A Study of Application Sandbox Policies in Linux

Trevor Dunlap
North Carolina State University
Raleigh, North Carolina, USA
tdunlap@ncsu.edu

William Enck
North Carolina State University
Raleigh, North Carolina, USA
whenck@ncsu.edu

Bradley Reaves
North Carolina State University
Raleigh, North Carolina, USA
bgreaves@ncsu.edu

ABSTRACT

Desktop operating systems, including macOS, Windows 10, and Linux, are adopting the application-based security model pervasive in mobile platforms. In Linux, this transition is part of the movement towards two distribution-independent application platforms: Flatpak and Snap. This paper provides the first analysis of sandbox policies defined for Flatpak and Snap applications, covering 283 applications contained in both platforms. First, we find that 90.1% of Snaps and 58.3% of Flatpak applications studied are contained by tamperproof sandboxes. Further, we find evidence that package maintainers actively attempt to define least-privilege application policies. However, defining policy is difficult and error-prone. When studying the set of matching applications that appear in both Flatpak and Snap app stores, we frequently found policy mismatches: e.g., the Flatpak version has a broad privilege (e.g., file access) that the Snap version does not, or vice versa. This work provides confidence that Flatpak and Snap improve Linux platform security while highlighting opportunities for improvement.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

access control, sandbox policy, linux applications

ACM Reference Format:
Trevor Dunlap, William Enck, and Bradley Reaves. 2022. A Study of Application Sandbox Policies in Linux. In *Proceedings of the 27th ACM Symposium on Access Control Models and Technologies (SACMAT) (SACMAT '22), June 8–10, 2022, New York, NY, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3532105.3535016

1 INTRODUCTION

Desktop application security is undergoing a fundamental change. Over the past decade, mobile platforms have altered users' expectations about application security. Despite permission usability concerns [15, 16], the general notion that applications are security principals is now commonplace. This shift from user-based to app-based access control has had a significant positive impact on mobile platform security, and it is now arriving on desktop platforms. Windows 10 and macOS distribute sandboxed apps through their application stores and provide permission options in system settings that mirror mobile platforms.

Linux application distribution is also moving to per-application sandboxing, and Linux advocates believe this will be the dominant application distribution paradigm in the near future [8, 40, 42]. Actions by widely used distributions support this belief. In April 2020, Ubuntu made the Snap Store the *default* application installation mechanism, only using traditional repositories when a Snap is unavailable [39]. Many other popular distributions, including CentOS, Mint, and Fedora Workstation, ship with Flatpak support.

To our knowledge, there is no prior work studying the security protection provided by the Flatpak and Snap ecosystems. On the surface, any access control protections around an application would provide better security for the user, as there are currently none. For example, all traditional applications can read and write a user's files (e.g., browser authentication cookies) and access attached peripherals (e.g., a webcam). Nevertheless, some critics claim that Flatpak and Snap provide users a false sense of security, noting sandbox escapes and unpatched CVEs [17, 29]. While the observation that these issues are possible is correct, the issues are created by package maintainers, not the platform design.

Our work represents the first broad empirical analysis of sandbox policies defined for Flatpak and Snap applications. We downloaded all 919 Flatpak applications from Flathub and all 2,264 Snap applications from the Snap Store in September 2020, along with a refresh in July 2021 to provide a longitudinal analysis. We identified a set of 283 matching applications to answer the following research questions. **RQ1:** *What access control policy features do package maintainers use?* Flatpak and Snap use different sandboxing techniques and different policy languages. Flatpak policy relies on package maintainers providing largely open-ended policy arguments, whereas Snap provides a menu of permissions. **RQ2:** *How often do package maintainers attempt to approximate least-privilege?* Other application ecosystems (e.g., Android) are well known to be over-privileged [14, 38]. **RQ3:** *Do package maintainers specify correct and secure policy?* Sandbox policies require package maintainers to balance functionality and security. Under-privilege can lead to poor user experiences, whereas certain types of over-privilege can lead to sandbox escapes.

Our analysis led to the following major findings.

- *Flatpak and Snap indeed improve the security of Linux deployments.* In a sample of 283 matching applications, we find that 90.1% of Snaps and 58.3% of Flatpaks specify policies that prevent sandbox escapes. We found that package maintainers overwhelmingly use fine-grained permissions for system and session inter-process communication (IPC) rather than coarse-grained permissions that provide access to the entire system and session bus. Fine-grained permission choices limit the abilities of a malicious or compromised app. This data overturns and directly contradicts the common belief that there is little security benefit to Flatpak [17].

- *Flatpak and Snap maintainers clearly attempt least-privilege policies.* For Snap, when considering device, system IPC, and session IPC permissions, nearly 3.1 times as many apps use fine-grained permissions over coarse-grained permissions. For Flatpak, when looking at filesystem, device, system IPC, and session IPC permissions, on average, 1.7 times as many apps use fine-grained permissions compared to coarse-grained permissions. Over a ten-month period (September 2020 to July 2021), on average, 30.2% of apps changed their policy for both Flatpak and Snap. Broadly, the policy changes consisted of introducing new fine-grained permissions or transitioning coarse-grained permissions to fine-grained permissions. These findings suggest package maintainers are engaged in balancing functionality with security.
- *Despite the sincere effort, maintainers fail to specify correct, least privilege policies.* Application testing by a subject matter expert identified clear instances of both over- and under-privilege. This finding motivates the need for automated tools that provide policy suggestions for package maintainers during packaging.

Availability: Our scripts and data (sans the app packages) are available at https://github.com/wspr-ncsu/linux-app-sandbox.

The remainder of the paper proceeds as follows. Section 2 provides background on Flatpak and Snap. Section 3 presents our threat model and analysis goals. Section 4 describes our methodology. Section 5 details our findings. Section 6 discusses the threats to validity. Section 7 provides recommendations. Section 8 overviews related work. Section 9 concludes.

2 BACKGROUND

The Android and iOS mobile platforms have made app-based security principals commonplace. Per-application sandboxes are now emerging in Microsoft Windows, Apple macOS, and Linux desktop distributions. The inclusion of per-application sandboxes in desktop platforms has piggybacked on new software distribution methods (e.g., app stores in Windows and macOS). In Linux, where `apt` and `yum` based software repositories are common, sandbox adoption is being driven by distribution-independent software packages, with Flatpak and Snap emerging as the *de facto* package ecosystems.[1]

Flatpak and Snap have key similarities and differences. Both distribute applications through app stores, often through community-based efforts. Flathub is the *de facto* store for Flatpak, while the Snap Store is the *only allowed* store for Snap. The Flatpak and Snap platforms are loosely analogous to Android and iOS, respectively, in that Flatpak is open, enabling arbitrary app stores, and Snap is under tight control, only allowing the Canonical/Ubuntu managed Snap Store. In contrast to mobile platforms, the package maintainer in both Flathub and the Snap Store is commonly a different entity than the software developer, particularly for proprietary applications (e.g., Zoom, Slack, Microsoft Teams). This distinction between package maintainer and software developer is important for our threat model (Section 3). Finally, Flatpak is designed to run in a desktop session, whereas Snap is designed for both desktop and server applications. This paper focuses only on desktop applications.

[1]AppImage is a relatively unpopular third alternative. It does not provide access control protection, therefore we do not discuss it further.

Despite their high-level similarities, Flatpak and Snap have fundamentally different implementation strategies. Here, we focus specifically on the details relevant to the application sandbox. We separate our discussion into the sandbox enforcement and policy.

2.1 Sandbox Enforcement

Flatpak and Snap take significantly different approaches to enforcing per-application sandboxes. Flatpak uses primitives from popular container technologies (e.g., Docker). Specifically, it builds on Bubblewrap [10], which uses Linux's *user namespaces* (e.g., `cgroup`, `mnt`, `ipc`) to allow unprivileged users to use container features. Initially, Bubblewrap creates a new mount namespace and defines what parts of the filesystem are visible in the sandbox. Specific directory paths (e.g., the user's home directory) can be mounted into the namespace based on the sandbox policy. The use of Linux namespaces also ensures that `/proc` only shows processes in the app sandbox. Flatpaks also only have access to a loopback network interface by default. Finally, Flatpak further hardens the environment by using seccomp to restrict specific risky system calls (e.g., `ptrace`).

In contrast, Snap uses traditional OS mandatory access controls (MAC) techniques to sandbox applications. Specifically, it uses the AppArmor Linux Security Module (LSM), automatically creating AppArmor profiles that confine the sandboxed application. AppArmor defines file access control policy using file paths, which is more flexible than mount namespaces. Specifically, it allows a Snap to access some files but not all (e.g., hidden dot-files) in a specified directory. This difference has meaningful implications in our empirical study. By using AppArmor, Snap sandbox policies inherit its expressive enforcement capabilities, including the ability to enforce policies for Linux capabilities, network, mounting, pivot_root, ptrace, signals, dbus, and Unix domain sockets. To handle Apparmor enforcement limitations, Snap also uses seccomp to filter system calls and cgroups to control access to devices. Most sandbox protections are not available if a distribution does not use AppArmor (e.g., Fedora uses SELinux).

Both Flatpak and Snap applications can bypass some aspects of sandbox enforcement by using XDG desktop portals, which are a form of user-driven access control [34] that presents a trusted user dialog box before a privileged action can be performed. A full list of XDG desktop portals is available in the portal documentation [30]. Flatpak is designed for desktop applications, which provide access to XDG desktop portals by default. Any Snap with the `desktop` permission has access to XDG desktop portals. However, both Flatpak and Snap require application code modification to use XDG desktop portals, which is not a viable alternative for package maintainers who are not also the application developer.

2.2 Sandbox Policy

Flatpak and Snap applications are given *permissions* that define which interfaces and resources the application may access. In contrast to mobile platforms such as Android, where app permissions mediate remote procedure calls (RPCs) to middleware services,

```
{
    "app-id": "org.inkscape.Inkscape",
    "runtime": "org.gnome.Platform",
    "runtime-version": "3.38",
    "sdk": "org.gnome.Sdk",
    "command": "inkscape",
    "finish-args": [
      "--share=ipc",
      "--socket=x11",
      "--socket=wayland",
      "--filesystem=host",
      "--filesystem=xdg-run/gvfs",
      "--talk-name=org.gtk.vfs",
      "--talk-name=org.gtk.vfs.*"
    ],
    "modules": [
        {
            "name": "inkscape",
            "sources": [
                {
                    "type": "archive",
                    "url": "https://inkscape.org/gallery/item
                           /21571/inkscape-1.0.1.tar.xz"
                }
            ]
        }
     ...
    ]
}
```

Figure 1: Simplified JSON Flatpak Manifest for Inkscape

```
name: inkscape
version: 1.0.1-3bc2e813f5-2020-09-07
apps:
  inkscape:
    command: bin/inkscape
    plugs:
    - desktop
    - desktop-legacy
    - gsettings
    - opengl
    - wayland
    - x11
    - home
    - unity7
    - cups-control
    - removable-media
    - dot-config-inkscape
    slots:
    - inkscape-dbus
  viewer:
    command: bin/inkview
    plugs:
    - desktop
    ...
slots:
  inkscape-dbus:
    interface: dbus
    bus: session
    name: org.inkscape.Inkscape
```

Figure 2: Simplified Snap Manifest for Inkscape

Linux desktop distributions have a largely file-based access control perspective. Even devices are accessed via files. Outside of files, Linux's Desktop Bus (D-Bus) is the primary interprocess communication (IPC) mechanism used by graphical applications to communicate with system services and each other (though some legacy interfaces still use Unix domain sockets). D-Bus defines two types of buses: a single *system bus* allows applications to use system services (e.g., NetworkManager, BlueZ), and a per-login *session bus* allows multiple applications run by the user to communicate.

For both Flatpak and Snap, the package maintainer defines the permission policy within the application's manifest file. The Flatpak manifest can be defined in both JSON and YAML, while the Snap manifest can only be defined in YAML. The Flatpak and Snap permission policy specifications differ significantly. The Flatpak policy is much lower level, defining permissions as parameters to broad types of system interfaces (e.g., filesystem). This specification is much more flexible, allowing the use of wildcards in some cases. In contrast, the Snap policy uses pre-defined permission names. The specified permissions are then used to generate an AppArmor profile for the application. Example policies for Flatpak and Snap are shown in Figures 1 and 2.

2.2.1 *Flatpak Policy.*

Flatpak permissions are defined as arguments categorized in six types of interfaces: `filesystem`, `device`, `share`, `socket`, `talk-name`, and `system-talk-name`. The `filesystem` interface is the most flexible. Paths can be defined directly (e.g., `/path`, `~/path`) or via predefined names such as `home`, `host`, `host-os`, `host-etc`. It also supports `freedesktop.org`'s special XDG user directories,[2] e.g., `xdg-desktop`, `xdg-documents`, and `xdg-download`. For the `home` and `xdg-*` directories, an optional path can be appended for more refined access control. Interestingly, `host` filesystem access does not provide access to `/dev`. Rather, Flatpak uses a separate `device` policy definition, possible values are `dri` (graphics), `kvm` (virtualization), `shm` (shared memory), and `all`. The `all` value allows access to all device nodes in `/dev` *except* `/dev/shm`. Notably, the `device` policy specification is much less flexible than Snap. The `share` interface lists subsystems to share with the host system, supporting values of `network` and `ipc`. The `socket` interface also defines a fixed set of parameters: `x11`, `wayland`, `fallback-x11`, `pulseaudio`, `session-bus`, `system-bus`, `ssh-auth`, `pcscd`, and `cups`. In some cases, `socket` permissions are coarse-grained controls that subsume other policies. The `session-bus` and `system-bus` sockets provide full access to the D-Bus session and system buses. Instead, package maintainers should use the `talk-name` and `system-talk-name` policy interfaces to define fine-grained access to the D-Bus session and system bus. D-Bus interfaces accept arguments corresponding to established D-Bus namespaces. Wildcards are also permitted.

Flatpak can also provide per-application storage using the `persist` configuration. Using `persist`, the package maintainer can specify paths (e.g., `~/.myapp`) that cause the sandbox to bind a mount to a per-application location (e.g., `~/.var/app/org.my.App/.myapp`). As such, the `persist` configuration allows the package maintainer to keep the Flatpak application's configuration files separate from non-Flatpak versions of the application. In doing so, the package maintainer may eliminate or greatly reduce the need for the application to access files in the user's home directory.

Finally, end users can *override* Flatpak permissions, either for the entire system or just for the user account. Overrides for the user account are stored in `~/.local/share/flatpak/overrides`. Users can make these overrides via the `flatpak override` command or via the third-party Flatseal Flatpak application, which provides a graphical interface to configure Flatpak permissions.

2.2.2 *Snap Policy.*

The Snap sandbox policy uses *plugs* and *slots* to define permissions. Conceptually, a plug grants privilege to a

[2] Note that XDG user directories are different than XDG desktop portals. XDG directories provide user-configurable locations for common user directories (e.g., `Documents`).

consumer (e.g., RPC caller), whereas a slot defines privilege requirements to access a provider (e.g., RPC callee). Slots allow package maintainers to define custom permissions for the application's interfaces and protect key interfaces and resources. Slots are useful for Snaps packaging system services (which are not our focus).

Snap defines a wide variety of plugs that define access to the file system, network, IPC, devices, and other system resources. In contrast to Flatpak permissions, Snap's plugs are very specific: e.g., `audio-playback`, `camera`, `joystick`, `pulseaudio`, `bluez`, `gpg-keys`, `ssh-keys`. Of interest is the `home` plug, which allows access to non-hidden files owned by the user in the `$HOME` directory. Some plugs have associated attributes that parameterize the plug. For example, the `personal-files` plug has `read` and `write` attributes that define sets of files and directories that can be accessed. A full list of the Snap supported interfaces can be seen in the documentation [37].

Similar to traditional Linux MAC access control models, Snap package maintainers must also specify a *confinement mode*, which may be `strict`, `classic`, or `devmode`. Most Snaps use `strict` mode, indicating the plug and slot policy should be enforced. In contrast, `classic` mode treats the application as a traditional package (i.e., no confinement), which requires manual approval by the Snap Store to ship. Finally, `devmode` logs policy denials instead of enforcing them.

Not all plugs are enabled by default. The Snap permission documentation [37] includes an *auto-connect* field. When auto-connect is "no," users must manually enable the permission. In contrast to Flatpak, there is an upper bound to the permissions that the end-user can grant. However, this default auto-connect behavior for plugs does not apply to all applications. Package maintainers can appeal to the Snap Store for their application to have auto-connect enabled. Requests approved by the Snap Store are distributed to end hosts in the form of a signed snap-declaration and available via the `snap known snap-declaration` command.

3 THREAT MODEL AND ANALYSIS GOALS

For the purposes of this paper, we consider three entities: *software developer*, *package maintainer*, and *end-user*. Consistent with traditional uses of sandboxes, we consider two scenarios. In the first scenario, the application is benign but may be exploited (e.g., rendering a malicious PDF), so all three entities are mutually aligned. In the second scenario, the application logic acts against the interests of the end-user by performing either malicious or privacy-invasive actions. In this scenario, the software developer is not trusted, but the package maintainer and the end-user are trusted. We do not consider scenarios where the package maintainer is malicious (e.g., including a malicious library in the package). Trusting the package maintainer is reasonable because in app stores such as Flathub, the build configuration is publicly available (https://github.com/flathub), and the build process is automatic.

Our trusted computing base (TCB) includes the sandbox enforcement mechanisms used by Flatpak and Snap. We trust that the mechanisms provide complete mediation and, given a proper policy, are tamperproof. Snap's AppArmor enforcement mechanism uses the well-studied LSM interface [19, 23, 47]. Auditing Flatpak's Bubblewrap [10] sandbox is beyond the scope of this paper. Our TCB also includes the Linux kernel, system daemons, and their respective access control enforcement and policies.

This paper is concerned with the *sandbox policies* defined by the package maintainer. Policy weaknesses can take two forms.

Over-privileged Policy: Ideally, the sandbox policy should follow the principle of *least privilege* [35]. In practice, true least-privilege policies are not achievable. Similar to prior work studying over-privilege in Android applications [14], we seek to characterize how well the sandbox policies defined by Flatpak and Snap package maintainers approximate least privilege. In contrast to Android, where API to permission mappings can be created by analyzing the middleware source code [1, 5], Linux applications access resources in a myriad of ways and use a variety of programming languages. Also, in contrast to Android, Flatpak and Snap provide both coarse-grained and fine-grained permissions for accessing similar resources. In this paper, we study over-privilege from the perspective of how policy is specified. It also uses runtime testing of the same application to identify over- and under-privilege.

Over-privilege is a multi-dimensional question. An application can be over-privileged in one dimension (e.g., filesystem) and under-privileged in another dimension (e.g., device access). Therefore, our characterization considers the following dimensions:

Filesystem permissions provide access to files and are typically specified as a file path.

Device permissions provide access to an internal device or external peripheral. Such permissions may provide access directly to a device node in `/dev` or indirectly through a system service that is devoted to that device (e.g., PulseAudio).

Network permissions allow the application to communicate with servers and applications on other hosts.

System IPC permissions provide access to interfaces of system daemons, which provide host-level configuration or functionality (e.g., NetworkManager).

Session IPC permissions provide access to interfaces of other applications or daemons running within the user's session, often as the user's identity.

For each of these dimensions, our analysis identifies if an application has: (1) *no access*, (2) *fine-grained access*, or (3) *coarse-grained access*. Note that we only consider the policy specification, and "no access" may include access through the XDG desktop portal trusted UI.

Policy Tamperproofness: The tamperproofness of a reference monitor [4] is contingent on the tamperproofness of both the enforcement and the policy. A sandbox policy is not tamperproof if it allows an application to perform unconfined execution outside of the sandbox. For Flatpak applications, this can occur in two ways. First, the policy may allow the application to write to a configuration file (e.g., `~/.bashrc`) that results in code execution outside the policy. Second, the policy may allow the application to change its policy (e.g., writing to the per-user policy overrides in the user's home directory [22]).[3] For Snap applications, unconfined execution only occurs when the application executes in `classic` or `devmode` confinement modes. A Snap application with the `home` permission cannot write to `~/.bashrc`, because the `home` permission does not

[3]At the time of writing, applications with the `home` permission could access ~ /.local/share/flatpak/overrides; more recent versions of Flatpak require explicit permission to access this directory.

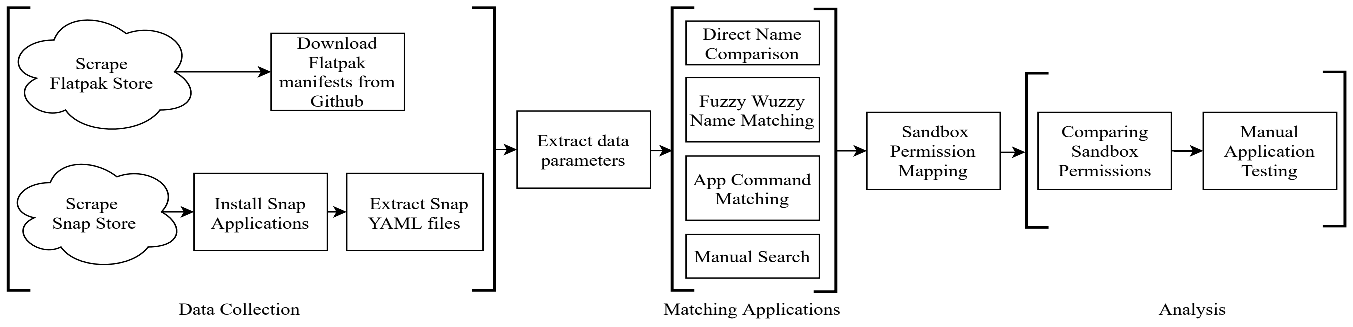

Figure 3: A high-level architecture of our approach

give access to hidden files (i.e, those that begin with a .).[4] This difference in policy semantics highlights an important contrast in the enforcement mechanisms used by Flatpak and Snap. Flatpak grants access to files based on mount points, whereas Snap's AppArmor enforcement can define policy to file paths.

Finally, certain aspects of the runtime environment may prevent the sandbox reference monitor from achieving complete mediation. For example, it is well known that X11 applications can eavesdrop on the keyboard input for other applications. Removing the X11 permission from applications is not a solution, as X11 is still commonly used by Linux distributions. Fortunately, if the user is running the Wayland windowing system and the application has the X11 permission, it cannot eavesdrop on keyboard input for other applications. Since it is possible to safely run an application with the X11 permission, we do not consider it a vulnerability.

4 METHODS

This section describes the methods used for answering our research questions. Figure 3 displays the high-level methodology of our approach. A key challenge was to compare the Flatpak and Snap ecosystems directly. Comparing sandbox policy specifications is meaningless if the subject applications are not comparable, such as comparing server applications to desktop applications. Therefore, a set of matching applications between ecosystems is required to make a comparison. Finally, Flatpak and Snap permissions are not directly comparable. Therefore, we identified semantic groupings of permissions that apply to both ecosystems.

4.1 Data Sets

Our analysis considers two data sets: *all applications* and *matching applications*. However, the *all applications* is not particularly useful for comparison. Flatpak is primarily GUI applications (98.9%), whereas Snap includes GUI (62.8%), command line, and server applications. The set of matching applications provides a direct comparison of the security posture between the two ecosystems.

Data Collection: To identify all applications available in each store, we developed a web scraping script using the Python BeautifulSoup library [9]. At the time of data retrieval (September 19, 2020), 919 applications were available from the Flathub store, and 2,264 applications were available from the Snapcraft store. In addition to the sandbox policy's package manifest, we also collected metadata, including application name, application refresh time, application version, and application category. Apart from the initial data retrieval, we also refreshed a set of matching applications for our longitudinal study within Section 5.2.1 on July 7, 2021.

Policy Extraction: Both Flatpak and Snap define the sandbox policy (permissions) in the package's manifest file. Given the open nature of Flathub, we were able to retrieve each application's manifest file directly from Flathub's Github project (https://github.com/flathub). Due to the closed nature of the Snap Store, retrieving the manifest file for Snaps was more difficult. For each Snap, we installed the package to our local machine and extracted the `snapcraft.yaml` file from the application's build directory.

We created Python scripts to parse the three types of manifest files: JSON and YAML for Flatpak and YAML for Snap. We extracted the sandbox permissions and application execution command property. For Flatpaks, the sandbox permissions are listed as `finish-args` (see Figure 1), whereas for Snaps they are listed under the `plugs` attribute of each app component (see Figure 2).

4.2 Matching Applications

The set of matching applications approximates the desktop applications desirable by the Linux community and enables a comparison between the Flatpak and Snap ecosystems. We first normalized the application names to perform the matching, filtering out punctuation, special characters, and numbers. Normalization alone identified 215 matching applications. To increase this set, we used the application execution commands extracted from the manifest file. By matching execution commands, we matched an additional 42 applications. While the execution command is a valuable matching heuristic, it is not sufficient on its own. In fact, only 184 of the exact application matches also had a match in the execution command.

Finally, we used Python's FuzzyWuzzy module to calculate the Levenshtein distance between the remaining normalized application names. For example, FuzzyWuzzy identified a match between "Airtame" (Flatpak) and "Airtame Application" (Snap), which was not matched by direct string comparison. We conservatively set the FuzzyWuzzy ratio to 50, which generated false-positive matches. The list of matches was manually inspected for accuracy. Any unmatched applications not present in the initial FuzzyWuzzy subset or previous methods were searched, using the application name, on

[4]We note that a Snap application with the home permission can overwrite user-owned executables (e.g., in `~/bin`), which may lead to unconfined execution. However, since such attacks are user-specific, we do not consider them in this paper.

Table 1: Permission groupings used by our analysis

Permission Group	Description	Flatpak Examples	Snap Examples

Filesystem	Access to user and system files	`--filesystem=host`
		`--filesystem=home`
		`--filesystem=/path/`
			`home`
			`personal-files`

Device	Access to devices and peripherals	`--device=all`
		`--socket=cups`
			`pulseaudio`
			`joystick`

Network	Network access	`--share=network`	`network-bind`
System IPC	Access to system daemons and services	`--system-talk-name=[name]`† `--socket=system-bus`	`system-observe` `network-manager`
Session IPC	Access to session apps and daemons	`--talk-name=[name]`†	`gsettings` `calendar-service`

Graphics	Windowing and graphics APIs	`--device=dri`
		`--share=ipc`
			`wayland`
			`opengl`

† `[name]` includes names such as `org.gtk.vfs` and `org.gtk.vfs.*`

the opposing app store, adding 26 matching applications. In total, we identified 283 matching applications. The full list of matching applications can be seen in our online appendix.[5]

4.3 Permission Normalization

Simply having matching applications is insufficient to compare the Flatpak and Snap ecosystems. We created permission groups, based on the privilege dimensions defined in Section 3, to coarsely group the very different types of permissions used by Flatpak and Snap. Three co-authors reviewed our permission group classification, each of which has one to two decades of experience using Linux.

4.3.1 Permission Groups. We performed a permission group classification based on the entire set of permissions used by the Flatpak and Snap applications in our *matching applications* dataset. Table 1 lists and describes our six permission groupings as well as several example permissions for both Flatpak and Snap. While the permission groups roughly correspond to Flatpak's `finish-args` types, there are several notable differences. Specifically, we took into account the end resource being accessed rather than the mechanism for access. For example, PulseAudio and CUPS are accessed via daemons, but they ultimately give access to devices (i.e., speakers, microphones, and printers). Finally, we created the "Graphics" group (not in Section 3) for windowing and graphics environment permissions to reduce noise in our comparative analysis. For example, X11 requires Flatpak applications to include `share=ipc`, and in some cases `device=dri`, which is technically a device, but also related to graphics. Snap has a similar set of permissions that may otherwise be classified as system IPC.

4.3.2 Identifying Privilege Use. Section 5 calculates the dimensions of privilege granted to applications based on the permission groups. For each privilege, an application is counted exactly once and categorized as either *no access*, *fine-grained access*, or *coarse-grained access*. If an application has at least one coarse-grained permission in a permission group, the application is counted as coarse-grained for that privilege. Otherwise, if it has at least one fine-grained permission for a permission group, it is counted as fine-grained for that privilege. If it has no permissions in a permission group, it is counted as no access for that privilege. Our complete classification of permissions as fine-grained and coarse-grained is online. [6]

Snap policy has three additional complexities that warrant mention. First, only some permissions are granted by default (i.e., `-auto-connect` is `yes`). Section 5 shows both the default privilege and the privilege if users grant all available permissions. When determining the default privilege for an application, our analysis accounts for the `auto-connect` overrides approved by the Snap Store. Second, Snap allows the package maintainer to define a different sandbox policy for each binary executable in the application. This policy union is needed to compare application-level privilege between Flatpak and Snap ecosystems. It also represents the fact that privilege-separated components can either collude or exploit one another [12]. Therefore, from a threat model perspective, the union is appropriate. Third, Snap applications with `classic` confinement mode can access all resources, counted as coarse-grained for each permission group.

4.4 Manual Testing of Applications

When assessing **RQ3**, a subject matter expert tested a set of applications to understand why package maintainers set some permissions. The focus is on network and filesystem permissions because they are the clearest to evaluate manually from a standard desktop. The goal is to understand the broad permission requirements of either filesystem or network access set by the package maintainers. Therefore, it was not necessary to exercise every code path during analysis. Instead, our procedure for application testing is as follows.

Initially, the tester reads the application's description to familiarize themselves with what the application does. The application is then installed from the defacto Flatpak and Snap stores. Next, the application is opened, ensuring the same binary entry point for Flatpak and Snap. Any necessary actions (e.g., account creation) were followed to get the application to work. Once the application is running, the exploratory phase of the main menu or the menu bar begins, selecting each feature with the intent of either accessing the user's filesystem or requesting network access. Generally, the need for filesystem or network access is a clear path for GUI-based applications. A common need for filesystem access is to load a file,

[5]https://github.com/wspr-ncsu/linux-app-sandbox/blob/main/data/matching.md

[6]https://github.com/wspr-ncsu/linux-app-sandbox/blob/main/data/permissions.md

such as saving an output file or uploading a custom file to the application. Common network access features include downloading new games or uploading data to the network.

If an application contains a permission, but no feature is found within the application to need the permission, the application is considered over-privileged. Multiple steps follow to confirm over-privilege. First, the tester searches online for documentation about the application, looking for features requiring the specific permission. If no information is found online regarding how to invoke the need for such permission, it was then removed from the application and re-tested. During the re-test, the tester checked the application for no apparent functionality lost. The functionality after removing the permission was confirmed to match the opposing platform's application that was released without the specific permission. Significant time was spent on each application before classifying it as over-privileged, as the classification is not taken lightly.

In contrast, if we found a function broken due to the lack of permission, we considered this application to be under-privileged. Broken functionality generally results in an error message. If the application was difficult to use due to the lack of a permission (e.g., transferring a file into the sandbox to access due to a missing file system permission), compared to the application on the opposing platform using the specific permission. We then considered the application with the missing permission to be under-privileged.

5 RESULTS

Here, we use the matching applications dataset and the normalized permissions to compare the Flatpak and Snap ecosystems. We start by understanding the policy features in use, then approximate least privilege, and end with general policy correctness and security.

5.1 Policy Features in Use

We begin addressing **RQ1** What access control policy features do package maintainers use?

Finding 1: *Applications use a large number of policy features.* When considering the all applications dataset, we found that the Flatpak ecosystem uses 162 unique permissions, while the Snap ecosystem uses 142 unique permissions. Several factors can explain the differences in permission count between the systems. Flatpak supports several runtimes, such as GNOME, KDE, and Freedesktop, each having its own portal permissions. Note that for Flatpak, we do not separately count permissions to specific file paths (e.g., `/home/user/.app/.config`), though we separately count defined XDG file permissions (e.g., `filesystem=xdg-download`). Many policy features enable fine-grained access control at the potential cost of complicated policy specification and maintainer burden.

On average, Snaps use 9.29 permissions in matching applications while Flatpaks use 7.03 permissions. Therefore, demonstrating the size of a "typical" policy is non-trivial, and Snaps tend to use more permissions than Flatpaks for roughly equivalent policies. [7] Next, we consider how policies vary among applications.

Finding 2: *Despite the complex menu of permissions, few are frequently used.* Of matching applications, 50.0% of Flatpaks only use a subset of the 11 most common permissions, while 55.2% of Snaps

[7]We exclude classical confinement apps from this analysis, as these apps do not need to declare a permission for an activity.

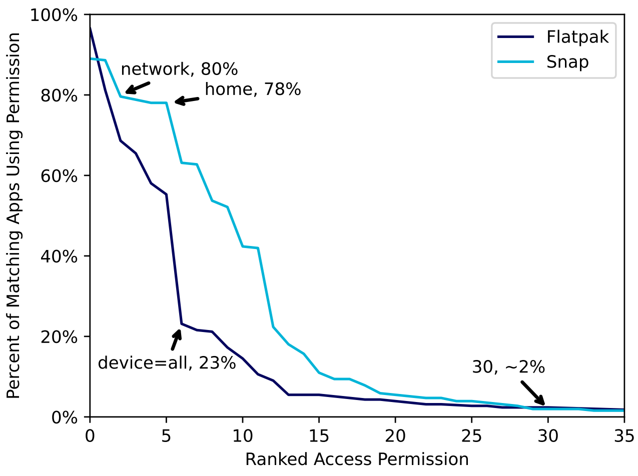

Figure 4: Ranked Access Permission Frequency for 255 Matching Apps (Excluding Snap Classic Confinement Apps)

only use a subset of the 16 most common permissions. Figure 4 displays the commonly occurring access permissions across each platform and how many applications use them. The x-axis displays the *rank* of permissions for matching Flatpak and Snap applications, with the most common permissions on the left. Equivalent permissions across the two platforms may be ranked differently. For example, the most used permission across matching applications in Flatpak is `share=ipc`, while in Snap, it is the `desktop` permission. Each highly ranked permission is relatively coarse-grained, including entire home filesystem access, network access, all devices, and host filesystem access, which satisfy most of the confinement requirements for applications. Both of the platforms start to taper off after the 16th permission. We next investigate how coarse- and fine-grained permissions use used.

5.2 Approximating Least Privilege

In this section we investigate **RQ 2:** How often do package maintainers attempt to approximate least-privilege?

Finding 3: *With the exception of filesystem, package maintainers use fine-grained permissions when available.* Figure 5 displays whether applications have coarse-grained access, fine-grained access, or no access to a particular permission group. Only the filesystem, device, system IPC, session IPC, and graphic groups offer fine-grained permissions. Network permissions are only coarse-grained on both platforms, and graphics permissions are only fine-grained, so we do not further discuss these categories. We distinguish between the requested and default (i.e., auto-connect) for Snap permissions.

Both platforms offer a significant amount of alternatives to coarse-grained permissions in terms of IPC. For Flatpak, 7% of matching apps used the fine-grained system IPC, and zero apps used the coarse-grained system IPC permissions. We see 5% of apps using fine-grained system IPC permissions compared to the 10% of apps using coarse-grained system IPC permissions for the Snap default settings and 21% of apps requesting fine-grained IPC permissions. For session IPC permissions, 42% of matching Flatpak

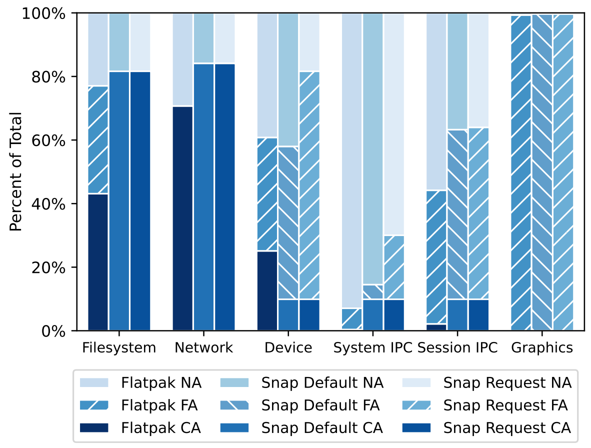

Figure 5: Breakdown of Matching Applications for No Access (NA), Fine-Grain Access (FA), and Coarse-Grain Access (CA) Permissions. Snap "Default" `auto-connect: yes` and "Request"all specified permissions.

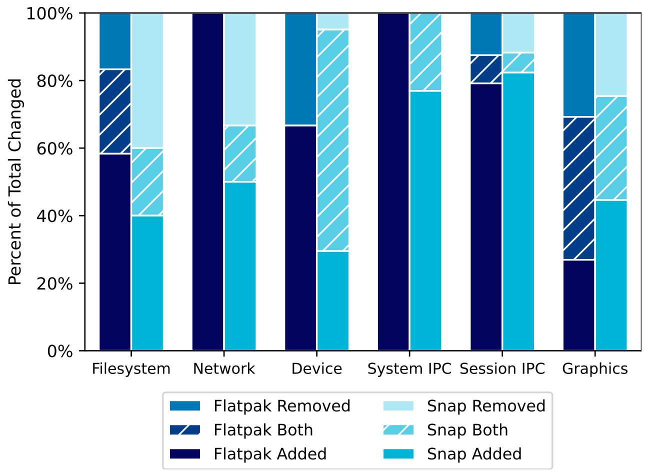

Figure 6: Evolved Sandbox Permissions of Matching Applications from September 2020 to July 2021

apps use fine-grained permissions compared to the 2% using coarse-grained permissions. For Snap, we saw five times as many apps using fine-grained system IPC permissions compared to coarse-grained system IPC permissions.

Snap package maintainers use a filesystem access approach that allows for either no access, home access, or host access. While a `personal-files` permission is available for specific file access, it requires approval and is primarily for reading hidden files. Only one Snap from the matching applications dataset uses the `personal--files` permission. On the other hand, Flatpak offers finer-grained options through portals or specific folder locations (e.g., `xdg-documents`). Of the matching applications, 9.9% of Snap applications and 27.6% of Flatpak applications have host filesystem permissions. Consequently, this observation debunks speculation that the majority of applications have host filesystem access [17]. The likely cause of the difference between Flatpak and Snap is the vetting process for the `classical` confinement in Snap. For Flatpak, 34% of matching apps use fine-grained filesystem permissions, while 43% of matching apps use coarse-grained filesystem permissions.

For device permissions, Snap provides fine-grained permissions to devices (e.g., for joysticks, cameras, and other `/dev` devices), while Flatpak only provides a coarse-grained permission (e.g., `device=all`) and limited fine-grained permissions. Despite the limited permissions in Flatpak, we still see 36% of matching Flatpak apps using fine-grained permissions compared to 25% with coarse-grain access, demonstrating that access to `/dev` is not as common on desktop applications. For Snap, we saw on average six times as many matching applications using fine-grained device permissions compared to coarse-grained device permissions.

When considering filesystem, device, and IPC permissions for Flatpak, on average, 1.7 times as many apps are using fine-grained permissions compared to coarse-grained permissions. For Snap, when looking at the device and IPC permissions, nearly 3.1 times as many apps use fine-grained permissions over coarse-grained permissions. These results suggest package maintainers use least-privilege policies that require fine-grained permissions.

5.2.1 *Policy Changes over Time.*

To understand how the ecosystem is changing over time, we first assessed how permissions in the same matching applications in September 2020 changed over a ten-month period (September 2020 to July 2021).

Finding 4: *On average, 30.22% of applications changed their policy over a ten-month period.* Demonstrating the ecosystems are still in flux and evolving. Of the unique permissions added to matching Flatpak applications over the ten months, 90.20% were fine-grained. For matching Snap applications, 90.91% of the added permissions were fine-grained. Broadly, when an existing permission group is modified, package maintainers either add new fine-grained permissions or change coarse-grained permission to fine-grained permissions.

Figure 6 displays the breakdown of categories and how package maintainers changed the permissions in each for Flatpak and Snap. The *Added* portion represents applications that only added permissions. The *Removed* portion represents applications that only removed permissions. The *Both* portion represents applications that added permissions and removed permissions. The following paragraphs highlight examples of these changes.

Filesystem: For filesystem access, 27 of the 30 applications that added filesystem permissions for Flatpak were fine-grained. The most common filesystem access removal was of the `filesystem=home` permission. For example, the Signal Desktop private messenger application in Flathub was originally granted full home access. The package maintainer removed the `filesystem=home` permission and replaced it with the safer `filesystem=xdg-*` paths, demonstrating a sandbox policy improvement and a transition towards fine-grain policies. For Snap, two applications added the `home` plug, two removed the `home` plug, and one transitioned from classic to strict.

Network: Flatpak introduced network access, while Snap mixed between adding and removing network access. Seven Flatpak applications introduced the `network` permission. The Snap Skype application transitioned from classic to strict confinement, still maintaining network access, demonstrating stricter access privileges.

Device: Flatpak package maintainers introduced minor changes

with device permissions. Two added `device=all`, two added `device=shm`, and two removed `device=shm`. In Snap, 40 applications transitioned from the `pulseaudio` plug to the `audio-playback` plug, likely addresing security issues inherited with PulseAudio [33].
System IPC: Only two Flatpak applications changed system IPC access. One added `socket=system-bus` permission, a highly permissive system IPC permission. The other added a fine-grained permission for the network monitor portal. For Snap, most of the applications added the fine-grained `network-manager-observe` plug.
Session IPC: Flatpak applications saw the most significant addition of virtual file system portal access, which aligns with XDG directories' transition. The change demonstrates sandbox policy improvement for applications and does not open permissive holes in the sandbox policy. The two most common changes for Snap were the addition of the `screen-inhibit-control` or the `gsettings` plug, both of which are fine-grain permissions.
Graphics: Graphics saw a unique split of how Flatpak package maintainers handled permissions. Half of the applications were split between either adding Wayland support and removing X11 or adding X11 and removing Wayland. We expect to see applications experimenting with Wayland support before making complete transitions away from X11. For Snap, 23 applications added Wayland support, and 22 applications added OpenGL support to access the system GPU, as most were gaming applications.

5.2.2 *Date of Last Package Update.*

To contextualize our finding that 30.22% of applications changing their policy over the analysis period, we also investigated what proportion of packages that updates regardless of if policy change. Having up-to-date software with security patches is an essential security measure.

Overall, we found that packages are more actively maintained in Flatpak than Snap. In total applications, 21.6% of Flatpak applications and 49.1% of Snap applications had not been updated since December 2020 (as of July 2021). For matching applications, 82.7% of Flatpak and 75.8% of Snap application packages were updated in 2021. We also note packages may or may not be "up-to-date" relative to the upstream project, as visibility is unavailable.

5.3 Policy Correctness and Security

In this subsection, we address **RQ3** Do package maintainers specify correct and secure policy?

5.3.1 *Permitting Sandbox Escapes.*

Finding 5: *41.7% of matching Flatpak applications can escape the sandbox, while only 9.9% of matching Snap applications run unconfined.* As mentioned in Section 3, clear sandbox escapes can be created through policy weakness. For Flatpak, the ability to tamper with the sandbox can come when granted either `home` or `host` filesystem access. Of the matching applications, 14.1% request `home` filesystem access and 27.6% request `host` filesystem access. These two permissions, by default, allow the ability to write to hidden files (e.g., the user's `~/.bashrc`), allowing the injection of command that would execute outside of the sandbox. A secondary example is to override the user's policy within the user's home directory, allowing for privilege escalation, but we did not find any instances within matching Flatpak applications. Note that package maintainers can define *read-only* permissions for filesystem access. However,

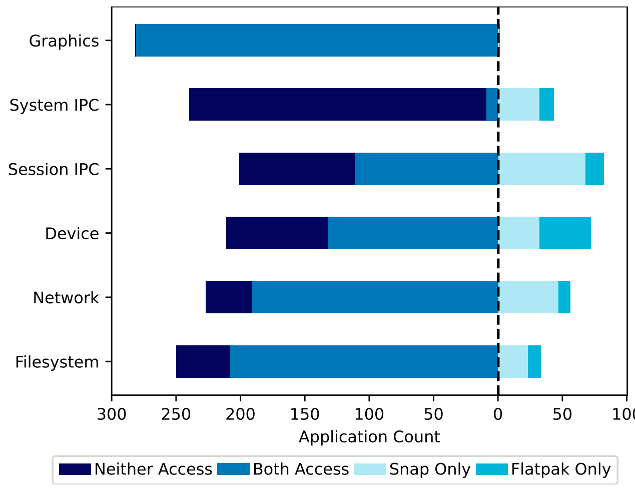

Figure 7: Default Access Permissions for 283 Matching Apps

only one matching application requests read-only `home`, and only three matching applications request ready-only `host`.

In contrast, Snaps cannot technically escape the sandbox. Recall that Snap's `home` plug does not allow writing to hidden files in the user's home directory. However, 9.9% of matching Snaps have `classic` confinement which does not run in a sandbox.

5.3.2 *Inconsistent Policy.*

We now investigate how matching applications differ in their use of broad categories of permissions: filesystem, network, device, session IPC, system IPC, and graphics. Because we consider broad privilege, we capture clear functional differences between applications, not minor differences in the specificity of a permission. For example, applications are designed to either use the filesystem or not. Permission errors can be over-privilege (unneeded permission was added) or under-privilege (needed permission was not provided). Figure 7 depicts the proportion of matching apps that are consistent (left of the line) or inconsistent (right of the line) in each permission group. Note, these are the default permissions, the auto-connect aspect of Snap.

Finding 6: *We found approximately 75% of matching applications declare inconsistent permission groups.* We note that this finding counts apps, not individual permission group differences, which is greater. The graphics permission group is negligible due to the GUI aspect of matching applications. The difference for system IPC comes from the classical confinement applications in Snap when Flatpak does not request any system IPC access.

The session IPC permission group had the greatest number of inconsistencies, with 82 (29%) differing applications, the majority of which, 68 (24%), were with Snap applications only accessing the session IPC. This inconsistency can be misleading, as the difference resulted from Snaps having the `gsettings` plug. Flatpak automatically allows access to gsettings without a specific permission.

The next most significant difference is the device permission group, which contained 72 (25%) differing applications. PulseAudio is the driving factor for differences in device access, in which 36 Flatpak applications use PulseAudio. Given that our analysis only considers default permissions for Snap (`autoconnect: yes`), the

Table 2: Under/Over-Privilege Analysis

	Under-privilege		Over-privilege	
	Flatpak	Snap	Flatpak	Snap
Filesystem	10	7	2	13
Network	9	0	5	37

differences are not surprising, since PulseAudio is not `auto-connect` due to security concerns [33]. When considering all requested permissions for Snap, we see a flip in PulseAudio requests, with 32 apps from Snap requesting PulseAudio when Flatpak does not.

Finally, filesystem and network access permissions represent crucial differences that define the application's functionality. For Snap, 47 applications declared network permissions undeclared by the matching Flatpak, while only 9 Flatpaks had network permissions undeclared by the matching Snap. In total, 56 (20%) applications contain differing network permissions. In the filesystem permission group, we found a total of 33 (12%) inconsistent applications, with 23 Snap applications requesting filesystem access and 10 Flatpak applications requesting filesystem access.

5.3.3 Manual Runtime Analysis. This subsection presents our findings from a manual analysis on 33 applications with differing filesystem permissions and 56 applications with differing network permissions. We excluded one application from the filesystem results and five applications from the network results after determining either a relevant binary was not present or would not execute correctly.

We focus on network and filesystem permissions because they are the clearest to evaluate manually from a standard desktop. We evaluated permission use in the broadest possible terms: any attempted or confirmed use of the filesystem or network activity was sufficient to determine if an application should have or not have permission. If a matching Flatpak and Snap make the same error, it would not be included in this analysis. As a result, these findings are a conservative *lower bound on permission errors*.

Finding 7: *Permission inconsistencies across matching applications suggest the confusion of application functionality.* The below results demonstrate how inconsistencies in permissions requests across matching applications suggest that confusion in defining policy is present. Table 2 reports our findings from the manual analysis.

Filesystem Analysis: Given the complexity of these permission systems, combined with the near-ubiquity of filesystem use, it is perhaps unsurprising to find some over-privileged apps. That said, the overall base rate of 5.2% filesystem over-privilege is relatively low, though it is a lower bound by construction.

A case study of filesystem over-privilege is the game *Mr. Rescue*, an arcade-style fire fighting game. The Snap app requested the `home` permission, but the Flatpak did not. In an exhaustive examination of the interface, we saw no indication of filesystem access. To further validate, we manually removed the `home` plug from the Snap and retested it. We found the game fully functional, including score history, indicating over-privilege.

While over-privilege may be expected to preserve functionality when permission needs are unclear, we found it surprising that more inconsistent applications had *under-privileged* filesystem permissions. Under-privilege does not directly indicate a compromise risk, though it does present a loss of functionality. Frustrated users could abandon the relevant app or seek it out from a less secure source (e.g., a non-sandboxed repository or an unofficial source).

An example of filesystem under-privilege is the NordPass Password Manager application on Flatpak. The Flatpak requested no filesystem access, while the Snap requested the `home` access permission. Within the application, an import functionality prompts the user to "browse for a CSV file," opening a file chooser dialog with only visibility within the Flatpak sandbox. While the application was functional, users would be required to manually move the files to the sandbox environment for import. Perhaps the most secure method, but using the application as intended is hampered.

Network Analysis: Network over-privilege was more common than filesystem over-privilege in both ecosystems, though Snaps were more likely to be overprivileged by a factor of over 5. To measure over-privilege, if we did not find an apparent need for network access in an app, we manually removed the network permission to ensure no functionality relied on network access.

Of the 37 instances of Snap over-privilege for network access, 15 came from KDE applications, all requesting the same permissions. We surmise this indicates one or more package maintainers for KDE apps used a one-size-fits-all approach to policy selection. Of our inconsistent apps, only Flatpaks failed to request needed network permission. An example of network underprivilege is the KBlocks games, where users can request new game themes. When requesting game themes, a Flatpak user will see an error indicating it could not reach a web server. Snap users can successfully request game themes, indicating the corresponding Flatpak is underprivileged.

In summary, we expected that over-privileged apps would be dominant, but we found an almost equal rate of under- and over-privilege across all apps. There was a clear trend that Snaps tend to be overprivileged, and Flatpaks tend to be underprivileged. In future work, researchers should investigate the root causes of this discrepancy and if it originated in differing community norms.

6 THREATS TO VALIDITY

Threats to Internal Validity: While we automated our processing and characterization of policy, our study had several manual aspects. First, three coauthors with extensive Linux experience reviewed and discussed our mapping of permissions. However, errors in the mapping could impact the resulting characterization and comparison. Second, assessing **RQ3** relied on manually testing applications, potentially allowing for a missed functionality within the application that required specific access permissions. We considered using static or dynamic analysis tools to determine application privilege needs. Unlike prior work on Android permission analysis, static analysis was impractical because Flatpaks and Snaps access resources in various ways and are written in multiple languages and runtimes. Dynamic analysis requires automatically creating test input generation and monitoring, which is an open challenge. Static or dynamic analysis needs customization per application, leaving manual analysis as our most reliable method.

When classifying under-privilege, we assumed that users use the functionality that is broken by the missing permission. In some cases, we classified applications as under-privileged even though there was technically a way to work around the lack of privilege (e.g., manually moving a file to a location). However, we decided that the application was underprivileged if the lack of privilege resulted in a significant negative impact on usability.

Threats to External Validity: While we downloaded all of the Flatpaks and Snaps from Flathub and the Snap Store, most of our analysis was performed on a much smaller set of matching applications. The analysis results for these matching applications may not extend to the broader ecosystem, particularly the IoT and server applications included in the Snap Store. We also only considered the Flathub repository for Flatpaks. While this is the *de facto* repository for Flatpak, there are other repositories, including those maintained by Fedora and Endless OS. Package maintainers for these other repositories may assign different sandbox policies.

7 IMPROVING ACCESS CONTROL

The additional security provided by the Flatpak and Snap sandboxes is highly dependent on the security policy defined by the package maintainer. This section reflects on different causes for weak sandbox policy and strategies for improvement.

Package Maintainer: The first cause for weak policy is the package maintainer. As shown in Finding 6, broad privilege dimensions for filesystem access or network access varied significantly for matching applications within Flatpak and Snap. This finding suggests that package maintainers are not always sure what policy to specify. We believe that package maintainers would be greatly aided by tools that automatically suggest initial fine-grained policies for applications. Some tooling does already exist. For example, Snap's `devmode` is similar to SELinux's `auditallow`, which logs policy denials rather than enforcing them. However, runtime testing has inherent limitations. Alternatively, static analysis tools can suggest policies based on methods called by the application. Creating such tools is nontrivial. Ideally, they should operate on package binaries rather than source code, as source code is not available for all applications. Applications are written in various languages and runtimes, and resources can be accessed multiple ways (e.g., Unix sockets vs. D-Bus), requiring data flow analysis.

App Developer: The second cause for weak policy is the app developer. When the package maintainer is not the app developer, they are limited in what policy can be specified without breaking app functionality. This intuition is supported by Finding 3, which shows that package maintainers use fine-grained permission when available, but coarse-grained permissions largely dominate file system access. We hypothesize that many applications could eliminate their need for file system permissions by adopting the XDG File Chooser portal API. However, this requires changes to the application. Future work should consider tools to automate this process.

App Framework: The third cause for weak policy is the app framework. Making changes to app frameworks often requires significant community coordination. For example, Electron is used by a number of popular Linux applications (e.g., Slack). The Electron GitHub issue tracker has a five-year discussion about using a desktop-aware file picker.[8] Ultimately, changing app frameworks to use more sandbox-friendly APIs will have a significant positive impact for Flatpaks and Snaps, though some app rewriting tools may still be required to port apps to the correct APIs.

System Architecture: The fourth cause for weak policy is system architecture. Some core system features (e.g., audio) were not built with fine-grained app permissions in mind. In the case of audio, the system architecture change from PulseAudio to Pipewire enables more least-privilege policies. As shown in Finding 4, once Snap released the `audio-playback` plug, package maintainers of applications started to transition. However, other system architecture features are still not compatible. For example, the Gnome BOXES Flatpak does not support USB device redirection[9] and interacting with SANE is a challenge[10] for Flatpak document scanners.

[8]https://github.com/electron/electron/issues/2911

8 RELATED WORK

Application sandboxing is a well-researched topic. Wahbe et al. [41] first introduced the concept of sandboxing in the context of software fault isolation. Other sandboxing methods include filtering system calls, e.g., Janus [21], Conch [3], Systrace [32], as well as other types of confinement [18, 20, 24, 31, 46]. Sandboxing using permissions or rules is common [2], ideally following the principle of least privilege [35] only to allow functionality needed by the application. Most Linux access control models use the Linux Security Modules [28] framework. Of note is AppArmor, designed initially to confine server applications [13, 27] which frequently have more predictable access control needs than end-user applications.

Android is the most widely studied ecosystem of application sandbox policy. Barrera et al. [6] performed the first broad analysis of Android permissions, using self-organizing maps to characterize 1,100 Android applications. Stowaway [14] first studied over-privilege in Android applications, finding one-third of their sample of 940 applications was over-privileged. Many subsequent tools were proposed to study over-privilege in Android [5, 7, 38, 43, 44]. Most recently, Droidtector [45] found 48% of sampled applications to be over-privileged. This over-privilege is likely enabled by end-users inability to comprehend the meaning of permissions [16].

Traditional Linux package managers such as `apt` and `yum` do not sandbox applications. Cappos et al. [11] discovered that by exploiting vulnerabilities in popular package managers, attackers with mirrors to distributions could crash hundreds to thousands of clients. Linux server applications are frequently distributed as Docker containers. Shu et al. [36] studied over 356,000 community-contributed images on Docker Hub, finding over 180 vulnerabilities and images that had not been updated for hundreds of days.

Finally, two recent academic studies have touched upon Flatpaks. Lauren et al. [25] reviewed various application-level sandboxing techniques, mainly oriented towards implementation mechanisms, restricting file system access, and graphical interface isolation. Their experiments consisted of 10 different sandboxing techniques and only briefly focused on Flatpak. More recently, Legay et al. [26] surveyed 170 Linux users, gauging their perception of package freshness. They found 21% of their participants used Flatpak or Snap to regularly install end-user open-source software, 22% for development tools, and 29% for programming language libraries.

9 CONCLUSION

Desktop operating systems are transitioning to the app-based security model that is commonplace in mobile platforms. For Linux, this transition coincides with distribution-independent methods of

[9]https://gitlab.gnome.org/GNOME/gnome-boxes/-/issues/236

[10]https://gitlab.gnome.org/GNOME/simple-scan/-/issues/21

distributing software: Flatpak and Snap. Linux desktop security has the potential to benefit significantly from this initiative, *if* packages have correct and secure policies. Our study of 283 applications that appear in both Flatpak and Snap ecosystems show that while the transition is already beneficial, there is room for greater improvement. The majority of Snaps (90.1%) and Flatpaks (58.3%) specify policies to prevent sandbox escapes, and package maintainers in most areas use fine-grained permissions. However, defining policy is still difficult and error-prone, though maintainers are showing positive progress in moving to finer-grained policies. In reporting these findings, we hope to inspire future research to continue to enhance the security of these ecosystems.

Acknowledgements: This work was supported in part by NSF grant CNS-1946273. Any findings and opinions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. 2018. Precise Android API Protection Mapping Derivation and Reasoning. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[2] Faisal Al Ameiri and Khaled Salah. 2011. Evaluation of popular application sandboxing. In *Proceedings of the International Conference for Internet Technology and Secured Transactions.* 358–362.

[3] A. Alexandrov, P. Kmiec, and K. Schauser. 1998. Consh: Confined Execution Environment for Internet Computations. (1998).

[4] J. P. Anderson. 1972. *Computer Security Technology Planning Study.* ESDTR-73-51. Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA. (Also available as Vol. I, DITCAD-758206. Vol. II DITCAD-772806)..

[5] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM conference on Computer and communications security.* 217–228.

[6] David Barrera, H. G unes Kayacik, Paul C. van Oorshot, and Anil Somayaji. 2010. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[7] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Automatically Securing Permission-based Software by Reducing the Attack Surface: An Application to Android. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE).* 274–277.

[8] Jon Brodkin. 2016. *Linux's RPM/deb split could be replaced by Flatpak vs. snap.* https://arstechnica.com/information-technology/2016/06/here-comes-flatpak-a-competitor-to-ubuntus-cross-platform-linux-apps/

[9] BS4 2021. *Beautiful Soup.* https://www.crummy.com/software/BeautifulSoup/

[10] Bubblewrap 2021. Bubblewrap. https://github.com/containers/bubblewrap.

[11] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. 2008. A look in the mirror: attacks on package managers. In *Proceedings of the ACM conference on Computer and Communications Security (CCS).*

[12] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. 2012. An Evaluation of the Google Chrome Extension Security Architecture. In *Proceedings of the USENIX Security Symposium.*

[13] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. 2000. SubDomain: Parsimonious Server Security. In *Proceedings of the USENIX conference on System administration (LISA)* (New Orleans, Louisiana). USENIX Association, 355–368.

[14] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[15] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdata Akhawe, and David Wagner. 2012. How to Ask for Permission. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec).*

[16] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android Permissions: User Attention, Comprehension and Behavior. In *Proceedings of the Symposium on Usable Privacy and Security.*

[17] Flatkill 2018. *Flatpak - a security nightmare.* https://flatkill.org/

[18] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. 2008. The Evolution of System-Call Monitoring. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC).*

[19] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. 2005. Automatic Placement of Authorization Hooks in the Linux Security Modules Framework. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[20] T. Garfinkel, B. Pfaff, and M. Rosenblum. 2004. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the ISOC Network and Distributed Systems Security Symposium (NDSS).*

[21] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. 1996. A secure environment for untrusted helper applications confining the Wily Hacker. In *Proceedings of the on USENIX Security Symposium.*

[22] Flatpak Issue. 2020. *Home permissions too relaxed and give full permission escalation #3637.* https://github.com/flatpak/flatpak/issues/3637

[23] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. 2004. Consistency Analysis of Authorization Hook Placement in the Linux Security Modules Framework. *Transactions on Information and System Security* 7, 2 (May 2004), 175–205.

[24] K. Jain and R. Sekar. 2000. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *Proceedings of the ISOC Network and Distributed Systems Security Symposium (NDSS).*

[25] Samuel Laurén, Sampsa Rauti, and Ville Leppänen. 2017. A Survey on Application Sandboxing Techniques. In *Proceedings of the International Conference on Computer Systems and Technologies.*

[26] Damien Legay, Alexandre Decan, and Tom Mens. 2020. On Package Freshness in Linux Distributions. *CoRR* abs/2007.16123 (2020). arXiv:2007.16123 https://arxiv.org/abs/2007.16123

[27] Bill McCarty. 2004. *SELinux: NSA's Open Source Security Enhanced Linux.* O'Reilly Media, Inc.

[28] James Morris, Stephen Smalley, and Greg Kroah-Hartman. 2002. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium.* ACM Berkeley, CA, 17–31.

[29] John Paul. 2016. Is Ubuntu's Snap Packaging Really Secure? https://itsfoss.com/snap-package-securrity-issue/

[30] Portal Documentation 2021. Portal Documentation. https://flatpak.github.io/xdg-desktop-portal/portal-docs.html.

[31] Vassilis Prevelakis and Diomidis Spinellis. 2001. Sandboxing Applications. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference.*

[32] Niels Provos. 2003. Improving host security with system call policies. In *Proceedings of the USENIX Security Symposium.*

[33] PulseAudio. 2016. Access Control. https://www.freedesktop.org/wiki/Software/PulseAudio/Documentation/Developer/AccessControl/.

[34] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. 2012. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P).*

[35] Jerry Saltzer and Mike Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* 63, 9 (Sept. 1975).

[36] Rui Shu, Xiaohui Gu, and William Enck. 2017. A Study of Security Vulnerabilities on Docker Hub. In *Proceedings of the ACM on Conference on Data and Application Security and Privacy (CODASPY).*

[37] Snapcraft. 2021. Supported Interfaces. https://snapcraft.io/docs/supported-interfaces.

[38] Vincent F. Taylor and Ivan Martinovic. 2016. SecuRank: Starving Permission-Hungry Apps Using Contextual Permission Analysis. In *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM).*

[39] Ubuntu 20.04 LTS Release Notes 2020. *FocalFossa/ReleaseNotes - Ubuntu Wiki.* https://wiki.ubuntu.com/FocalFossa/ReleaseNotes

[40] Steven J. Vaughan-Nichols. 2019. *The future of Linux desktop application delivery is Flatpak and Snap.* https://www.zdnet.com/article/the-future-of-linux-desktop-application-delivery-is-flatpak-and-snap/

[41] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles.*

[42] Jack Wallen. 2020. *Why snap and flatpak are so important to Linux.* https://www.techrepublic.com/article/why-snap-and-flatpak-are-so-important-to-linux/

[43] Yang Wang, Jun Zheng, Chen Sun, and Srinivas Mukkamala. 2013. Quantitative Security Risk Assessment of Android Permissions and Applications. In *Data and Applications Security and Privacy XXVII.* Springer.

[44] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. 2012. Permission evolution in the Android ecosystem. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC).*

[45] Sha Wu and Jiajia Liu. 2019. Overprivileged Permission Detection for Android Applications. In *Proceedings of the IEEE International Conference on Communications.*

[46] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy.*

[47] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. 2002. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the USENIX Security Symposium.*