or improvements internally, without impacting third-party apps.

When regular apps use libraries to bypass restrictions and access hidden methods, they still need to rely on reflection to call these methods. This exposes them to breaking changes in the API, as reflection uses strings to invoke methods, which cannot be checked at compile-time, leading to potential runtime errors. VirtualApp, which heavily relies on this mechanism, encountered frequent crashes, introduced by breaking changes in the Android SDK codebase.

Most of the times, these crashes were caused by changes in the signature of methods, such as new parameters being introduced, or their type changed. Examples of typical fixes are the following:

- Field type change: the update to Android 14 changed the type of the private field `mActions` of `IntentFilter` from `java.lang.ArrayList` to `android.util.ArraySet`. This change required to introduce some code forking into existing functions.

```
public static void protectIntentFilter(IntentFilter filter) {
    List<String> actions = mirror.android.content.IntentFilter.mActions
        .get(filter);
    ListIterator<String> iterator = actions.listIterator();
    while (iterator.hasNext()) {
        String action = iterator.next();
        if (SpecialComponentList.isActionInBlackList(action)) {
            iterator.remove();
            continue;
        }
        if (SYSTEM_BROADCAST_ACTION.contains(action)) {
            continue;
        }
        String newAction = SpecialComponentList.protectAction(action);
        if (newAction != null) {
            iterator.set(newAction);
        }
    }
}
```

Listing 5: Existing implementation of a method in the framework.

```
public static void protectIntentFilter(IntentFilter filter) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.UPSIDE_DOWN_CAKE) {
        ArraySet<String> actions = mirror.android.content.IntentFilter
                .mActionsUpsideDownCake.get(filter);
        for (String action : actions) {
            if (SpecialComponentList.isActionInBlackList(action)) {
                actions.remove(action);
                continue;
            }
            if (SYSTEM_BROADCAST_ACTION.contains(action)) {
                continue;
            }
            String newAction = SpecialComponentList.protectAction(action);
            if (newAction != null) {
                actions.remove(action);
                actions.add(newAction);
            }
        }
    } else {
       // Previous implementation...
    }
}
```

Listing 6: The new method implementation, adapted for changes.

- Internal class becomes independent: Android 14 moved the class `android.content.pm.PackageParser$SigningDetails` to a dedicated class `android.content.pm.SigningDetails`.
  This caused the constructor `SigningInfo(SigningDetails details)` not to be found using reflection. A new mirror class (Listing 7) and a dedicated constructor for `SigningInfo` (Listing 8) had to be introduced, in order to create a `SigningInfo` object, using a `SigningDetails` parameter of the right type.

  As shown in Listing 9, this type of change requires some code forking, based on the current Android version.

```
public class SigningDetails {
    public static Class<?> TYPE = RefClass
        .load(SigningDetails.class, "android.content.pm.SigningDetails");
    @MethodReflectParams({
        "[Landroid.content.pm.Signature;",
        "int",
        "[Landroid.content.pm.Signature;"
    })
    public static RefConstructor<SigningDetails> ctor;
}
```

Listing 7: Dedicated `SigningDetails` mirror class.

```
@MethodReflectParams("android.content.pm.PackageParser$SigningDetails")
public static RefConstructor<android.content.pm.SigningInfo> ctor;
// Android 14 moved SigningDetails to a dedicated class
@MethodReflectParams("android.content.pm.SigningDetails")
public static RefConstructor<android.content.pm.SigningInfo> ctorUpsideDownCake;
```

Listing 8: The two `SigningInfo` constructors in its mirror class.

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.UPSIDE_DOWN_CAKE) {
    Signature[] signatures = mirror.android.content.pm.PackageParser
        .SigningDetails.signatures.get(signingDetails);
    Integer signatureSchemeVersion = mirror.android.content.pm.PackageParser
        .SigningDetails.signatureSchemeVersion.get(signingDetails);
    Signature[] pastSigningCertificates = mirror.android.content.pm.PackageParser
        .SigningDetails.pastSigningCertificates.get(signingDetails);
    // Create Android 14 version of SigningDetails
    signingDetails = mirror.android.content.pm.SigningDetails.ctor
        .newInstance(signatures, signatureSchemeVersion, pastSigningCertificates);
    // Use Android 14 constructor
    cache.signingInfo = mirror.android.content.pm.PackageParser
        .SigningInfo.ctorUpsideDownCake.newInstance(signingDetails);
} else {
    cache.signingInfo = mirror.android.content.pm.PackageParser
        .SigningInfo.ctor.newInstance(signingDetails);
}
```

Listing 9: Code snippet that includes the `SigningDetails` update.

- Parameter changes: methods changed their parameters in multiple occasions, adding or removing them, causing vague errors to be raised. These provided no explanation about the issue, requiring a manual analysis of the source code of every new Android version, to spot the difference in a method signature.

  An example of this issue, as shown in Listing 10, is the introduction of the `deviceId` parameter in `android.app.ClientTransactionHandler.handleLaunchActivity`.

```
public abstract Activity handleLaunchActivity(ActivityThread.ActivityClientRecord r,
        PendingTransactionActions pendingActions, Intent customIntent);
// Android 14
public abstract Activity handleLaunchActivity(ActivityThread.ActivityClientRecord r,
        PendingTransactionActions pendingActions, int deviceId, Intent customIntent);
```

Listing 10: `handleLaunchActivity` method signature change.

5.1.2 Build System Update

The removal of a required dependency from Maven repositories prompted the need to update the build system for the project. Fortunately, since the missing dependency was open-source, it was possible to compile it manually and embed it locally into the project. However, this necessitated an update to the build toolchain, which, in turn, required broader updates to several outdated project components:

- The Gradle build system was updated to support more recent Android plugin and wrapper versions. This also introduced an automatic method for selecting the JDK to use for compilation, which had previously needed to be set manually in the environment. The update significantly improved compilation speed and overall user experience, while also triggering the need for additional changes across the project.
- Dependencies were fixed by compiling the missing one and setting it for local use.
- The project was migrated to AndroidX libraries, replacing older support libraries. This operation required extensive refactoring across the large codebase and enabled the use of newer components and dependencies versions, which were previously tied to the old libraries.
- Java language support was updated, as the VirtualApp component was previously still using Java 7. The new setup now supports the latest Java features, including functional interfaces and switch expressions.

These updates were essential for improving the development environment, making it more efficient to use and easier to manage overall.

5.1.3 Multi-User UID System

One of the main features of VirtualApp is the possibility to install multiple copies of a same application. The framework's original implementation did not assign cloned apps a dedicated UID. Instead, a same app would always have the same UID, even when installed under a different virtual user to act as a clone. While this design worked fine for VirtualApp itself, since it did not actively use them, it was a limit for implementing the virtual permission model. Permissions in Android are managed by UID, so without a unique one, it would not have been as straightforward to manage permissions separately for individual instances of an app.

In order to address this, the UID system in VirtualApp was updated to align with Android multi-user support approach. In a multi-user Android system, each installed app is assigned a unique application-specific UID in the range 10000–19999 [22]. This is then composed with the current user ID to generate a unique app-user combination. The range of UIDs assigned per user is 100000 [23], which means that the final result is calculated as:

$$userId \times 100000 + appId$$

For example, an app with UID 10005 would retain it for the default user (user 0). Instead, for user 2, it would be 210005. This is conveniently done in the hidden API method `public static int getUid(int userId, int appId)` of the class `UserHandle`, which VirtualApp provides as visible in its `VUserHandle` class.

The change was introduced in the `getOrCreateUid()` method of the `UidSystem` class, as shown in Listing 11 and Listing 12.

```
public int getOrCreateUid(VPackage pkg) {
    String sharedUserId = pkg.mSharedUserId;
    if (sharedUserId == null) {
        sharedUserId = pkg.packageName;
    }
    Integer uid = mSharedUserIdMap.get(sharedUserId);
    if (uid != null) {
        return uid;
    }
    int newUid = ++mFreeUid;
    mSharedUserIdMap.put(sharedUserId, newUid);
    save();
    return newUid;
}
```

Listing 11: Old VirtualApp UID creation method.

```
public int getOrCreateUid(int userId, VPackage pkg) {
    String packageName = pkg.mSharedUserId;
    if (packageName == null) {
        packageName = pkg.packageName;
    }
    // Get existing application UID for package name
    Integer appId = mPackageUidMap.get(packageName);
    if (appId == null) {
        // Create new application UID
        appId = ++mFreeUid;
        mPackageUidMap.put(packageName, appId);
        save();
    }
    // Return actual UID for current user
    return VUserHandle.getUid(userId, appId);
}
```

Listing 12: New VirtualApp UID creation method for multi-user system.

5.2 Design Overview

The virtual model retains Android's permission model's key, high-level components outlined in Section 4.3.2, with some adjustments to adapt them to a simpler—smaller in scope—implementation in a virtual environment. These are present because the system's implementation has to address many system-level access cases, where apps might want to perform tasks that require them to be granted elevated permissions, or elevated users might try to access protected resources, needing a different treatment than normal ones. VirtualApp, operating in the normal user space, has not access to privileged resources in the first place, thus it is not required to address these edge cases.

The main difference with the system's model is the *policy engine* not being centralized. Since its logic is simpler, it is distributed across different components. Certain responsibilities are

included in user interaction and the management core, while other elements are directly embedded into the state model.

Additionally, it is worth noting that the *registry* component is mostly publicly accessible using Android APIs, making a full re-implementation unnecessary. Instead, only few specific aspects of the registry are included inside the *state model*, to address the limited specific needs required by the virtual model.

Finally, the virtual model introduces a new *redirection* component, which provides the necessary link between virtual apps' standard behavior—based on Android's permission model—and the virtual permission model.

Together, these considerations lead to the definition of five primary components in the virtual permission model, as illustrated in Figure 9:

1. State model.
2. State persistence.
3. Management core.
4. User interaction.
5. Redirection.

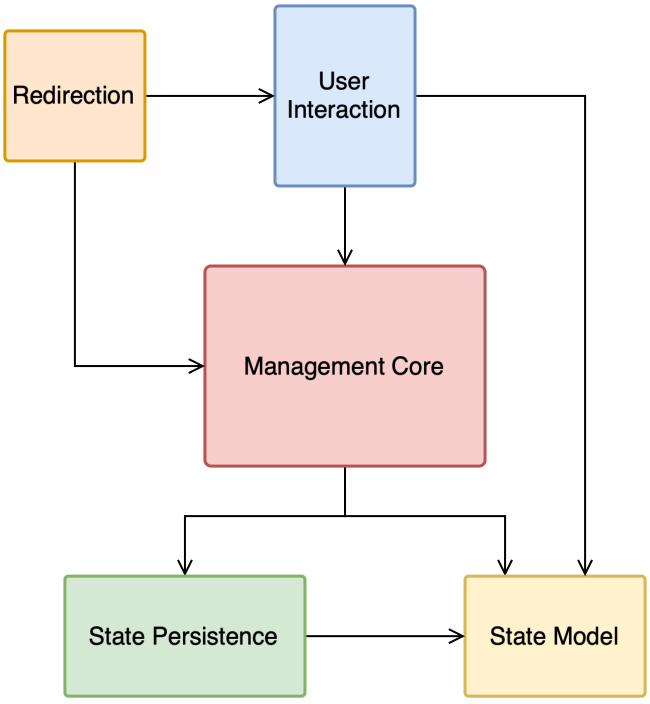

Figure 9: The components of the virtual model and their interactions.

5.3 Components Analysis

This section explores the components of the virtual permission model, starting with an overview of their design and then moving into a detailed examination of their implementation.

Each following subsection is dedicated to a specific component and is structured in two parts:

1. Design: describes the component's design, outlining its responsibilities and interactions with other components in the system. It also addresses special cases that the component must handle, with a focus on how the behavior differs from its equivalent in the Android system.
2. Implementation: explains how the design concepts are realized in practice, providing a detailed look at the classes that implement previously identified design requirements. It analyzes key methods and presents example code snippets. A class diagram of the component is always included, offering a detailed look at classes architecture and their interactions.

Figure 10 presents a simplified view over all the classes providing an implementation to the design requirements.

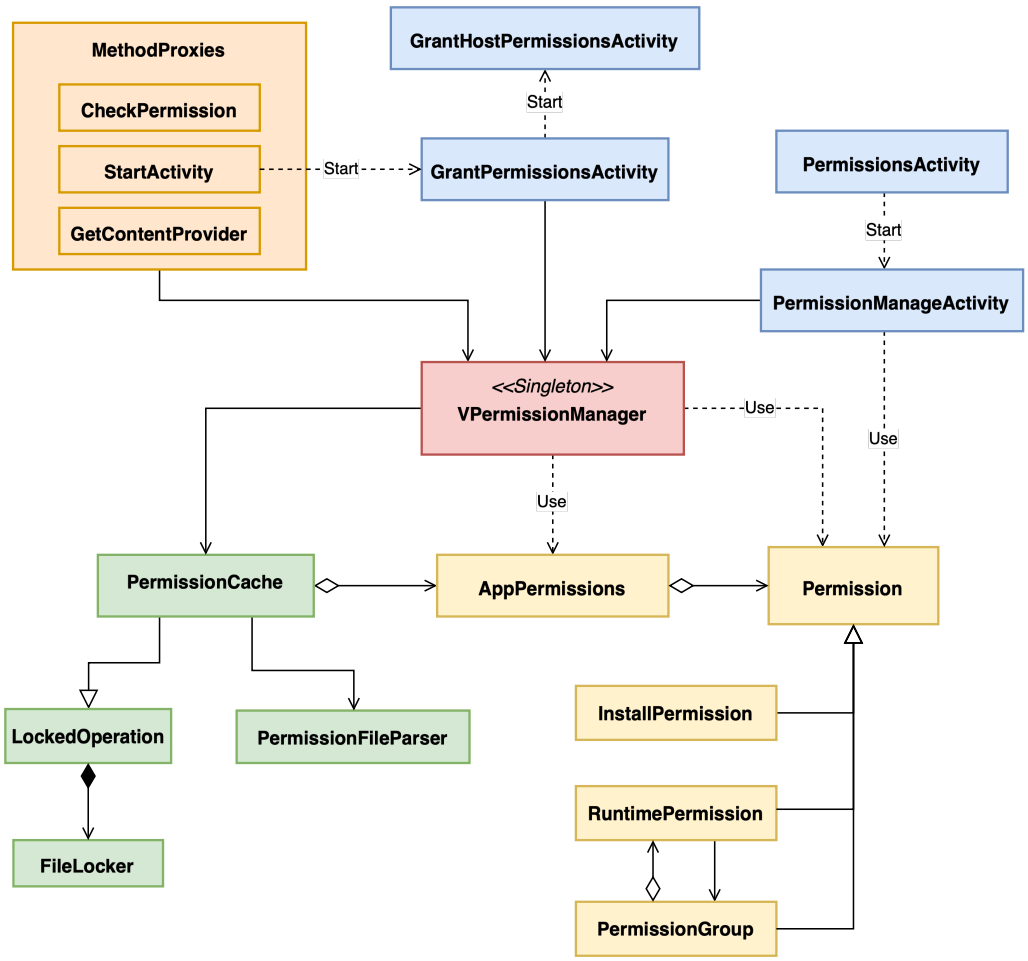

Figure 10: Simplified architecture of the virtual permission model's classes.

5.3.1 State Model Component

Design. It defines the data structures needed to store and manage permission-related information. It supports three key aspects of permissions: install-time permissions, runtime permissions, and permission groups. State is managed differently, based on their type.

Additionally, permissions are organized in a hierarchy to group them under multiple UIDs and users.

Install-time Permissions. They have a simple, binary state: they either are granted or not. These permissions are typically static, reflecting whether they were approved at the time of installation, but, they could also theoretically support updates. This is aligned with Android's approach, since information about install-time permissions is stored alongside runtime permissions in the `runtime-permissions.xml` file, making status changes theoretically possible if needed.

Beyond the basic granted/not-granted status, no additional metadata is required for their management.

Runtime Permissions. They are more complex, requiring the state model to store detailed information about their current status and history of changes.

The possible states for a runtime permission are:

- Unrequested (default): the permission has not been requested by the application.
- Granted: the permission has been permanently granted and will not be requested again, unless the user explicitly changes its status from the settings. Auto-revoking permissions are not addressed in the virtual model.
- Denied once: the user rejected the last permission request, or explicitly denied it—or its group—from the settings. The next permission request will still prompt a permission request dialog.
- Permanently denied: after being denied once, the permission request has been rejected again. Further requests will not prompt a permission request dialog, unless the user explicitly changes its status from the settings.
- Always ask (not granted): the permission has been granted once in a previous session, or it has been set as "Always ask" from the settings.
- Always ask (granted for current execution): the permission is granted for the current session, but will need to be requested again in the future.

Additional details for runtime permissions have to be addressed:

- The "Denied once" status has to be reset when permissions transition to a more permissive status, like "Always ask" or "Granted".
- The "Granted once" status, assigned when a permission that is set to "Always ask" is granted for the current session, has to be reset between different app executions and when transitioning between other states.

In the context of the virtual permission model, runtime permissions are allowed to be overridden, that is, they can be assigned a status diverging from their current group's status. This provides more granularity on user's permission control, but still partially aligning to Android's model.

Permission groups. They reference multiple runtime permission records and are used to reduce the dependency from Android register APIs. Their possible state can be:

- Unrequested (default): none of the permissions in the group have been set. Permissions could have been denied once, though.
- Granted: a permission in the group has been granted, or the group has been granted from the settings. Further requests for the other permissions in the group will not prompt a dialog and automatically grant the permission.
- Denied: a permission has been permanently denied. Further requests for the other permissions in the group will not prompt a dialog and automatically deny access to the permission.
- Always ask: a permission in the group has been granted once, or the group status has been set to "Always ask" from the settings.

Hierarchical Structure for Permissions. The state model organizes permissions into hierarchical collections that store permissions declared by a specific UID. Unlike Android's implementation, a user-dedicated structure is not intended to be defined, since user information is included in the UID itself.

Implementation. The component consists of a hierarchy of classes, inheriting from the abstract `Permission` base class, and of a UID permissions container called `AppPermissions`. The three permission types have a dedicated class each, specializing the base implementation of `Permission`.

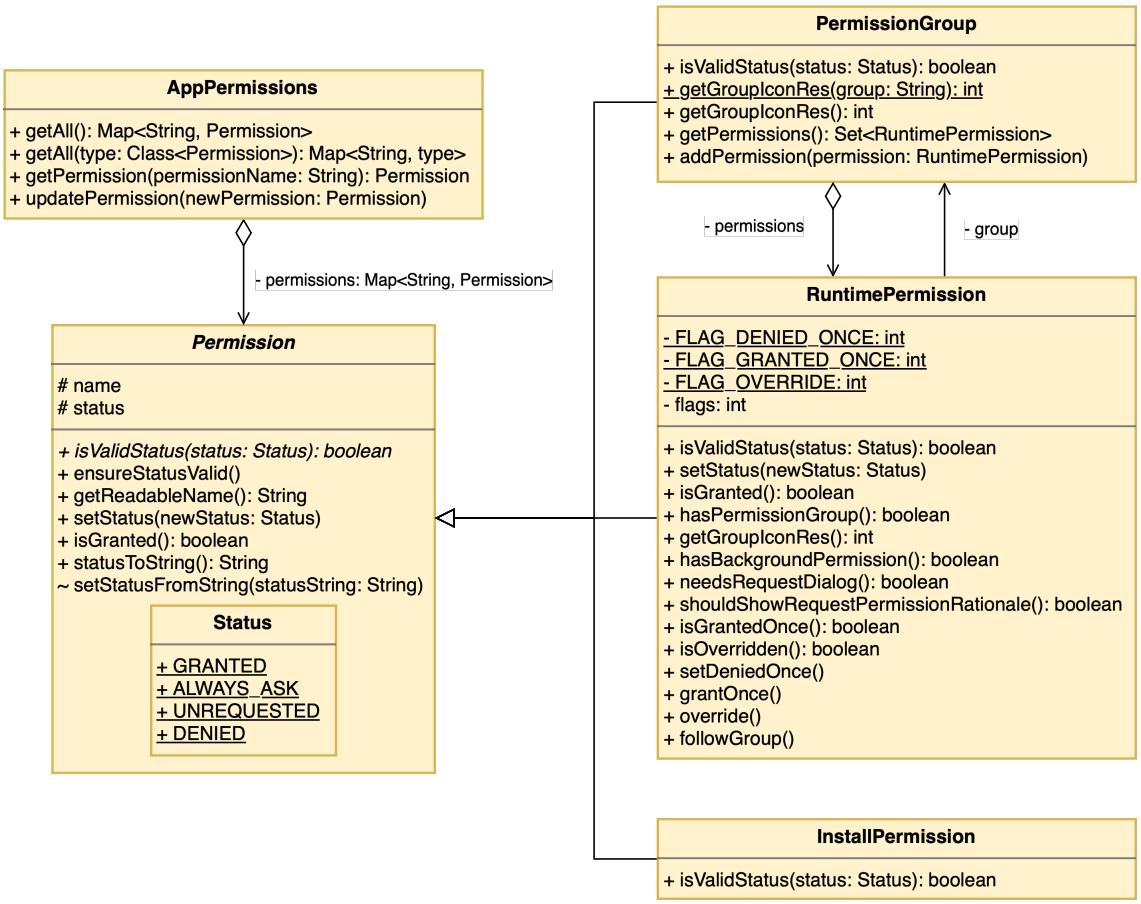

Figure 11: State model component class diagram.

Permission. This class serves as the abstract base for managing permissions in the state model. It is meant to be extended by the concrete permission types, providing a flexible interface for permission management. This abstraction ensures safe and robust permission states handling, validation, and transitions, while simplifying integration with the management core component.

This class also interacts with the state persistence, providing utility methods for simplifying its tasks by hiding the complexity of the permission state logic.

The main features it provides are:

- `PermissionStatus` enum: defines the possible states a permission can assume:
  - `GRANTED`: the permission has been permanently granted.
  - `ALWAYS_ASK`: the permission is set to always prompt for user approval or it was granted for the current session.
  - `UNREQUESTED`: the permission has not yet been requested.
  - `DENIED`: the permission has been permanently denied, either by the user or via system settings.

- Constructors: they support various initialization scenarios:
  - Create permission instances with a name and status (as an enum or string).
  - Duplicate an existing permission.
  - Validate inputs, ensuring that the initial state complies with the specific permission type's constraints.
- Logic methods: they encapsulate the logic behind possible permission states and status transitions:
  - `boolean isGranted()`: check if the permission is currently granted, based on its status and other possible state parameters.
  - `boolean isValidStatus(Status status)`: designed to check whether a status can be assigned to the permission, considering its type and current state.
  - `ensureStatusValid()`: ensures the status is consistent, throwing an exception if invalid.
- Getter and setter methods:
  - `String getName()`: returns the name of the permission.
  - `Status getStatus()`: returns the current status of the permission.
  - `setStatus(String newStatus)`: allows setting the permission's status, ensuring that status transitions are valid.
- Helper methods:
  - `getReadableName()`: formats the permission name into a user-friendly representation, ideal for displaying it in the UI.
  - `statusToString()`: converts the current status to a string representation, useful for supporting the serialization process.
  - `setStatusFromString(String statusString)`: sets the permission's status based on a string input, ensuring that the value represents a valid status. It is used to support the parsing process.

`InstallPermission`. It extends the `Permission` base class to represent install-time permissions. The only notable addition is its overridden `isValidStatus()` method, which restricts valid states to `GRANTED` and `DENIED`. This reflects the simple nature of install-time permissions' binary state, as noted in the design. No additional methods or functionality are needed, as the base class already supports all the features required for install-time permissions.

`RuntimePermission`. The `RuntimePermission` class is a detailed specialization over the base class. Since runtime permissions are inherently much more complex than install-time permissions, the class introduces several specific concepts:

- Flags for complex state modeling: inspired by Android's implementation, runtime permissions model additional status modifiers with a set of flags, composed in the `flags` field to

allow multiple combinations, which would need several dedicated enum values to be represented as the main status. The possible flags are:

- `FLAG_DENIED_ONCE`: indicates that the permission has been rejected in the last user interaction. This flag helps determine whether the app should display a rationale when requesting the permission again.
- `FLAG_GRANTED_ONCE`: indicates that the permission has been granted for the session, useful for handling cases where permissions are granted temporarily.
- `FLAG_OVERRIDE`: indicates that the permission's behavior overrides its group's status, allowing for individual permission control.

- Permission group integration: runtime permissions are associated with a `PermissionGroup`. This can simplify state management, giving a direct access to the group's status, which is necessary to evaluate the actual runtime permission status. Additionally, it is useful for supporting cases where group information is more relevant, such as in permission dialogs which show the group's icon and description.

  To provide support for this concept the class includes:

  - A nullable `group` private field, accessible with its dedicated `getGroup()` method.
  - `boolean hasPermissionGroup()`: determines whether the runtime permission is associated with a group.
  - `int getGroupIconRes()`: returns the resource id of the permission's group icon, used by the user interaction component.

- Enhanced state management:

  - `boolean isValidStatus()`: implements the abstract method to validate the consistency between the permission's status and its group's status, managing edge cases such as temporary grants (`ALWAYS_ASK`) in conjunction with group overrides.
  - `boolean isGranted()`: overrides the default implementation, by accounting for additional cases that depend on group's status and temporary grants.

```
@Override
public boolean isGranted() {
    return switch (status) {
        case Status.GRANTED -> isOverridden() || group.isGranted();
        case Status.ALWAYS_ASK -> isGrantedOnce()
            || ( hasPermissionGroup() && group.isGranted() );
        default -> false;
    };
}
```

Listing 13: `isGranted` method implementation for runtime permissions.

  - `shouldShowPermissionRationale()`, `isGrantedOnce()`, and `isOverridden()` return the current status of the relative flags, while `setDeniedOnce()`, `grantOnce()`, `override()`,

and `followGroup()` allow for setting—or unsetting, in the case of `followGroup()`—a specific flag.

```
public boolean shouldShowRequestPermissionRationale() {
    return (flags & FLAG_DENIED_ONCE) != 0;
}
```

Listing 14: Method returning the current status of a flag.

```
public void grantOnce() {
    setStatus(Status.ALWAYS_ASK);
    flags |= FLAG_GRANTED_ONCE;
}
```

Listing 15: `grantOnce` method, updating the status and setting a flag.

  - `statusToString()` and `setStatusFromString()` extend the base class's functionality to handle serialization and parsing of flags, ensuring an exhaustive state representation.

- Utility methods:
  - `boolean needsRequestDialog()`: determines whether a permission dialog should be displayed for requesting the permission, based on its current complete state.
  - `boolean hasBackgroundPermission()`: checks if a runtime permission has an associated background permission, mainly used to determine how buttons should be displayed in the request dialog.

`PermissionGroup`. The `PermissionGroup` class provides a higher-level abstraction for managing sets of related runtime permissions. It enables a hierarchical approach to permission management, where permissions can inherit or override the group's status.

The class is mostly used in user interaction, to retrieve informations like its display icon. It is also used when granting or revoking a batch of permissions, for example when denying a group from the settings.

The following are its main features:

- Permission aggregation: it stores a set of associated runtime permissions in its `permissions` private field, providing a mapping from groups to individual permissions. It defines two simple methods to manage this set:
  1. `Set getPermissions()`: returns a shallow copy of the group's permissions. This avoids returning a direct reference to the internal set, but allows management for the individual permissions as they still reference the original ones.
  2. `addPermission(RuntimePermission permission)`: adds a runtime permission to the group, but only if it's not a background permission, because those require individual management. Including a background permission would cause the foreground one to automatically grant access to it, since they belong to the same group.

- Group icon retrieval: the class provides a method to fetch the group's icon from Android's official resources, using `getGroupIconRes()` to get access to the internal names. Since permission group icons mostly follow a consistent naming pattern (`perm_group_$groupName`), their retrieval can be partially automated in the implementation of `getGroupIconRes(String group)`, as demonstrated in Listing 16.

```
public static int getGroupIconRes(final @Nullable String group) {
    final var resources = VirtualCore.get()
        .getContext()
        .getResources();
    final int defaultId = resources
        .getIdentifier("ic_perm_device_info", "drawable", "android");
    if (group == null) {
        return defaultId;
    }
    final var groupName = group
        .replaceAll("[a-z0-9.-]*", "")
        .toLowerCase();
    // Handle exceptions
    final var res = switch (groupName) {
        case "phone" -> "perm_group_phone_calls";
        case "notifications" -> "ic_notifications_alerted";
        default -> "perm_group_" + groupName;
    };
    final int id = resources.getIdentifier(res, "drawable", "android");
    if (id == 0) {
        // Use default permission icon if id not found
        return defaultId;
    }
    return id;
}
```

Listing 16: `getGroupIconRes` method implementation, providing an algorithm to retrieve original permission groups icons.

`AppPermissions`. The `AppPermissions` class provides a container for managing and interacting with all permissions associated with a specific UID.

Here's a concise summary of its functionality:

- Permission storage: the class maintains a map in its `permissions` private field, where each permission is keyed by its name for quick lookups and efficient management.
- Access and filtering:
  - `Map getAll()`: returns a direct reference to all stored permissions.
  - `Map getAll(Class type)`: returns a filtered map of all permissions of a specific type. For example, it can be used to retrieve a complete map of all runtime permissions with `getAll(RuntimePermission.class)`.

```
public <T extends Permission> Map<String, T> getAll(final Class<T> type) {
    return permissions.entrySet()
        .stream()
        .filter(entry -> type.isInstance(entry.getValue()))
        .collect(Collectors.toMap(Map.Entry::getKey, entry -> (T) entry.getValue()));
}
```

Listing 17: `getAll(type)` method implementation, filtering permissions by type.

  - `Permission getPermission(String permissionName)`: returns a specific permission by name.

- Permission updates:
  - `updatePermission(Permission newPermission)`: replaces a permission in the map with a new instance if a matching permission, identified by name and type, already exists. This approach ensures the permission is fully updated with the new instance, while maintaining consistency by only modifying declared permissions.

5.3.2 State Persistence Component

Design. The *state persistence* component is responsible for managing the interaction with the permission file that stores the state model. It has two main responsibilities:

1. Concurrent access management: handling concurrent file access, since virtual apps operate in separate processes and may attempt to read or write the file simultaneously.
2. File parsing: reading and writing the permission state to and from the file, ensuring the state model reflects the latest data.

To achieve this, the component ensures thread and process-safe operations, using locking mechanisms to prevent conflicts or data corruption. It abstracts these complexities, providing a simple interface for other components to access and modify the permission state as needed.

These mechanisms are necessary because virtual apps run in separate processes, thus stronger synchronization means like file system locks are necessary to avoid inconsistencies.

Implementation. The component's implementation reflects its design responsibilities by realizing the following classes:

1. Concurrent access management: this aspect is divided between two classes:
   - `FileLocker`: provides low-level functionality for acquiring and releasing shared or exclusive locks on files, ensuring safe operations when multiple processes or threads interact with the same file.
   - `LockedOperation`: abstracts `FileLocker`'s mechanisms into a higher-level framework, defining a lifecycle for file interactions.
2. File parsing: the `PermissionFileParser` class realizes the serialization and parsing of permission data between the state model and the persistence file.

The `PermissionCache` class is the primary interface for interactions with the state persistence, sitting between the state model and persistence components. It manages the in-memory representation of permissions while ensuring synchronization with stored data, providing efficient access for querying or updating the model.

The four classes, illustrated in Figure 12, are presented in detail in the following subsections.

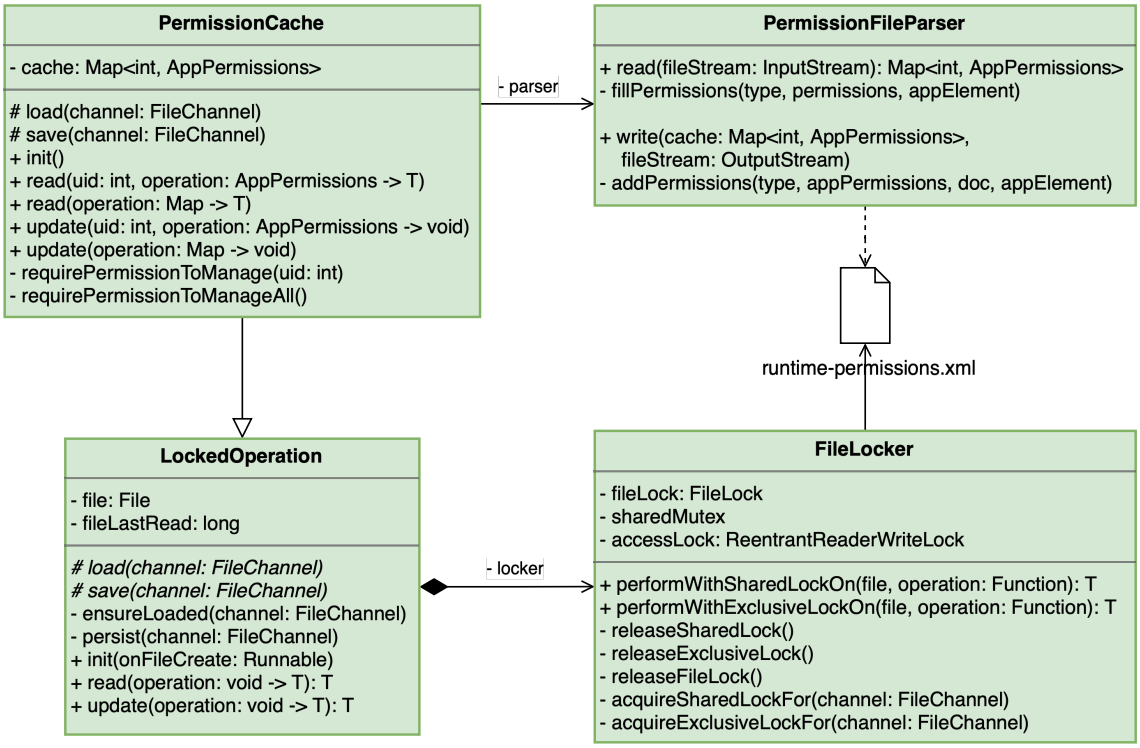

Figure 12: State persistence component class diagram.

Concurrent Access Management. The two classes implementing concurrent access to the permission file are layered, to address the task from different abstraction levels.

Following is a description of the practical solutions used by the classes to ensure atomic file operations:

- `FileLocker`: the class supports two different types of locking mechanisms:

  1. Shared locks: these allow concurrent read operations by multiple threads or processes.
  2. Exclusive locks: these prevent any other access (read or write) to the file while the lock is held, ensuring safe write operations.

  It provides a generic interface to perform locked operations on a file, while holding one of the two possible locks. These operations are passed to one of the two public methods defined in the class: `performWithSharedLockOn(File file, Function operation)` and `performWithExclusiveLockOn(File file, Function operation)`.

```
public <T> T performWithSharedLockOn(final File file,
        final Function<FileChannel, T> operation) {

    FileChannel channel = null;
    try {
        channel = new FileInputStream(file).getChannel();
        acquireSharedLockFor(channel);
        return operation.apply(channel);
    } catch (FileNotFoundException e) {
        VLog.e(TAG, "Cannot open file %s: file not found", file.toString());
        e.printStackTrace();
    } catch (IOException | OverlappingFileLockException e) {
        VLog.e(TAG, "Cannot acquire shared lock on file %s", file.toString());
        e.printStackTrace();
    } finally {
        releaseSharedLock();
        if (channel != null) {
            try {
                channel.close();
            } catch (IOException ignored) {
            }
        }
    }
    return null;
}
```

Listing 18: `performWithSharedLockOn` implementation.

Internally, the private methods in `FileLocker` handle the specific steps required to safely manage file locks. These encapsulate the low-level logic to support the functionality provided by the public methods. Here is a breakdown of the general sequence of events, like the one seen in Listing 18:

1. A lock is acquired for the file channel, using either `acquireSharedLockFor(FileChannel channel)` or `acquireExclusiveLockFor(FileChannel channel)`.
2. Once the lock is acquired, the method executes the provided operation. This ensures that file operations happen safely, without interference from other threads or processes.
3. To avoid resource contention or deadlocks, the private methods explicitly release the acquired lock as soon as the operation is completed. This is handled in a `finally` block to guarantee the lock is released even if the operation throws an exception.
4. All the steps above are wrapped inside robust error handling blocks, to manage cases where lock acquisition fails, such as when the file is inaccessible or already locked.

- `LockedOperation`: it abstracts the locking mechanisms provided by `FileLocker` into a higher-level framework. Its implementation focuses on managing the workflow of loading and saving the state. Here is how it achieves this through its methods:
  - `init(Runnable onFileCreate)`: ensures the file exists and sets up the necessary preconditions for locked operations. If the file is missing, it invokes the provided `onFileCreate`

callback to initialize an empty state. This method must be called manually during the system's initialization phase to set up the file for subsequent operations.

```
public final synchronized void init(final Runnable onFileCreate) {
    if (file.exists() && file.isFile()) {
        // Load the file contents for the first time
        locker.performWithSharedLockOn(file, channel -> {
            ensureLoaded(channel);
            return null;
        });
    } else {
        // Open the file for writing to create it
        locker.performWithExclusiveLockOn(file, channel -> {
            // Initialize data
            onFileCreate.run();
            // Persist data initialized in `onFileCreate`
            persist(channel);
            return null;
        });
    }
}
```

Listing 19: `init` method implementation.

- `read(Supplier operation)`: acquires a shared lock via `FileLocker`, enabling safe read access to the file without blocking other readers, then executes the given operation within the lock's scope, ensuring consistent state retrieval.

```
public final <T> T read(final Supplier<T> operation) {
    return locker.performWithSharedLockOn(file, channel -> {
        ensureLoaded(channel);
        return operation.get();
    });
}
```

Listing 20: `read` method implementation.

- `update(Supplier operation)`: uses an exclusive lock to guarantee write safety. It applies the provided operation while ensuring no other process or thread can read or write to the file during the update.
- `load()` and `save()`: they handle the detailed file interactions during read and update operations. These methods are implemented by subclasses and tailored to their specific state persistence needs.
- `ensureLoaded()`: verifies whether the file's last modification timestamp has changed since the previous read, before loading the state. If the timestamp—stored in `fileLastRead`—indicates that the file has been updated externally, it invokes `load()` to refresh the in-memory state. Otherwise, no action is taken, avoiding unnecessary file accesses.

```
private synchronized void ensureLoaded(final FileChannel channel) {
    if (file.lastModified() > fileLastRead) {
        fileLastRead = System.currentTimeMillis();
        load(channel);
    }
}
```

Listing 21: `ensureLoaded` implementation.

- `persist()`: ensures the in-memory state is saved back to the file system by invoking `save()`. Once the operation completes successfully, it updates `fileLastRead`, to ensure that subsequent checks correctly reflect the file's current state.

```
private synchronized void persist(final FileChannel channel) {
    save(channel);
    fileLastRead = System.currentTimeMillis();
}
```

Listing 22: `persist` implementation.

File Parsing. The `PermissionFileParser` is responsible for reading and writing permission data to and from an XML-based structure, using streams that reference the permission file. These are obtained by using `LockedOperation`, delegating the complex task of physical access to other classes in the component. This allows the class to focus purely on processing the data, without having to handle files directly.

The parser relies on standard XML parsing and transformation libraries to handle structured permission data efficiently The high-level methods, `read()` and `write()`, handle the setup for parsing and serialization tasks, but delegate most of the actual work to dedicated helper methods.

Here follows a description of the defined operations:

- XML parsing (`read()`): the method initializes the XML structure by reading from an input stream, identifying `<app>` elements for each UID, and invoking the `fillPermissions()` helper to populate the in-memory representation with permission objects for different types (install, runtime, and groups). This design ensures a modular handling of XML parsing logic.
- XML Serialization (`write()`): this method creates an XML document by iterating over UIDs and their permissions, appending them as `<app>` elements. It uses the `addPermissions()` helper method to create and structure XML nodes for each permission type. The serialized output is then transformed and written to an output stream.

Listing 23 provides an example of a permissions file, showing its XML structure and the type of data it contains.

```xml
<!-- /data/user/0/io.va.exposed64/virtual/data/app/system/permissions.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<permissions>
    <app uid="10003" >
        <install-permissions>
            <permission name="android.permission.INTERNET" status="GRANTED" />
        </install-permissions>
        <runtime-permissions>
            <permission name="android.permission.READ_CONTACTS"
                group="android.permission-group.CONTACTS"
                status="UNREQUESTED|deniedOnce|override" />
            <permission name="android.permission.CALL_PHONE"
                group="android.permission-group.PHONE"
                status="ALWAYS_ASK|grantedOnce" />
            <permission name="android.permission.WRITE_CONTACTS"
                group="android.permission-group.CONTACTS"
                status="GRANTED" />
            <permission name="android.permission.CAMERA"
                group="android.permission-group.CAMERA"
                status="GRANTED" />
            <permission name="android.permission.RECORD_AUDIO"
                group="android.permission-group.MICROPHONE"
                status="UNREQUESTED" />
        </runtime-permissions>
        <permission-groups>
            <permission name="android.permission-group.CONTACTS" status="GRANTED" />
            <permission name="android.permission-group.PHONE" status="ALWAYS_ASK" />
            <permission name="android.permission-group.MICROPHONE" status="UNREQUESTED" />
            <permission name="android.permission-group.CAMERA" status="GRANTED" />
        </permission-groups>
    </app>
    <!-- Other apps... -->
</permissions>
```

Listing 23: Permissions file contents.

`PermissionCache`. The class extends `LockedOperation`, inheriting its locking mechanisms for safe file access, and integrates the `PermissionFileParser` to handle XML parsing and serialization.

It stores the entire permission cache of the virtual model in its `cache` private field, which is a mapping of `AppPermissions` objects for each UID installed in the virtual system.

Following is a breakdown of the class methods:

- `init()`: if the permission file does not exist, it will create an empty permissions cache. It ensures that the cache is ready before any operations are performed on it.
- `read()`: provides a way to safely retrieve permissions for either a specific UID or the entire permission cache, delegating safe file access management to `LockedOperation`'s `read()` method. The method ensures that the current process has permission to access the given UID's permissions, using `requirePermissionToManage(int uid)`, or `requirePermissionToManageAll()` if the process is trying to access the entire permission cache.

```
public final <T> T read(final int uid, final Function<AppPermissions, T> operation) {
    requirePermissionToManage(uid);
    return read(() -> {
        final var permissions = cache.get(uid);
        return operation.apply(permissions != null
                ? permissions : new AppPermissions());
    });
}
```

Listing 24: Implementation of the method for accessing a specific UID's permissions.

```
public final <T> T read(final Function<Map<Integer, AppPermissions>, T> operation) {
    requirePermissionToManageAll();
    return read(() -> {
        return operation.apply(cache);
    });
}
```

Listing 25: Implementation of the method for accessing the entire permission cache.

- `update()`: allows modifications to the permissions of either a specific UID or the entire cache. Similarly to the `read()` method, it ensures that the calling process is allowed to modify the data. It acquires an exclusive lock to ensure safe write operations by using `LockedOperation`'s `update()` method, preventing other processes from accessing or modifying the data during updates.
- `requirePermissionToManage(int uid)` and `requirePermissionToManageAll()`: they enforce security restrictions based on the calling UID.
  - `requirePermissionToManage()` ensures that normal app processes can only access or modify permissions for their own UID.

```
private void requirePermissionToManage(final int uid) {
    final int callingUid = VBinder.getCallingUid();
    if (CORE.isVAppProcess() && callingUid != uid) {
        throw new SecurityException(String.format(
                    "User %d is trying to access user %d permissions",
                    callingUid, uid));
    }
}
```

Listing 26: Implementation of the restriction on other users' permissions.

  - `requirePermissionToManageAll()` restricts access to all permission data for processes that belong to virtual apps, ensuring that only the virtualization framework's process can perform such operations.

```
private void requirePermissionToManageAll() {
    // Trigger only if current process is from a virtual app and it has been started
    if (CORE.isVAppProcess() && CORE.isStartup()) {
        throw new SecurityException(String.format(
                    "User %d is trying to access all users permissions",
                    VBinder.getCallingUid()));
    }
}
```

Listing 27: Implementation of the restriction on all users' permissions.

- `load()` and `save()`: they implement the abstract method defined in `LockedOperation` by parsing and serializing the permission cache, using the `PermissionFileParser` instance present in the class. `LockedOperation` internal implementation of other methods will use these to keep the permissions cache synchronized with the persistence layer.

```
@Override
protected void load(final FileChannel channel) {
    try {
        VLog.i(TAG, "Loading permission cache from file");
        cache = parser.read(Channels.newInputStream(channel));
    } catch (Exception e) {
        VLog.e(TAG, "Could not read permissions file");
        e.printStackTrace();
    }
}
```

Listing 28: `load` method implementation, using the parser.

5.3.3 Management Core Component

Design. The *management core* provides essential operations for handling permission states. It is the interface replacing operations defined in Android's management component, emulating their implementation and adapting it to the virtual environment. Additionally, it includes a small compatibility layer within the virtualization framework's native library. This layer provides native code with access to the permission management core's features implemented in the main framework, allowing patches applied to system methods to redirect native-level permission checks to the virtual model.

Since its interactions with virtual apps are designed to mimic the system's original model, the interface maintains a certain degree of similarity with the original component, to simplify the redirection process.

The main responsibilities of this components are:

- Querying the current state of permissions.
- Performing operations such as granting, revoking, and supporting UI-related actions.
- Managing and maintaining the overall permission state model.
- Handling exceptions or special cases for specific permissions.

Implementation. The component is implemented as a single service in the virtualization framework, inspired by Android's `PermissionManager`. Like all other services in VirtualApp, `VPermissionManager` is implemented following the singleton pattern, allowing it to be easily accessed and referenced across the codebase, similar to what the Android service mechanism itself provide.

Additionally, the component includes a native compatibility layer, which acts as a wrapper around the Java service. This layer simplifies access to `VPermissionManager` from JNI, allowing native code to conveniently interact with permissions, without needing to handle Java service calls directly.

Below is a breakdown of the component architecture, illustrated in Figure 13.

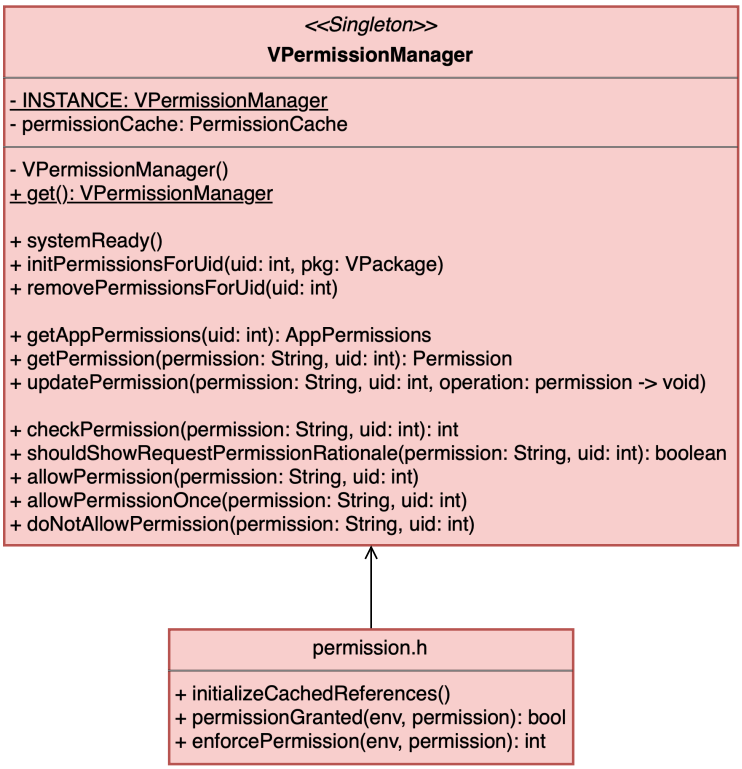

Figure 13: Management core component class diagram.

Singleton Implementation. The singleton pattern is implemented in the class with:

- `INSTANCE`: the instance object, stored as a private, static, and final field, ensuring that the class is actually instantiated once. Because it is static it is also automatically instantiated by the Java runtime.
- Private constructor: prevents instantiation of the class from outside.
- `get()`: static method to get the singleton instance.

System Configuration Methods. These methods are responsible for setting up and managing system configurations related to permissions:

- `systemReady()`: initializes the permission system when the virtual app starts by loading the permission state and revoking permissions' "granted once" status.

```
public synchronized void systemReady() {
    // Initialize the permission cache
    permissionCache.init();

    // Reset granted once status from all runtime permissions
    permissionCache.update(permissions -> {
        // Iterate on all virtual apps' permissions
        permissions.values().forEach(appPermissions -> {
            appPermissions.getAll(RuntimePermission.class).forEach((name, permission) -> {
                if (permission.isGrantedOnce()) {
                    // Reset status to revert granted once
                    permission.setStatus(Status.ALWAYS_ASK);
                }
            });
        });
    });
}
```

Listing 29: `systemReady` method, initializing `permissionCache`'s state.

- `initPermissionsForUid(int uid, VPackage pkg)`: initializes permissions when a new app is installed, creating permissions associations based on provided package informations.
- `removePermissionsForUid(int uid)`: removes permissions associated with an app when it is uninstalled.

General-Purpose Permission Management. These methods provide access to permission data and manage permission states:

- `AppPermissions getAppPermissions(int uid)`: returns permissions information for a specific UID.

```
public AppPermissions getAppPermissions(final int uid) {
    return permissionCache.read(uid, Function.identity());
}
```

Listing 30: `getAppPermissions` method, reading `permissionCache`'s state.

- `Permission getPermission(String permissionName, int uid)`: returns a specific permission or permission group by name for a given UID.
- `updatePermission(String permissionName, int uid, Function operation)`: allows updating a permission's status with the generic `operation` callback. It gives the callback access to the permission in its current state and automatically persists any changes.

Specific Permission Logic Methods. These methods implement the core logic exposed to the redirection and user interaction components, for handling specific permission checks and operations:

- `int checkPermission(String permission, int uid)`: mirrors Android's permission checking mechanism, returning `PERMISSION_GRANTED` or `PERMISSION_DENIED`.
- `boolean shouldShowRequestPermissionRationale(String permission, int uid)`: also mirrors its Android API counterpart, determining whether a rationale should be displayed for explaining a specific permission request.
- `allowPermission(String permissionName, int uid)`: implements the "Allow" button behavior in the permission dialog. It grants the requested permission or permission group.
- `allowPermissionOnce(String permissionName, int uid)`: it is related to a situation where the user chooses to grant a permission temporarily, typically associated with the "Allow once" button in the UI. It grants the permission for the current session only, meaning that it is only effective until the app is closed or the session ends.
- `doNotAllowPermission(String permissionName, int uid)`: reflects the user's choice not to allow a permission or permission group, which corresponds to the "Don't Allow" button in the dialog. It handles the logic of denying the permission, or marking it as "denied once". It also handles the denial of permissions within groups, ensuring that when a permission group is denied, the individual permissions in it are properly denied as well.

```
public void doNotAllowPermission(final String permissionName, final int uid) {
  updatePermission(permissionName, uid, permission -> {
    if (permission instanceof InstallPermission) {
      // Simply deny install permissions
      permission.setStatus(Status.DENIED);
    } else if (permission instanceof PermissionGroup permissionGroup) {
      // Set as unrequested, the individual runtime permissions will reflect the fact
      // that the group has been denied.
      // This allows to permissions to display the dialog, which is lines up with Android
      permissionGroup.setStatus(Status.UNREQUESTED);
      permissionGroup.getPermissions().stream()
      .filter(runtimePermission -> !runtimePermission.isOverridden())
      .forEach(runtimePermission -> {
        // Set all (non-overridden) runtime permissions in the group as denied once
        runtimePermission.setDeniedOnce();
      });
    } else if (permission instanceof RuntimePermission runtimePermission) {
      // Check if permission had been denied once
      if (runtimePermission.shouldShowRequestPermissionRationale()) {
        // RuntimePermission's setStatus will automatically set the group status
        runtimePermission.setStatus(Status.DENIED);
      } else {
        runtimePermission.setDeniedOnce();
      }
    }
  });
}
```

Listing 31: `doNotAllowPermission` method, showing a usage of `updatePermission`.

Native Compatibility Layer. It provides the following utility functions in the `permission.h` header:

- `initializeCachedReferences()`: sets up static references to Java classes and methods for permission checks. This function is called in the framework native library initialization process.
- `bool permissionGranted(JNIEnv*, const char*)`: verifies if the calling process has the required permission.
- `int enforcePermission(JNIEnv*, const char*)`: enforces a permission check, returning `android::OK` if granted, or logging an error and returning `android::PERMISSION_DENIED` if denied.

```
int enforcePermission(JNIEnv* env, const char* permission) {
  if (permissionGranted(env, permission)) {
    return android::OK;
  }
  jint pid = env->CallStaticIntMethod(binderClass, getCallingPidMethod);
  jint uid = env->CallStaticIntMethod(binderClass, getCallingUidMethod);
  // Similar to the message in CameraService.cpp
  log_err("Permission Denial: %s pid=%d, uid=%d", permission, pid, uid);
  return android::PERMISSION_DENIED;
}
```

Listing 32: Native `enforcePermission` function implementation.

5.3.4 User Interaction Component

Design. This component defines how the user interacts with the permission model. It needs to closely mirror the experience provided by Android to leverage users' familiarity with the system. While aesthetic details can differ, the observable behavior—such as transitions between permission states—must align exactly with Android's implementation to maintain consistency and predictability.

The main use case of this component is to display and manage UI elements, particularly the permission dialogs that appear when a plugin app requests permissions. It is also responsible for handling permission requests and providing the container application with settings activities to let users manage their permission preferences.

Additionally, this component addresses a crucial consideration: when granting virtual permissions, it checks whether the container app itself holds the corresponding permission on the actual system. Granting a virtual permission without this verification would be both meaningless and inconsistent, as the container app would not be able to grant the virtual apps access to the associated functionality.

Implementation. The component is implemented in two distinct ways, reflecting the separation between the virtualization framework and the container app. Each approach addresses different aspects of user interaction with the permission system:

- In the virtualization framework, user interaction is handled through dialogs that are displayed when permission requests occur.
- In the container app, user interaction focuses on managing permission preferences.

Because these interactions are distinct and implemented in physically separate parts of the codebase, the following sections describe each approach in separately.

Permission Requests. As shown in Figure 14, there are two activities managing permission requests:

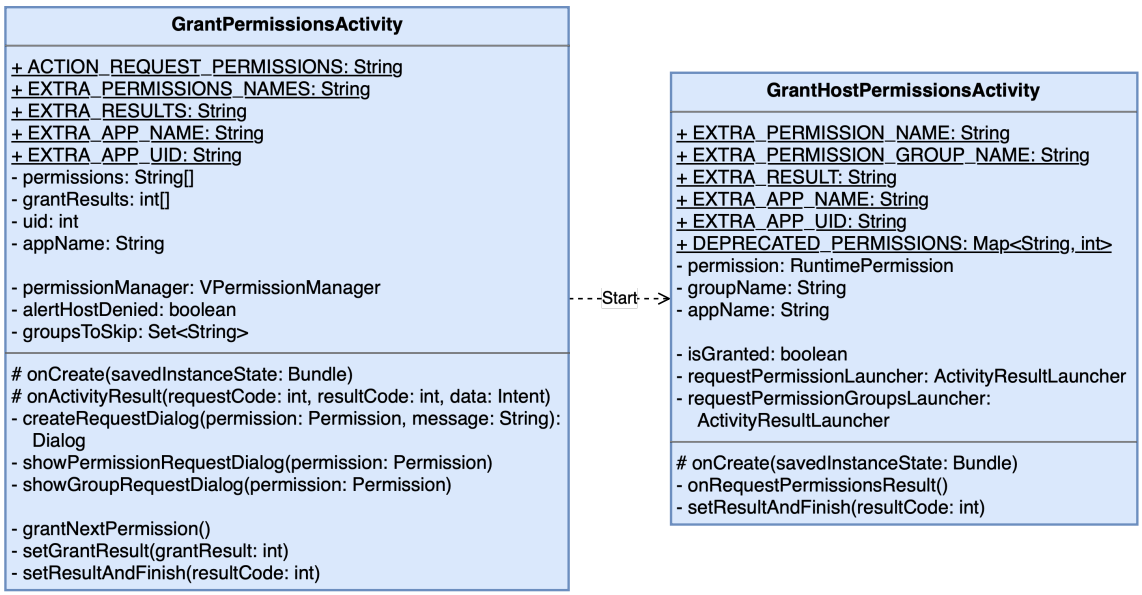

Figure 14: Permission requests activities class diagram.

1. `GrantPermissionsActivity` is the entry point for handling permission requests, coming from the redirection component. It mirrors the `GrantPermissionsActivity` in Android's `PermissionController` component, adapting it for the virtualization framework.

   Its core methods are:

   - `onCreate(Bundle savedInstanceState)`: initializes the activity by parsing permission request data from the intent and invokes `grantNextPermission()` to begin processing requests.
   - `grantNextPermission()`: processes the current permission request based on its status and displays the appropriate dialog (group or individual permission). It also checks whether the host has the required permission. If not, it launches the `GrantHostPermissionActivity` to alert the user and request it.
   - `grantPermission()`: perform the actual decision of either showing the dialog or directly returning a result, based on whether the permission is already granted or denied.

```
private void grantPermission(final String permissionName) {
    final var permission = permissionManager.getPermission(permissionName, uid,
            RuntimePermission.class);
    final var group = permission.getPermissionGroup();
    if (!permission.needsRequestDialog()) {
        switch (permission.getStatus()) {
            case Status.ALWAYS_ASK, Status.UNREQUESTED -> {
                if (group != null && group.isGranted()) {
                    permissionManager.allowPermission(permissionName, uid);
                }
            }
        };
        setGrantResult(permission.isGranted()
            ? PERMISSION_GRANTED
            : PERMISSION_DENIED);
        return;
    }
    if (permission.isOverridden()) {
        // If not linked to group, show request dialog for the individual permission
        showPermissionRequestDialog(permission);
        return;
    }
    if (group != null && group.isGranted()) {
        // Group is already granted, grant permission too
        permissionManager.allowPermission(permissionName, uid);
        return;
    }
    // Show dialog for the permission group
    showGroupRequestDialog(permission);
}
```

Listing 33: `grantPermission` method.

- `onActivityResult(int requestCode, int resultCode, Intent data)`: handles results from host permission requests. If denied, it skips further requests for the same group, otherwise, it proceeds with the request for the virtual app.

Some methods are dedicated to the dialog handling:

- `showPermissionRequestDialog(RuntimePermission permission)`: displays a dialog for a permission not assigned to a group or that is overridden, thus it has to be managed individually.
- `showGroupRequestDialog(RuntimePermission permission)`: displays a dialog for `permission`'s permission group.
- `Dialog createRequestDialog(RuntimePermission permission, String message)`: creates a dialog showing the right options for `permission`. The dialog may not always be perfectly matching to the system's verison, due to higher complexity for specific permissions, such as location, media access, or background permissions.

```
private Dialog createRequestDialog(RuntimePermission permission, String message) {
    final var permissionName = permission.getName();
    final var dialog = new Dialog(this);

    // ...

    icon.setImageResource(permission.getGroupIconRes());
    messageView.setText(Html.fromHtml(message));
    allowButton.setOnClickListener(view -> {
        permissionManager.allowPermission(permissionName, uid);
        dismissWith(PERMISSION_GRANTED, dialog);
    });
    onlyOnceButton.setOnClickListener(view -> {
        permissionManager.allowPermissionOnce(permissionName, uid);
        dismissWith(PERMISSION_GRANTED, dialog);
    });
    doNotAllowButton.setOnClickListener(view -> {
        permissionManager.doNotAllowPermission(permissionName, uid);
        if (group != null) {
            // Don't keep requesting permissions for the same group
            groupsToSkip.add(group.getName());
        }
        dismissWith(PERMISSION_DENIED, dialog);
    });
    dialog.setOnCancelListener(view -> {
        setResultAndFinish(RESULT_CANCELED);
    });
    if (!hasBackgroundPermission) {
        onlyOnceButton.setVisibility(TextView.GONE);
    }

    // ...

    return dialog;
}
```

Listing 34: `createRequestDialog` method, creating and customizing the dialog.

- `setGrantResult(int grantResult)`: updates the `grantResults` array and advances the request loop.
- `setResultAndFinish(int resultCode)`: finalizes the activity with the results.

2. `GrantHostPermissionActivity` manages host-level permission requests when the virtualization framework identifies that the host lacks necessary permissions.

   Key methods include:

   - `onCreate(Bundle savedInstanceState)`: initializes the activity by retrieving permission details from the intent and displays an alert dialog explaining the request.
   - `onRequestPermissionsResult()`: handles the result of the user's decision. If the permission is granted, it automatically requests all permissions in the group to prevent any future warning. If it seems denied permanently, it presents another dialog linking to the app settings.
   - `setResultAndFinish(int resultCode)`: finishes the activity with the result.

Deprecated permissions, identified through `DEPRECATED_PERMISSIONS`, are skipped automatically without user interaction, based on the current Android API level. This ensures obsolete requests do not disrupt the workflow.

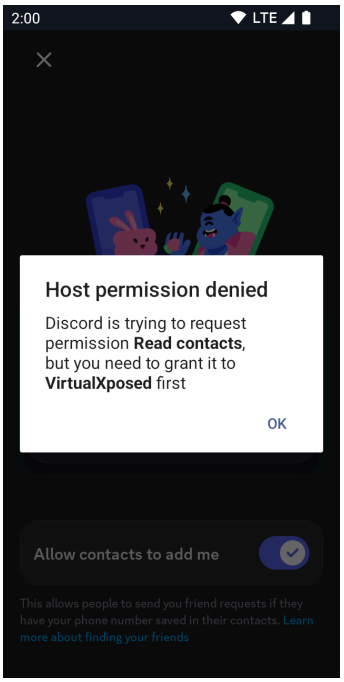

Figure 15: Missing host permission alert.

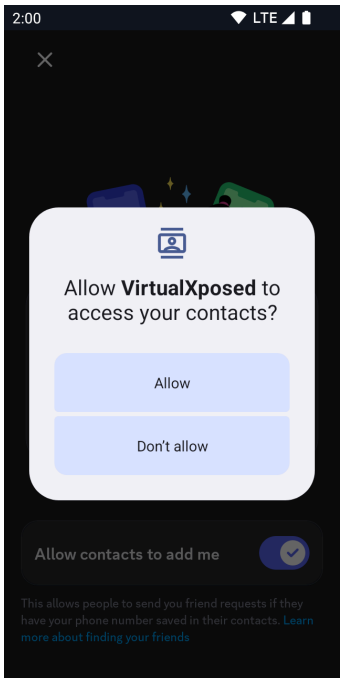

Figure 16: Host permission dialog, notice VirtualXposed's name

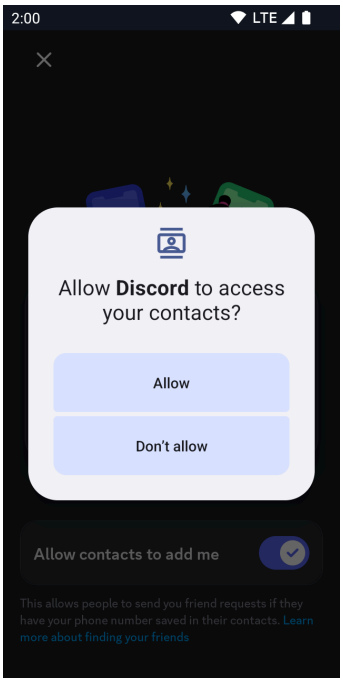

Figure 17: Virtual permission dialog.

Permission Preferences. Permission preferences are also managed by two different activities, represented in Figure 18:

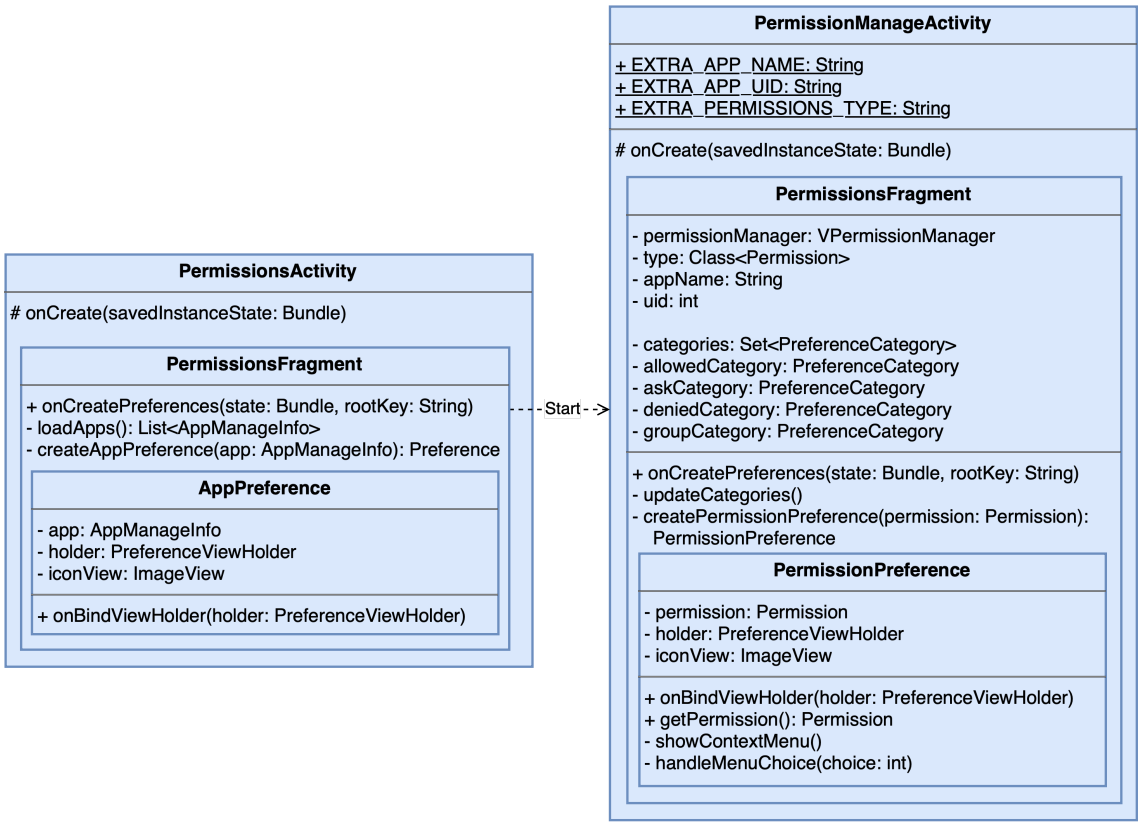

Figure 18: Permission preferences activities class diagram.

1. `PermissionsActivity` is the entry point for managing app permissions via a settings-like interface. It includes an inner `PermissionsFragment` that displays and manages the UI components for permission preferences.

   After setting being set as the main content of the activity, `PermissionsFragment` calls its `onCreatePreferences()` method. This method loads the top-level preferences from the layout and initializes the fragment's context and UI elements. The layout consists of:

   - A switch to toggle alerts for denied host permissions. This switch is tied to a key stored in shared preferences and is dynamically updated based on the current settings.
   - A list of all installed virtual apps preferences, redirecting to their specific `PermissionManageActivity`.

   The applications list is then created with `loadApps()`. This list is used to dynamically create the list of `AppPreference` objects for each app with `createAppPreference()`, associating it with the relevant app-specific data and click behavior.

```
private Preference createAppPreference(final AppManageInfo app) {
    final CharSequence appName = app.getName();
    final AppPreference appPreference = new AppPreference(ctx, app);
    final int uid = VPackageManager.get().getPackageUid(app.pkgName, app.userId);

    appPreference.setKey("app_permission_preference_" + appName);
    appPreference.setTitle(appName);
    // The actual icon will be set dynamically
    appPreference.setIcon(DEFAULT_APP_ICON);
    appPreference.setLayoutResource(R.layout.item_app_permission);
    appPreference.setOnPreferenceClickListener(preference -> {
        final Intent intent = new Intent(ctx, PermissionManageActivity.class);
        intent.putExtra(EXTRA_APP_NAME, appName);
        intent.putExtra(EXTRA_APP_UID, uid);
        intent.putExtra(EXTRA_PERMISSIONS_TYPE, PermissionGroup.class);
        startActivity(intent);
        return true;
    });

    return appPreference;
}
```

Listing 35: `createAppPreference` method, creating a button to launch the relative `PermissionManageActivity`.

2. `PermissionManageActivity` is designed to provide detailed control over app permissions, allowing users to manage permissions for a specific app. The activity is initialized with intent extras that pass the app name, UID, and permission type. Based on `type`, the activity shows different layouts, to reflect the differences between the states that install-time permissions, runtime permissions, and permission groups can assume. Initially, `PermissionsActivity` starts the activity with type "group", which shows a preference in its layout that allows the overriding of runtime permissions. This is done in this same activity, by passing `RuntimePermission` as type. It could also support install-time permissions out of the box, but this deviates from Android's model design.

   Just like `PermissionsActivity`, the activity replaces its content with a `PermissionsFragment` in its `onCreate()` method. The fragment categorizes permissions into four groups, based on permissions' state and type:

   - Allowed: granted to the app.
   - Ask: runtime permission or permission group needing a request.
   - Denied: permanently denied.
   - Follow group: runtime permission that is set to follow its group's status.

   In its `onCreatePreferences()` method, the fragment creates a `PermissionPreference` for each permission of the specific type with `createPermissionPreference()`.

   `PermissionPreference` includes methods to modify permissions' status:

- `showContextMenu()`: when clicked, the preference shows a context menu with options to update the permission's status.
- `handleMenuChoice(int choice)`: uses `permissionManager` to perform the permission update in the model. Then it calls `updateCategories()`, to move the preference accordingly in the UI.

```
private void handleMenuChoice(final int choice) {
    if (choice == R.id.action_group) {
        permissionManager.updatePermission(RuntimePermission.class,
            permission.getName(), uid,
            newPermission -> {
                final var group = newPermission.getPermissionGroup();
                newPermission.followGroup();
                newPermission.setStatus(group.getStatus());
            });
        updateCategories();
        return;
    }

    if (permission instanceof RuntimePermission runtimePermission) {
        runtimePermission.override();
    }
    if (choice == R.id.action_allow) {
        permissionManager.allowPermission(permission.getName(), uid);
    } else if (choice == R.id.action_ask) {
        permissionManager.updatePermission(permission.getName(), uid,
            newPermission -> {
                newPermission.setStatus(ALWAYS_ASK);
            });
    } else if (choice == R.id.action_do_not_allow) {
        permissionManager.doNotAllowPermission(permission.getName(), uid);
    }
    updateCategories();
}
```

Listing 36: `handleMenuChoice` updating the permission state and refreshing the UI.

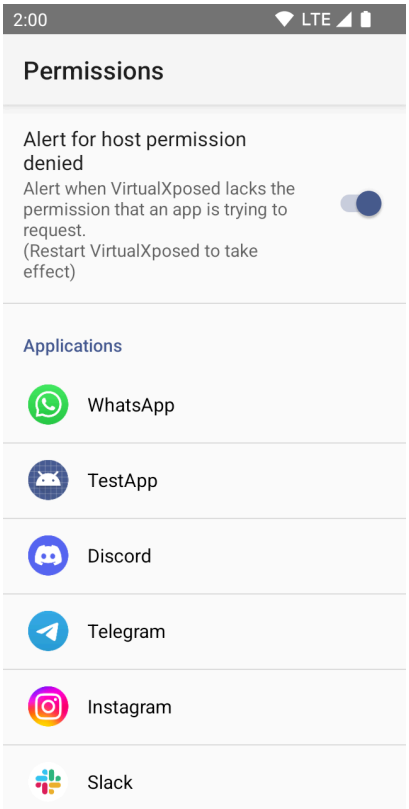

Figure 19: Apps list in permissions activity.

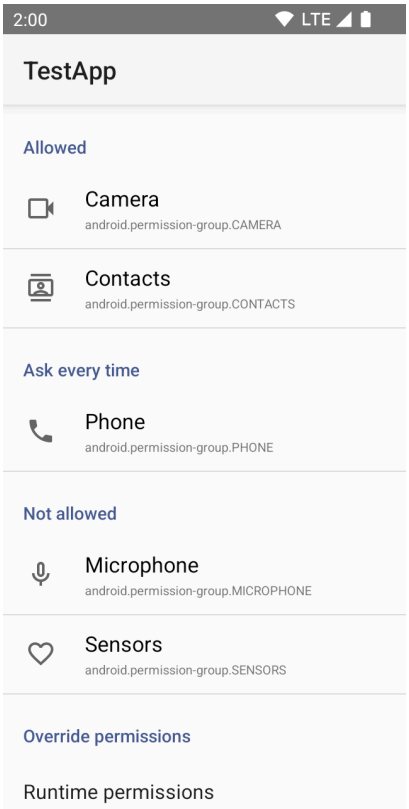

Figure 20: *TestApp* permission manage activity.

5.3.5 Redirection Component

Design. The redirection component is necessary to allow communication between virtual apps and the virtual permission model. Virtual apps may either directly invoke Android's permission management APIs or execute operations that inherently trigger permission checks within their code. This component addresses both scenarios, by implementing dynamic proxies and native method patches to create hooks. These hooks intercept permission-related calls and redirect them to the virtual permission management core, creating the illusion that permission requests are being processed directly by the system, when in reality, they are being managed by the virtual model, inside the virtualization framework.

Before redirecting any request to the virtual model, this component first checks the container app's permissions to ensure that the virtual app's request can be properly granted. This step ensures that the virtual model is only used when appropriate, maintaining the integrity of the permission system and preventing the redirection of requests when the container app itself lacks the necessary permissions.

Implementation. The main focus is on redirecting core methods of the Android API, particularly:

- `checkPermission`: as "the only public entry point for permissions checking" [24], it is a fundamental element of the system. By redirecting this method, all permission checks made directly from virtual apps will target the virtual permission model.
- `requestPermissions`: handles permission requests, sending them to the `PermissionController` using an Intent. By hooking into the `startActivity` method, it is possible to redirect permission requests to the virtual model.

Both `checkePermission` and `requestPermission` are managed by the `ActivityManagerService`, allowing the use of dynamic proxies to handle their redirection. The framework already defines proxies for these methods, so the implementation mostly consists in integrating virtual permission logic into the existing system.

In addition to redirecting core methods, specific features are addressed to extend the support to practical use cases:

- Permission checks are applied before executing actions on content providers, with all operations patched to enforce the required read and write permissions.
- A native patch is implemented on the `native_setup` method of the `Camera` class to verify virtual permissions before granting access. This extension ensures that components relying on native code can also be subject to the virtual permission model.

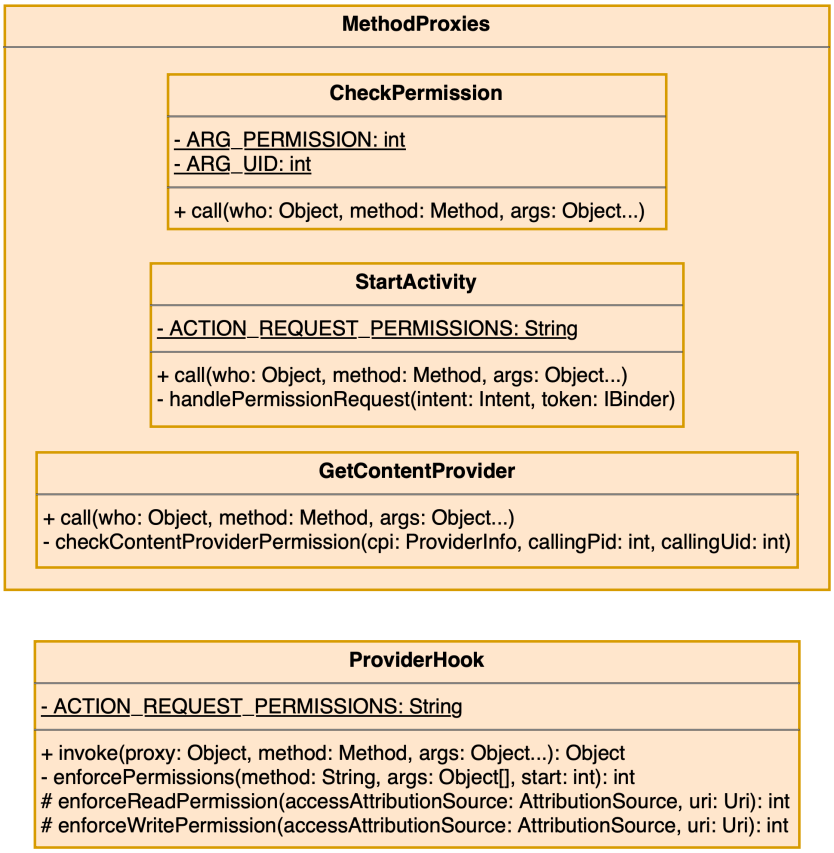

Figure 21: Redirection component class diagram.

Method Proxies. Methods supporting dynamic proxy hooking are redirected by defining inner classes inheriting from `MethodProxy` in the `MethodProxies` class. These proxies are eventually injected into the `ActivityManagerStub`, replacing plugin apps' calls to their system counterparts with custom implementations of the `call()` method.

The implemented proxies are the following:

- `CheckPermission`: it first validates host-level permission and then checks the corresponding virtual permission, delegating model access and logic to the management core.

```
@Override
public Object call(Object who, Method method, Object... args) throws Throwable {
    final String permission = (String) args[ARG_PERMISSION];
    final int vuid = getVUid();
    if (vuid != getBaseVUid()) {
        // android.os.UserHandle.getAppId(uid) returned the base app id,
        // although here it's important to preserve userIds.
        // Restore the VUid for the correct userId, otherwise it would return a
        // SecurityException for accessing the baseVUid permissions
        args[ARG_UID] = vuid;
    }
    final int uid = (int) args[ARG_UID];

    // Check host permission first
    args[ARG_UID] = getRealUid();
    final int hostStatus = (int) method.invoke(who, args);
    if (hostStatus == PackageManager.PERMISSION_DENIED) {
        VLog.e(TAG, "Checking permission '%s' for uid=%d: permission is denied for host",
                permission, uid);
        return PackageManager.PERMISSION_DENIED;
    }

    final int status = VPermissionManager.get().checkPermission(permission, uid);
    VLog.d(TAG, "Checking permission '%s' for uid=%d: permission %s", permission, uid,
            status == PackageManager.PERMISSION_GRANTED
                ? "GRANTED"
                : "DENIED");
    return status;
}
```

Listing 37: `checkPermission` method proxy implementation.

- `StartActivity`: it identifies actions matching ACTION_REQUEST_PERMISSION and redirects permission requests to the user interaction component. It ensures that all permission requests from virtual apps are intercepted, whether started via API calls or intents.

```
private void handlePermissionRequest(final Intent intent, final IBinder token) {
    final Context appContext = VActivityManager.get().getActivityRecord(token).activity;
    final ApplicationInfo applicationInfo = appContext.getApplicationInfo();
    final int stringId = applicationInfo.labelRes;
    final String appName = stringId == 0
        ? applicationInfo.nonLocalizedLabel.toString()
        : appContext.getString(stringId);

    intent.putExtra(GrantPermissionsActivity.EXTRA_APP_NAME, appName);
    intent.putExtra(GrantPermissionsActivity.EXTRA_APP_UID, getVUid());
    intent.setComponent(new ComponentName(getHostContext(),
                GrantPermissionsActivity.class));
}
```

Listing 38: Code redirecting the permission request to `GrantPermissionsActivity`.

- `GetContentProvider`: it adds a generalized check on read and write permissions for getting a content provider by using the `checkContentProviderPermission` method, which mirrors the equivalent in the Android API.

```
private String checkContentProviderPermission(ProviderInfo cpi, int callingPid, int callingUid) {
    final VPermissionManager permissionManager = VPermissionManager.get();
    if (cpi.readPermission == null && cpi.writePermission == null) {
        return null;
    }
    if (permissionManager.checkPermission(cpi.readPermission, callingUid)
            == PackageManager.PERMISSION_GRANTED) {
        return null;
    }
    if (permissionManager.checkPermission(cpi.writePermission, callingUid)
            == PackageManager.PERMISSION_GRANTED) {
        return null;
    }
    return "Permission Denial: opening provider " + cpi.name
            + " from (pid=" + callingPid + ", uid=" + callingUid + ")"
            + " requires " + cpi.readPermission + " or " + cpi.writePermission;
}

@Override
public Object call(Object who, Method method, Object... args) throws Throwable {
    String name = (String) args[ARG_NAME];
    int userId = VUserHandle.myUserId();
    ProviderInfo info = VPackageManager.get().resolveContentProvider(name, 0, userId);
    if (info == null) {
        final ProviderInfo cpi = getPM().resolveContentProvider(name, PackageManager.GET_META_DATA);
        final int pid = VBinder.getCallingPid();
        final int uid = VBinder.getCallingUid();
        final String msg = checkContentProviderPermission(cpi, pid, uid);
        if (msg != null) {
            throw new SecurityException(msg);
        }
    }
    // ...
}
```

Listing 39: Permission checking flow for general content provider retrieval.

Content Providers Permission Enforcement. To maintain precise control over content provider access, individual content providers methods are patched. All operations are redirected to enforce permissions before execution, as detailed in Listing 40.

To achieve this, an `enforcePermissions()` method is invoked before performing any operation, from the `ProviderHook` dynamic proxy. Depending on whether the operation involves reading or writing restricted data, the method delegates checks to either `enforceReadPermission()` or `enforceWritePermission()`. Both methods closely mirror their Android system counterparts, generating similar error messages when access is denied.

This approach allows the system to enforce permissions consistently across all interactions with content providers, ensuring that virtual apps cannot bypass restrictions in any way.

```
private int enforcePermissions(final String method, final Object[] args, final int start)
        throws OperationApplicationException {

    if (args != null && args.length > start && args[0] != null && args[start] != null
            && args[0] instanceof android.content.AttributionSource accessAttributionSource
            && args[start] instanceof Uri uri) {
        return switch (method) {
            case "query", "canonicalize", "uncanonicalize", "refresh" ->
                enforceReadPermission(accessAttributionSource, uri);
            case "insert", "bulkInsert", "delete", "update" ->
                enforceWritePermission(accessAttributionSource, uri);
            case "applyBatch" -> {
                final var operations = (ArrayList<ContentProviderOperation>) args[start + 1];
                for (final var operation : operations) {
                    if (operation.isReadOperation()) {
                        if (enforceReadPermission(accessAttributionSource, uri)
                                != PackageManager.PERMISSION_GRANTED) {
                            throw new OperationApplicationException("App op not allowed", 0);
                        }
                    }
                    if (operation.isWriteOperation()) {
                        if (enforceWritePermission(accessAttributionSource, uri)
                                != PackageManager.PERMISSION_GRANTED) {
                            throw new OperationApplicationException("App op not allowed", 0);
                        }
                    }
                }
                yield PackageManager.PERMISSION_GRANTED;
            }
            default -> PackageManager.PERMISSION_GRANTED;
        };
    }
    return PackageManager.PERMISSION_GRANTED;
}
```

Listing 40: `enforcePermissions` implementation, for checking content providers.

Camera Patch. To enforce permissions in native-level camera operations, a patch is applied to the `native_setup()` method of the `Camera` class. This ensures that permission checks are performed before any camera access is granted, integrating seamlessly with the virtual permission model.

The patch utilizes the native management core compatibility layer, specifically the `enforcePermission()` function, to validate the Camera permission. Instead of returning the camera ID, the patched method returns an error code: `android::OK` on success or a failure code if the permission check fails. This behavior aligns with standard error-handling conventions in native Android systems.

Listing 41 showcases the implementation of the method patch.

```
static jint new_native_cameraNativeSetupFunc_T5(JNIEnv *env, jobject thiz, jobject cameraThis,
                                                jint cameraId, jstring packageName,
                                                jboolean overrideToPortrait,
                                                jboolean forceSlowJpegMode) {

    int rc = enforcePermission(env, "android.permission.CAMERA");
    if (rc != android::OK) {
      return rc;
    }
    jstring host = env->NewStringUTF(patchEnv.host_packageName);
    return patchEnv.orig_native_cameraNativeSetupFunc.t5(env, thiz, cameraThis, cameraId, host,
                                                         overrideToPortrait, forceSlowJpegMode);
}
```

Listing 41: `native_setup` native method patch.

6

Evaluation

The evaluation of the virtual permission model was performed using two distinct methods: a custom *TestApp* and *Telegram*, a widely used real-world application. This dual approach ensured a comprehensive evaluation of the model, testing its functionality in both controlled and practical contexts. Additionally, this evaluation highlighted the inherent challenges and limitations of achieving a generalized solution, capable of addressing every possible edge case in permission management.

6.1 TestApp

The *TestApp* was specifically designed to create a controlled environment for testing key aspects of the permission model. Its activities included layouts that supported the testing of specific permission-related operations. This approach allowed for an in-depth verification of system's features correct integration with virtual app's interactions, in the context of a controlled and manageable environment,

Additionally, the app provided valuable insights into Android's permission model, when installed and run outside the virtualization framework. This dual use offered a unique perspective on how the model operates in both contexts.

Comparisons of the same operations performed within and outside the virtual environment were critical for ensuring consistent behavior. Differences in the app's behavior identified during these tests allowed for the systematic identification and resolution of errors, ensuring that app functionality remained unchanged across environments.

The following subsections describe *TestApp*'s activities and the implemented operations.

6.1.1 PermissionRequestActivity

The `PermissionRequestActivity` is designed to validate the core permission mechanisms in the virtual environment, focusing on two critical operations: `checkPermission()` and `requestPermissions()`. The activity provides an interactive interface that allows users to test and visualize the permission states of individual permissions, allowing for a direct comparison between the virtual environment and the native Android system.

The operations are implemented in a user-friendly way, providing easy interaction and immediate visual feedback:

1. Permission check: it allows users to retrieve the current permission status of a predefined permissions list. Each permission is associated with a label displaying its current status, such as “Granted”, “Denied”, or “Should show rationale”. A button next to each permission triggers the `checkPermission()` method, which updates the displayed status. By checking permissions both inside and outside the virtual environment, it is possible to verify whether the system is consistent in managing permission statuses across different contexts.
2. Permission request: it tests the app’s ability to request permissions and ensures that the correct permission dialogs are displayed. Each permission in the list has a button that triggers the `requestPermissions()` method for that specific permission. Additionally, a “Request all permissions” button allows for testing multiple permission requests simultaneously. This functionality is essential for verifying that the app can handle bulk permission requests correctly and that the system responds properly to multiple requests at once, displaying the appropriate dialogs for each permission.

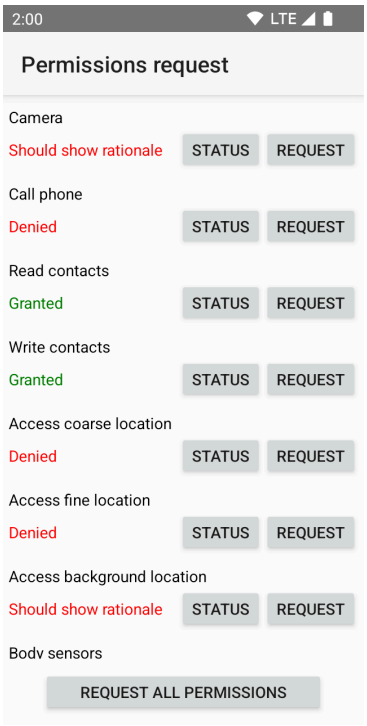

Figure 22: Permission check view inside the virtual environment.

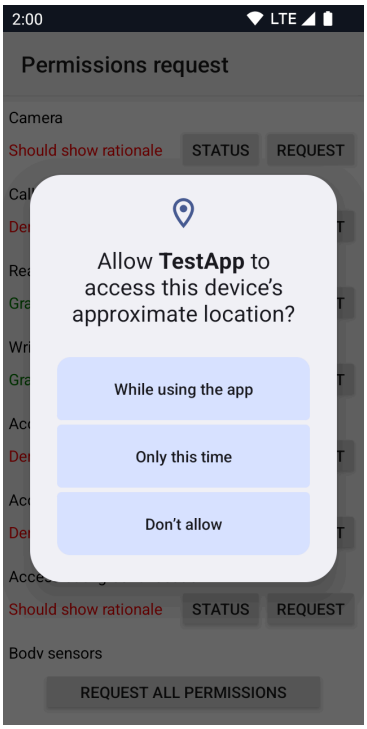

Figure 23: A virtual permission request dialog.

6.1.2 ContactsActivity

The `ContactsActivity` is designed to test the permission control mechanisms applied to content providers, specifically the *Contacts* content provider. This activity verifies the app's ability to read, modify, and manage contacts data, which is tightly controlled by permissions in Android. Each operation in this activity is crucial for ensuring that the permission model properly manages and enforces permissions for every possible content provider operation.

The operations are as follows:

- Read contact: `readContact()` reads the contacts content provider, using its `query()` method. On a successful attempt it displays the name, phone number, and email address of the first contact. It is the only operation requiring just the `READ_CONTACTS` permission.
- Add contact: `addContact()` inserts a new contact into the content provider, using its `insert()` method. It is the only operation requiring just the `WRITE_CONTACTS` permission.
- Delete contact: `deleteContact()` deletes the first contact by querying the contacts provider and calling `delete()` on the corresponding entry.
- Update contact: `updateContactName()` modifies the name of the first contact by querying the contacts provider and calling `update()` on the corresponding entry, passing the new name.
- Update batch contact information: `updateContact()` modifies the name, phone number, and email address of the first contact by querying the contacts provider and calling `applyBatch()` on the corresponding entry. This applies multiple operations at the same time, which needs to be checked individually by the model to enforce read or write permissions accordingly.

All operations include exception handling to catch and log any `SecurityException` errors. This ensures permission errors cause an explicit feedback, facilitating the identification of inconsistencies.

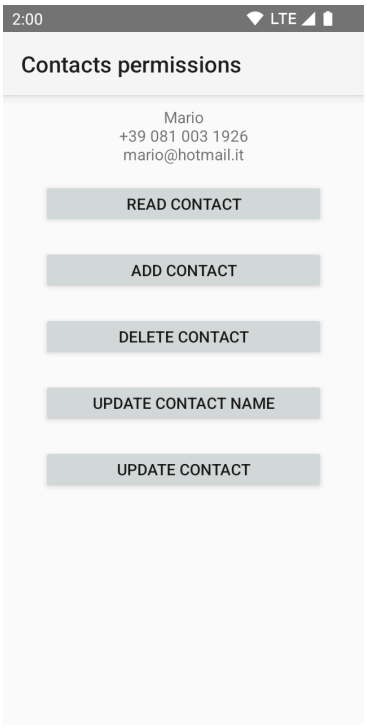

Figure 24: *TestApp* reading a contact.

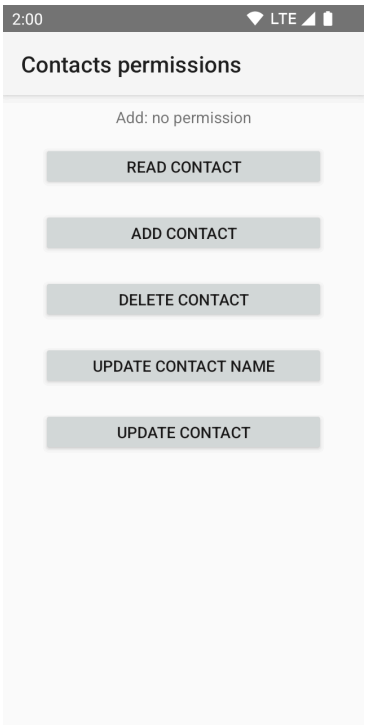

Figure 25: Operation denied for virtual *TestApp*.

6.1.3 InternetActivity

The `InternetActivity` tests the app's behavior when attempting to perform a network operation, specifically fetching data from a web page. The expected behavior in a standard Android environment is that the network request will fail and raise an exception if the app lacks the necessary Internet permission.

However, when testing this activity in the virtual environment, the request always succeeds, even though the Internet permission has not been granted. This behavior raises a significant issue, proving that redirection of core methods alone is not enough to ensure that the permission model operates as expected across all use cases.

In this instance, the test reveals that additional considerations are needed to enforce permission restrictions at the level of system operations, such as networking, which are outside the scope of simple permission checks and requests.

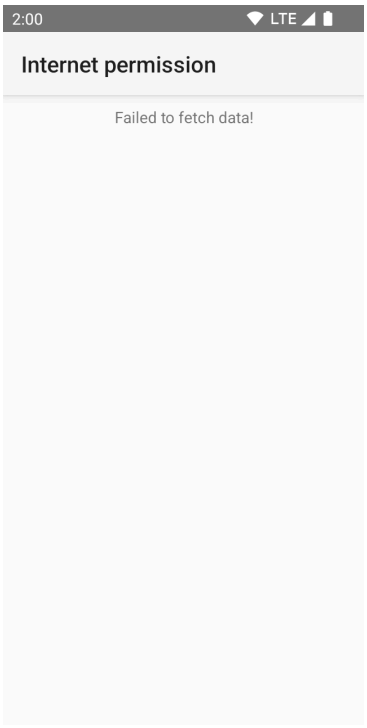

Figure 26: Internet permission denied on native Android.

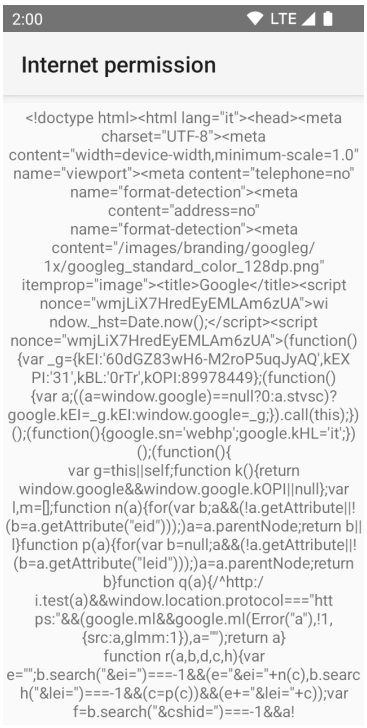

Figure 27: Permission not denied, even when not declared in manifest.

6.2 Real-World Example

To complement the controlled testing with the custom *TestApp*, the real-world application *Telegram* was selected for evaluating the virtual permission model's performance in practical scenarios, mainly for two reasons:

1. Complexity: it is a feature-rich application that extensively uses permissions, including access to the camera, microphone, location, and many other. This makes it a robust test case for evaluating the virtual permission model's ability to handle real-world apps behavior.
2. Compatibility: among the complex applications tested in VirtualXposed, Telegram emerged as one of the few capable of running successfully. Many other apps had limited compatibility, experienced crashes, or were unable to launch at all.

The evaluation of Telegram in the virtual environment produced the following positive results:

- Permission checks and requests: direct permission checks and all permission requests requests behaved as expected. The system returned accurate messages for permission states, as if the app were running in a standard Android environment, with error messages correctly displayed when permissions were denied in the virtual model.

- Content provider permissions: the model successfully enforced permissions for content providers. For example, operations involving contacts (e.g., reading or writing) were verified to work correctly with appropriate permission checks.
- Camera permission: the native camera patch, described in Section 5.3.5, was verified as working. Attempts by Telegram to access the camera without proper permissions resulted in a black screen, demonstrating that the patch effectively blocked access by intercepting native calls and enforcing permission checks.

Despite these successes, a critical flaw was identified: if Telegram or any other app ignored a `PERMISSION_DENIED` response and attempted to acquire a restricted resource anyway, the resource was granted without raising an exception. This issue reveals a limitation in the model's enforcement mechanism: while the redirection component effectively manages straightforward, direct requests for permissions, it struggles with cases where permission checks are handled within other operations or bypassed internally by the app.

Manual solutions, like the camera and content provider patches, address some of these gaps but do not provide complete coverage. A more comprehensive, universal system is needed to enforce permissions reliably, even for indirect or bypassed requests—something this model does not currently provide.

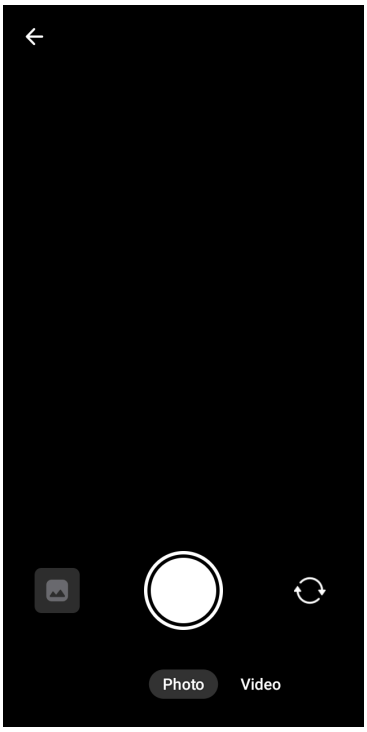

Figure 28: Camera resource denied in the virtual environment.

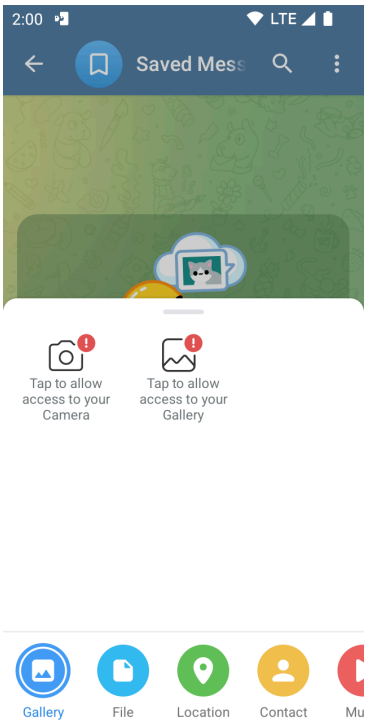

Figure 29: Telegram restricting Gallery, although it could easily access it without permission.

7 Discussion

Chapter 6 highlights significant challenges in implementing a robust and scalable virtual permission model. These primarily arise from the limitations of IPC calls to system services in app-level virtualization, the absence of a comprehensive permission mapping for Android, and the complexities of manual hook implementations.

7.1 Services IPC Calls

A critical limitation comes from the nature of Android's architecture, where many permission-sensitive operations are delegated to system services via Binder IPC calls. Once virtual apps cross the virtualization boundary via IPC, the execution logic is handled entirely by the Android system service. This prevents the virtualization framework from intercepting or redirecting further API calls, because the execution flow now moved to a separate, elevated process.

System services typically perform internal permission checks at this stage, making the virtualization framework unable to control the result, unless specific hooks are implemented for these operations beforehand.

7.2 Potential Solutions

While various solutions could address these challenges, each has significant limitations.

7.2.1 Manual Hooks

One possible solution is to manually create hooks for specific methods that interact with sensitive resources. This requires replicating the permission enforcement logic of Android system services within the virtualization framework.

This method was explored by realizing the patches described in Section 5.3.5, where manual hooks were developed for the camera and content provider operations. These successfully enforced permissions in those specific cases and served as a proof of concept for this approach, evaluating the effectiveness and complexity of this approach.

The evaluation however, revealed that the solution is not scalable. Each method hook requires a deep knowledge of specific Android framework sections, and often leads to reimplementing

significant portions of their logic. Considering the large number of methods and resources that require permission checks, this manual approach quickly becomes unmanageable for a large-scale implementation.

7.2.2 Reinstantiating System Services

A different approach involves replicating full copies of Android system services directly within the virtualization framework. By redirecting all system services requests to these replicated services, the virtual permission model could have full control over permission checks. Since the services would exist entirely within the virtualization framework, dynamic proxies could be configured to intercept all their permission management-related calls.

However, this approach would demand substantial memory and computational resources to replicate and manage the system services. Furthermore, it would not be a simple plug-and-play approach, likely requiring extensive modifications to adapt system services to be instantiated in the virtualization framework. These adaptations would need to address compatibility issues, as system services are designed to operate as part of Android's core infrastructure, and also permission-related issues, since they make extensive use of system-related permissions and resources, which are not available to the container app.

7.2.3 Automated Permission Analysis

Another solution could involve intercepting every method call that could require a permission later on in the call stack, and preemptively throwing a `SecurityException` for operations requiring permissions not granted by the virtual model.

While theoretically comprehensive, this approach faces several practical challenges:

- Overly restrictive behavior: preemptively blocking methods could prevent a large number of legitimate operations to be performed, leading to frequent unexpected application crashes and a poor general user experience.
- Overpermissioning: users may respond to frequent crashes by granting all permissions to virtual apps, defeating the purpose of a fine-grained permission management that the model tries to address in the first place.
- Lack of a permission mapping: no current, comprehensive mapping exists that associates Android API methods with their required permissions. Efforts to create such mappings in the past, while useful, remain incomplete or outdated, making an automated solution not achievable.
- Evolving Android APIs: Android's API surface is vast and continuously expanding and changing. This complicates efforts to automate a solution, because even if an updated and exhaustive permission mapping existed, maintaining it would require continuous and periodic efforts that are resource-intensive and not sustainable over time.

The two most notable Android permission mappings were provided by *aexplorer* [25] and *PScout* [26]. Unfortunately, both projects have significant limitations and are outdated for modern Android versions:

1. aexplorer: it uses static analysis to identify permissions required by various Android APIs. However, as a static analysis tool, it may not address every dynamic execution path, making its mapping incomplete. The project's published results address versions up to Android 7.
2. PScout: it implements a dynamic analysis approach to track permission requirements at runtime, offering a more precise approach than aexplorer. However, the project has not been updated for over six years, and as a result, it is no longer valid for analyzing the permissions of current Android versions. Its published results, instead, are limited to Android 4.2 (Jelly Bean), which means that it does not even account for critical updates like runtime permissions introduced in Android 6.

Ultimately, both tools lack coverage of key modern Android components, such as *APEX modules* —introduced in Android 10—and the *androidx libraries*. Thus, neither is sufficient for providing a comprehensive permission mapping for modern Android APIs.

8

Conclusion

The virtual permission model presented in this thesis serves as a proof of concept for a generic and scalable permission management solution. By drawing inspiration from Android's native permission system, this model tries to address the limitations of traditional app-level virtualization frameworks' permission controls. The virtual permission model's design and implementation have been evaluated through both controlled and real-world testing, proving its potential to enforce permissions in a virtual environment.

One of the main aspects of the model is its integration with Android's mechanisms for checking and requesting permissions. The virtual permission model replicates and extends these core mechanisms, ensuring that permissions are managed correctly within the virtual environment. However, the evaluation phase revealed critical challenges when the model interacts with Android's system components, such as content providers and networking operations. These limitations highlight the inherent complexity of Android's framework and the difficulties of accurately replicating it in a virtual space.

While the current approach has proven effective for many basic use cases, it is clear that app-level virtualization reaches its limits when handling more complex, system-level interactions. This happens because system services—which typically handle permission checks internally for their operations—are difficult to intercept through app-level virtualization, due to their separation in the Android architecture. While the manual hook approach proved to be effective for specific scenarios, it lacks scalability. Additionally, methods such as automated permission analysis remain unfeasible without an up-to-date and comprehensive permission mapping.

An alternative solution that could be explored is to explore the approach proposed in Section 7.2.2, by trying to reinstantiate system services within the virtual framework. The limitations and complexity of this approach still remain to be verified, and it is essential to determine whether they truly are as significant as it appears. If these challenges can be overcome with further implementation, this approach would enable direct interception of system-level operations, providing a potential way to overcome many of the limitations of the current virtual model.

Looking ahead, OS-level virtualization presents a promising alternative for overcoming many of the constraints of app-level virtualization. It could provide more control over permission

checks and allow for a universal approach to enforcing permissions in system-wide operations. Android's inclusion of the kPVM hypervisor presents a particularly interesting starting point in this context. While the Android Virtualization Framework is not yet ready for practical use —due to the Microdroid OS not being fit for this specific context—it could offer a valuable foundation for secure OS-level virtualization in the future. Exploring this area could pave the way for more comprehensive and scalable solutions for permission management, ultimately making the virtual permission model more effective and flexible to the evolving landscape of Android applications.

Bibliography

[1] "Docker." [Online]. Available: https://www.docker.com/

[2] "Podman." [Online]. Available: https://podman.io/

[3] W. Song *et al.*, "Towards Transparent and Stealthy Android OS Sandboxing via Customizable Container-Based Virtualization," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, in CCS '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 2858–2874. doi: 10.1145/3460120.3484544.

[4] S. Pizzi, "VirtualPatch: Fixing Android Security Vulnerabilities with App-Level Virtualization," Master's thesis, University of Padua, 2022. [Online]. Available: https://thesis.unipd.it/bitstream/20.500.12608/32823/1/Pizzi_Simeone.pdf

[5] "Java API framework." [Online]. Available: https://developer.android.com/guide/platform/#api-framework

[6] "Non-SDK interfaces." [Online]. Available: https://developer.android.com/guide/app-compatibility/restrictions-non-sdk-interfaces

[7] "Privileged permission allowlisting." [Online]. Available: https://source.android.com/docs/core/permissions/perms-allowlist

[8] "Security-Enhanced Linux in Android." [Online]. Available: https://source.android.com/docs/security/features/selinux#supporting_documentation

[9] "LSPosed." [Online]. Available: https://github.com/LSPosed/LSPosed

[10] "EdXposed." [Online]. Available: https://github.com/ElderDrivers/EdXposed

[11] "VirtualXposed." [Online]. Available: https://github.com/android-hacker/VirtualXposed

[12] "VirtualApp." [Online]. Available: https://github.com/asLody/VirtualApp

[13] D. Dai, R. Li, J. Tang, A. Davanian, and H. Yin, "Parallel Space Traveling: A Security Analysis of App-Level Virtualization in Android," in *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*, in SACMAT '20. Barcelona, Spain: Association for Computing Machinery, 2020, pp. 25–32. doi: 10.1145/3381991.3395608.

[14] L. Zhang *et al.*, "App in the Middle: Demystify Application Virtualization in Android and its Security Threats," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 1, Mar. 2019, doi: 10.1145/3322205.3311088.

[15] C. Zheng, T. Luo, Z. Xu, W. Hu, and X. Ouyang, "Android Plugin Becomes a Catastrophe to Android Ecosystem," in *Proceedings of the First Workshop on Radical and Experiential Security*, in RESEC '18. Incheon, Republic of Korea: Association for Computing Machinery, 2018, pp. 61–64. doi: 10.1145/3203422.3203425.

[16] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. Styp-Rekowsky, "Full-fledged App Sandboxing for Stock Android," 2015, p. .

[17] W. Song *et al.*, "App's Auto-Login Function Security Testing via Android OS-Level Virtualization." [Online]. Available: https://arxiv.org/abs/2103.03511

[18] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, "Cells: a virtual mobile smartphone architecture," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, in SOSP '11. Cascais, Portugal: Association for Computing Machinery, 2011, pp. 173–187. doi: 10.1145/2043556.2043574.

[19] "Android Virtualization Framework." [Online]. Available: https://source.android.com/docs/core/virtualization

[20] "Tristate location permissions." [Online]. Available: https://source.android.com/docs/core/permissions/tristate-perms

[21] "PermissionController." [Online]. Available: https://source.android.com/docs/core/ota/modular-system/permissioncontroller

[22] "Application UID Reference." [Online]. Available: https://developer.android.com/reference/android/os/Process.html#LAST_APPLICATION_UID

[23] "Per-User Range Source Reference." [Online]. Available: https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/core/java/android/os/UserHandle.java#46

[24] "ActivityManagerService checkPermission method." [Online]. Available: https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/services/core/java/com/android/server/am/ActivityManagerService.java#5881

[25] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis," in *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX: USENIX Association, Aug. 2016, pp. 1101–1118. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/backes_android

[26] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: analyzing the Android permission specification," *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, [Online]. Available: https://api.semanticscholar.org/CorpusID:3401975