# Application Sandboxing for Linux Desktops – A User-Friendly Approach (Author Copy)

Lukas Brodschelm[1], Marcus Gelderie[1]
[1]*Department of Electrical Engineering and Computer Science, Aalen University of Applied Sciences, Beethoven Str 1, Aalen, Germany*
{*Lukas Brodschelm, Marcus Gelderie*}*@hs-aalen.de*

Keywords: Sandbox,Linux,Desktop,User-Friendly,Usability,Security

Abstract: Sandboxes are a proven tool to isolate processes from the overall system. Although desktop computers face significant risks, there is no widely adopted way to use sandboxes on the Linux desktops, since sandboxing on desktop PCs is more challenging. We name the specific challenges for the Linux desktop and derive requirements that we argue are essential for widespread adoption of any sandbox solution. We then introduce a concept to isolate Linux desktop software using UIDs and GIDs as well as namespace-based sandboxes. Furthermore, we provide a PoC implementation including sandbox profiles for example applications. Based on this, we conducted a survey to assess the usability of our sandboxing concept. We report on the results, analyze the security of our concept, and detail how our sandbox meets the aforementioned requirements.

# 1 INTRODUCTION

The IT security of individual systems is an important and active research area. General solutions are difficult, since the needs of a system depend highly on the way it is being used. The desktop setting is particularly challenging and users make mistakes. As such, desktop PCs are a common entry point for malware into corporate networks (Waterson, 2020). On a desktop system, many different applications run within the context of a single user account: A single vulnerable program might gain access to all data and processes that belong to the account. This extends an attacker's sphere of influence considerably, once the initial compromise has been made.

On Windows, much has been done in the way of hardening the desktop setting against attacks. Mandatory Integrity Control (MIC), Control Flow Guard, and User Account Control (UAC) – to name only a few – (Yosifovic et al., 2017). Unlike Windows, the Linux desktop has to date not seen widespread adoption of a similarly comprehensive desktop hardening strategy. We hypothesize that, to be adopted a security feature must meet usability characteristics (cf. section 3.1).

Our contribution in this paper is a container based architecture, that separates individual apps users run on their desktop and respects usability aspects. We demonstrate the feasibility of this architecture using a PoC implementation and showcase how it addresses the issues given in section 3.1. Finally, we evaluate the usability of our solution with the target user group in the form of survey. Our approach can serve as the first step in an incremental strengthening of the standard Linux desktop.

**Related Work:** In related work, much research has been devoted to the development of sandboxes, specifically those based on containers. We group our treatment of related work into categories.

*Secure Operating Systems:* Secure operating systems that also enforce trust boundaries between applications, like Cubes, Whonix and SubgraphOS, have been developed (The Qubes OS Project and others, 2022; ENCRYPTED SUPPORT LP, 2022; Subgraph, 2014). They all use launchers to run applications in there own isolated environment, enforced either by virtual machines or container based sandboxes. In contrast our sandbox provides restricted, but direct access to the host file system and user's home directory.

The Android platform security model (Mayrhofer et al., 2021; Android Open Source Project, 2021) follows a different approach where applications are aware of the sandbox and need to use specific APIs to request access to resources. Android meets our usability requirments but requires cooperation from applications. Our solution borrows from the Android model in that we also run applications with dedicated

user IDs. However we do not require cooperation from the sandboxed applications.

*Containers and Container Based Sandboxes:* Many implementations of sandboxes and containers exist. Depending on the exact definition of a sandbox, any container runtime, like `runc` (Open Container Initiative , 2022) or Firecracker (Agache et al., 2020) can be considered a sandbox. However, typically we mean designs specifically targeting security by this term. Examples for such systems are Bubblewrap and Firejail (Containers Community, 2016; Firejail Contributers, 2020). They do not cater to any specific use-case and instead of enforcing access models the goal of these sandboxes is to isolate one given application. They operate at a lower level than what we try to achieve with the proposed desktop sandboxing scheme. Indeed, our design uses Bubblewrap as a building block. In fact, our sandboxing scheme can be thought of a way to automatically configure Bubblewrap in a desktop session.

In scientific research container based sandboxes have studied extensively (Bélair et al., 2019; Khalimov et al., 2019; Anjali et al., 2020; Agache et al., 2020). The aim of the research is diverse and covers aspects like kernel interactions, shielding the host from the workloads and deployment aspects. But in all these works, the focus is not on desktop systems. An exception are container based sandboxes for new application domains, such as malware analysis (Khalimov et al., 2019). Here, the focus is on running suspected-malicious binaries, where near total isolation, erring on the side of caution, is paramount. A multipurpose concept is TxBox, it relies on monitoring and reverting changes to the kernel state through system calls from sandboxed applications (Jana et al., 2011). TxBox requires a modified kernel, which is difficult to align with our goal of supporting arbitrary Linux distributions (section 3.2).

Software distribution and deployment also leverages container sandboxes. Examples are Flatpak and Snapd (Flatpak Team, 2018; Canonical Ltd., 2022). Both use container technologies to run the shipped applications within a dedicated file system. When configured right, they are secure but different to our approach they ship entrie or partial root-file-systems for the applications and they are bound to a package manager. Furthermore administrative privileges are required to modify the configuration, which is contrary to our goals.

# 2 PRELIMINARIES

Linux Namespaces (NS) are kernel features providing isolated abstraction layers for system resources (Linux Manpage Team, 2021). There are eight types of NS, but we only mention user namespaces (user NS) in this paper, which provide isolation of UIDs, GIDs, and other security-related attributes such as capabilities. Our treatment of namespaces glosses over several details, but is sufficient to follow the remainder of this paper.

When a process with UID $u$ creates a user NS, that NS is *owned* by $u$. This means that any process with UID $u$ holds all capabilities inside the NS. Once inside the NS, such a process holds `CAP_SETUID` can alter its UID to any one of the UIDs *mapped* inside the NS. The *UID mapping* is a translation of UIDs inside to UIDs outside (*host UIDs*). In the simplest case, it translates UID 0 to UID $u$ and no other UIDs exist inside the NS. This is called a *trivial map*. For a non-trivial map to be defined, a privileged process outside must define the map. The setuid binary `newuidmap` can be used to do this as an unprivileged user while restricting the user to a permitted range of host UIDs.

On the Linux desktop, applications are typically launched via a menu item or a launcher tool (such as `rofi`, `dmenu` or the like). All these tools support using *desktop entries* to launch applications. Desktop entries are configuration files that provide information on how to start applications (CLI arguments, environment variables etc). Most desktop environments support desktop entries (Brown et al., 2020) and provide them for most graphical user interface software. It is possible to shadow system-wide desktop entries by providing a file of the same name inside a location in the user's home directory (usually `~/.local/share/applications`).

Access control systems enforce trust boundaries and defines rules for data flow between them. There are many different ways of approaching access control models known from standard literature (Anderson, 2020). Two that have traditionally been very popular are the *hierarchical model*, which dates back to (Bell, 1975; Biba, 1975) and is the basis of Microsoft's "Mandatory Integrity Control". Another is the *type enforcement (TE)* model (Badger et al., 1996), which is successfully deployed on many systems in the form of SELinux.

In the hierarchical access control model resources and applications are assigned levels from "untrusted" to "trusted" and only processes in levels that are at least as high as the one of a resources may access them. In the type enforcement mode, one instead assigns *types* to resources and *domains* to processes. A

policy then specifies, which types may be accessed by a domain, and how one domain may *transition* into another domain (such as by executing another program).

A type may contain many resources but a resource is associated with exactly one type. A domain may access many types – a one to many association. The communication between applications, and which application is allowed to launch other applications, is defined by a domain to domain relation.

# 3 SANDBOXING ON DESKTOP SYSTEMS

Sandboxes can be used in many different scenarios and as such will serve different needs in different deployment scenarios. We focus on desktop use cases, where one or many users interact with a personal computer or laptop via a graphical user interface. In these situations, it is difficult to predict the software that is installed or the precise configuration in which the system is used. While true on desktop systems in general, this observation is particularly relevant to the GNU/Linux ecosystem with its variety of distributions and desktop environments.

## 3.1 Observations

We highlight the main aspects of the Linux desktop (or desktop systems in general) that are, in our view, relevant to designing a user-friendly, yet reasonably secure sandboxing solution for the desktop:

**Choice of Software:** Linux users expect to be able to install software of their choice and work with that software. This is true for the Linux distribution it self, but also for installed software.

**Unknown Workflow:** Users are unlikely to accept security mechanisms that require them to change their workflow, but users work in different ways. This manifests in different folder layouts, different interactions between tools, and so forth.

**Out of the box Functionality:** Most users expect, that software on their system works out of the box in most cases. They usually do not want to make manual configurations to obtain basic functionality. This is true, in particular, for security measures.

**Multi-User Systems:** Desktop systems are occasionally used by multiple users (such as in an office setting or at home). A sandboxing solution must support this use-case and uphold the separation between accounts.

**Rootless:** Sometimes, users do not have administrative rights on their machine (e.g. when computers are part of a laboratory setup or when they are centrally managed by an organization's IT department).

## 3.2 Requirements

From these observations, we derive the following usability requirements that a sandbox must meet in order to be accepted by the user. The sandbox must be:

R1 Able to support any application in principle.

R2 Able to Co-exist and even interact with unsandboxed applications.

R3 Amendable to support new applications and self-developed software in particular.

R4 Provided as an additional application, that can be installed using the package manager of any standard Linux distributions.

R5 Customizable to accommodate user's preferred workflow.

R6 Configurable. This must be easy and not require administrative privileges.

R7 Provided with predefined profiles for most common applications.

R8 Able to support multi-user environments.

R9 Transparent as possible to users and once configured not require their attention.

In addition to the requirements above, the software must be secure in the following threat model:

**Threat Model.** We assume that an adversary has full control of a process $P$ on the desktop system. This means that the adversary can execute arbitrary code within the context of that process, read all of that process's memory and access any system resource that process can access.

Note that in the traditional Linux security model, if an adversary has full control over a process running as some user, the adversary has full control over the entire desktop session of that user: All files are accessible, and all processes can be signaled. Permission to trace processes is typically restricted by the Yama LSM, but this need not be the case and usually permits tracing of descendent processes. We aim to strengthen security in that the adversary is restricted to only those parts of the session that the benign code within that process would have legitimately had access to.

# 4 SANDBOXING CONCEPT

In this section we outline the access control model we use for our proposed sandboxing solution and de-

scribe how we implement that model for any recent Linux kernel (i.e. one that supports required functions for unprivileged user NS).

Note that these models are usually seen in the context of mandatory access control, but we are not building a MAC system in the traditional sense. Instead, we devise a policy that is built on these models and which can be enforced using discretionary access control in conjunction with standard Linux sandboxing software.

An issue with poorly protected, application-specific resources (e.g. config-files), which exists for the hierarchical access control model, can be solved in the TE model by introducing an application-specific type for every application and granting only the corresponding domain access to it. Futher, the TE model promises to be more suitable for our purposes and we choose to base our sandbox policy on it.

The TE model knows domains and types, associated with processes and resources, respectively. Those model concepts must be represented in practice by using the features provided by an unpatched Linux kernel. In the following, we outline how TE can be realized using namespaces to assign UIDs to domains and GIDs to types. We also detail how a container based sandbox can enforce a policy defined in this way. The resulting model is visualized in fig. 1.

On common Linux desktops all processes of an account run with the same privileges. But to implement TE, it is necessary to enforce trust boundaries between them. Therefore we execute every application with its own unique UID and thereby think of UIDs as domains.

The concept of a type labels resources (files, in particular) is characterised by a many to many relation to domains and by a many to one relation to resources. Since on Unix, famously, everything is a file, we may think of resources as files (so file does *not* mean "regular file" in this paper, but may be a block device, socket etc.). As a consequence GIDs are a suitable representation of types. A file has exactly one GID but many files can have the same GID, and a processes can be a member of several GIDs. Membership of a process in a group can be thought of as a domain having access to a type.

On Linux the association between UID and GIDs is usually part of the account. However, creating an account for every required UID is not appropriate in our case, since creating or modifying accounts requires elevated privileges (cf. requirement R6). Instead of using accounts we define the relations between UID and GIDs in a user-editable configuration file. Besides the UID to GIDs relation, the mapping of executables to UIDs, as well as the relation between GIDs and paths is defined in this file.

Defining path permissions in a configuration file rather than on the directory or file itself is clearer. Unfortunately, it is not possible to apply those permissions to common Linux file systems in real-time without intercepting file system access or conflicting with the classic Linux file system and its access control. Therefore a software is required, which corrects the file system permissions on demand. In the PoC such a script is provided, it traverses the directories recursive and corrects files UID / GID and permissions. Further it it applies the SETGID bit, and an access control list rule to directories, to change the default file owner and mode declarations. This is not a security but a usability feature.

Until this point not required is the file owner, this is used to ensure that unsandboxed processes continue to have access to user files in the usual way. The file owner property is set to the UID of the user account. In a sense, processes running with this UID are more privileged than the other UIDs in the system. This is done to support requirement R2.

On Linux, processes require `CAP_SETUID`/ `CAP_SETGID` to change UID and GIDs. Assigning these capabilities to any process in the user's session would mean that we weaken security compared to the classical Linux desktop setup. Instead, we use bubblewrap to run applications in user NS and create UID/GID mappings, using the setuid application `newuidmap`, to implement a container based sandbox that does not require elevated privileges to be executed. Note that the user account's UID owns the new user NS and has full capabilities inside. As a result, it can assume any UID/GID inside, as long as it is mapped. This gives another reason to think of the user account's UID as privileged. A side effect of using bubblewrap and user NS is that we need to define bind mounts, since a mount NS is always unshared. Our PoC does not use the mount NS for isolation and simply provides the entire root file system to the sandbox.

Besides implementing enforcement of domain-type relations, we also must specify domain-domain relations, which dictate which other domains a given domain might start. On Linux there is no suitable relation between UIDs that governs this kind of access: Setuid binaries allow transitioning to other domains, but they are set globally and affect every account and they will also not launch a new user NS.

We chose to enforce this relation using a user-space service, which runs with the permissions of the user's primary UID. We assume the administrator defined a UID range for this user in `/etc/subuid` (and likewise for GIDs) when creating the account. Then

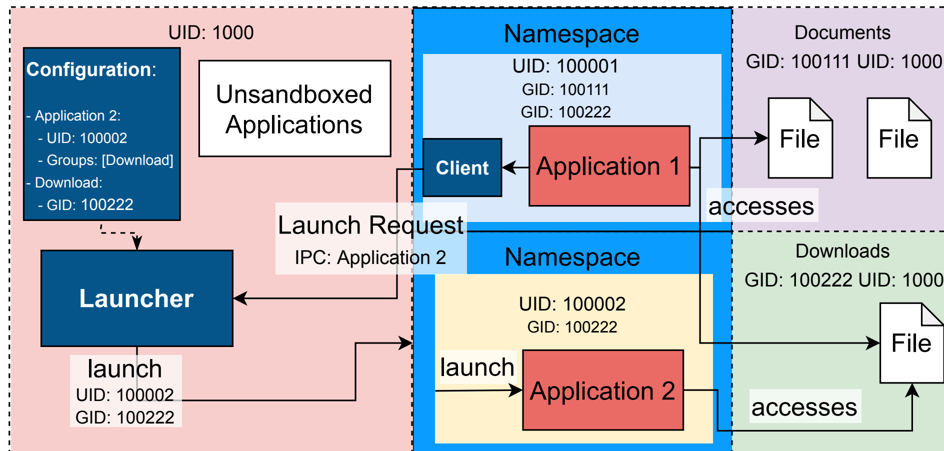


Figure 1: Sandbox Design

our service can define UID mappings via `newuidmap` and run applications in their correct domains. A process that wants to start another process then launches a client binary, which communicates with the service via an D-Bus IPC interface and requests the launch. To enable this the D-Bus session bus is available to all sandboxed applications. Upon receiving a launch request, the service will look up the permissions of the requesting UID in a configuration file and decide whether or not to launch the requested application with its configured privileges. It is important that no other UID than the user's primary one is capable of changing the configuration.

In order to launch an application after validation, the service runs a helper script in the namespaces using bubblewrap. After the namespaces are unshared the execution is blocked by bubblewrap, and configures the service uses `newuidmap` to setup UID and GID mappings. Once the mappings are setup the helper script in the sandbox change its UID and GIDs to the configured ones and executes the target application.

When using UIDs and GIDs to identify types and domains, the UID/GID range assigned to a user's account must be carefully chosen. For distinct users, these ranges must be disjoint. Moreover, no two applications must share the same host UID. To ensure this the presented PoC assigns a global UID to every domain $d$, and a global GID to every type $t$. The actual ID $u$ is computed by combining the global ID, called the *offset*, with a user specific *base UID* to avoid ID collisions: $u = \mathsf{base} + \mathsf{offset}(d)$

A similar equation holds for the GIDs $g$ corresponding to types. This method has the advantage that data can be transferred between systems as long as the user base is identical. A typical value for the user base would be 100000, which means that the UID/GID range $[100000, 100999]$ might be a good UID/GID range for this user.

In order to add an application to the sandbox system an user might add an entry containing its permissions and an unused offset by hand or install an application profile. The profile may be provided by a package maintainer and consists of the applications permissions (using file paths and global offsets). Whit the installer, that is part of our PoC it can be merged with the current config, but on conflict no user configuration is over written.

# 5 EVALUATION

To evaluate the requirements from section 3.2, we conducted a usability survey on the prototype and provide an analysis of the access control model and our prototype.

## 5.1 Theoretical Analysis

All requirements except requirement R9 and parts of requirement R6 may be verified by analysing the access control model and the PoC implementation.

**Requirement R1: Support any application** Our sandbox does not expect applications' cooperation, but can interfere with the way it expects to access files, IPC, or other processes. If an application requires access to a resource, that access can be granted by adjusting group membership appropriately. This is obvious for files but for IPC the situation is more complex. For example during development, some work was required to enable sound of sandboxed applications via *PulseAudio*. We allow D-Bus communica-

tion, but other IPC can be challenging, we believe that it affects very few applications.

There is a large group of tools that are challenging to sandbox in our model: Command line tools and terminal emulators. For command-line tools it is difficult to wrap them because they are not launched via desktop files. For terminal emulators, it is possible to provide a profile, but it is very difficult to restrict access in a meaningful way.

**Requirement R2: Co-exist with unsandboxed applications** Our sandbox does not conflict with unsandboxed applications, since the sandbox access model maintains file ownership of the primary UID of the user and only group ownership is changed for access control. Unsandboxed applications have full access to all of user-session's resources, just as they do on traditional Linux desktop environments.

**Requirement R3: Support new applications** Given our analysis of requirement R1 it only remains to discuss that adding new software will not pose logistical issues arising from collisions in the UIDs assigned to individual applications.

We first note that our sandbox assigns global unique UIDs but running out of UIDs seems unlikely. If users install software outside the package manager, there is a risk that the UID $u$ manually chosen for such an app will collide with a packaged app that is later installed: $u = \mathsf{UserBase} + \mathsf{Offset}(\text{Firefox})$, resulting in a security breach. This can be avoided, by specifying a range of private offsets (much like private IPv4 network ranges) that is forbidden to be used by packaged software.

**Requirement R4: Installable as additional package** The prototype can be packaged like any other application and installed on standard Linux distributions. Our PoC comes with a Debian package that can be used to install it on systems using `dpkg`. However, it does not depend on any distribution-specific content – only Python 3, Bubblewrap, `subuidmap`, and D-Bus. Additionally, the way we use Bubblwrap in conjunction with NS requires support for unprivileged user NS. However, unprivileged unsharing of user NS is enabled on recent kernels in popular distributions today (e.g. Debian, Ubuntu, Arch Linux).

**Requirement R5: Customizable** Our sandbox is customizable via its configuration files. The labeling of files and directories can be changed according to the user's preference. Files and directories can then be relabeled automatically using the provided tools. Customization includes both the ability to change a file or directory's type, as well as the types of any application. Moreover, those changes are not undone on update. There is some overhead if the package maintainer alters the applications initial configuration file and those changes are ported to the user's local copy.

**Requirements R6 and R8: Multi-user and easy customization** The configuration of the sandbox is defined for every user on a system in a user-specific, human readable configuration file that is owned by the primary UID of the user's account. Users may edit this configuration file without root privileges to make smaller modifications or change the entire access model. As mentioned, the question whether configuration is "easy" is best answered using an survey and this is done further below.

The precondition for the way we launch applications with UIDs different from the user's own primary UID is that the user may set up UID mappings in the new user NS. Since we use the `newuidmap` family of tools for this purpose, a valid range for the user needs to be written to `/etc/subuid`. This is the only configuration step that requires administrative privileges, but can be done when the user's account is created. If those ranges are set up without overlap, the sandbox may be used for multiple users without risk of weakening the isolation between those accounts.

**Requirement R7: Sandbox must provide profiles for standard applications** While our PoC cannot provide profiles for every "common" application for obvious reasons, we demonstrated that providing such a profile is a simple task for any user with at least moderate knowledge: A new permission file need only describe the application offset (cf. section 4) and required group memberships. The PoC already includes a number of groups for common tasks, e.g. office applications. Those files can be merged using the installer provided with the PoC.

## 5.2 Survey

We conducted a survey to investigate whether requirements R6 and R9 are fulfilled: if the sandbox, once configured, is transparent to the user and whether it is easy to customize. The test environment was provided in the form of a virtual machine configured with the PoC. Participants are required to perform certain tasks on the system, and answer questions about their interaction with the sandboxed applications. The survey was conducted with students in computer science or electrical engineering at a college of higher educa-

tion in Germany. The questionnaire and manual were provided in German.

The questionnaire consists of pairs of a task to be performed, followed by questions about the task. With every task description, a short sandbox manual is provided. Furthermore the questionnaire contains free text fields, so that subjects can provide additional comments. The tasks the subjects had to perform are:

1. Starting the Firefox web browser and visiting an arbitrary website.
2. Downloading a PDF file with Firefox, opening it with the Okular PDF viewer and saving it in folder "Documents".
3. Creating a text file in "Documents" with the text editor Geany and reopening it with VSCode.
4. Adding a new folder to the users home directory and grant access to it to both Geany and VSCode.

**Results:** 22 students took part in the survey, but not every question was answered by every student.

During the first task, we expect subjects not to notice the sandbox. 95% of the test users respond that they would use the sandbox for this scenario. All subjects did not notice the sandbox.

For the second task, 60% of the users had to interact with the sandbox, which was expected due to Firefox internals. 42.66% of the users that had to interact with the sandbox reported difficulties but 76.19% of all users would use the sandbox for this scenario.

The third task was not expected to lead to any issues. Although 40% of the users experienced unexpected errors at this point 68.42 % of the subjects responded they would use the software for this scenario. We can trace it back to permission issues of VSCode with access to tmpfs and a bug in our PoC.

With the fourth task, we checked if subjects are able to customize the policies. 68.4% percent of the subjects claimed to succeed. Since 57.9% of the subjects had problems with the documentation, more comprehensive documentation is desirable.

Evaluation of the free-text comments showed that several subjects were confused about the purpose of the survey and did not necessarily evaluate the usability, but rather tried to understand the security benefits of the sandbox. Those subjects reported that they were not ready to use the sandbox in the future, because they were not convinced of its benefits. This is an interesting criticism, but since we did not attempt to explain those benefits to subjects, it is also not very surprising. Future evaluations should be constructed to avoid this misunderstanding by communicating the intent of the survey more clearly.

## 5.3 Security Evaluation

We begin by noting that, since the UID ranges assigned to distinct users do not overlap, our sandbox will not be less secure than the traditional Linux desktop with respect to file permissions. If access control checks based on UIDs in IPC frameworks are deactivated, this may weaken the security compared with the classical desktop setting. However, if access to the IPC interface can otherwise be restricted (e.g. through file permissions on `AF_UNIX` sockets), this problem is mitigated. A generic solution based on a proxy is possible, but has not been implemented in our PoC. Nevertheless, the security of a system running our sandbox can be considered at least as high as that of a standard Linux desktop.

If an attacker *A* controls a process *P*, our sandbox ensures that *A* cannot access any file that *P* could not already have accessed. Recall that file access includes IPC access where the IPC interface is represented as a file (e.g. an `AF_UNIX` socket). This is also true for child-processes launched by *P*. It is possible for *A* to launch applications in other sandboxes through their desktop file by invoking the launcher via its D-Bus interface. Again, this does not enable *A* to do anything that was not explicitly permitted for *P*. There are still two primary sources for security breaches in this scenario: First, *A* has access to a resource, such as IPC, that can be used to influence *other* processes. Second, *A* may exploit programming errors (e.g. command injection) in other applications. As dangerous as this situation is, we consider it out of scope, because there is nothing the sandbox can do to mitigate it.

Xorg is a well-known display server used to display the UIs of applications. Applications use it via a socket that is widely accessible. Xorg does not support the concept of trust boundaries or window ownership on a single Xorg desktop, which means that applications can access other X applications. To deal with this either the Xorg successor Wayland or a multi-instance desktop implementation like Xpra might be used (Antoine Martin and others, 2021). We designed a concept, where every sandboxed application starts its own Xpra instance and a client under the user's primary UID connects to it. This is not currently part of the PoC due to time constraints.

Finally, D-Bus access is a potential issue. If clients may access arbitrary D-Bus Interfaces, this may allow them to access sensitive material. It is possible to add D-Bus configuration templates as part of an application's profile, which can be used to restrict that applications access to the session bus. Again, this is not implemented within our PoC due to time constraints.

# 6 CONCLUSION AND FUTURE WORK

In this paper we introduced a method, how to use UIDs, GIDs, and user NS to create a sandbox for Linux desktops. The proposed sandbox is designed to meet requirements that we consider to be important for wide spread adoption. We implemented a prototype and performed a usability survey. The results indicate that an easy to use, transparent sandbox will likely be adopted, provided users understand the benefits of using the software. Furthermore provided an analysis of how the sandbox addresses each of the requirements mentioned above, and analyzed its security impact on the overall system.

Our research indicates several areas that future research should address. First of all a long term evaluation should be conducted to obtain results about the applications stability. As mentioned above, the current prototype does not support access control for the D-Bus session bus. A solution to restrict this access, is a necessary in our opinion. Another challenge for future work is that Xorg does not separate the graphical user interfaces of the applications. Therefore, either a multi instance display server like Xpra or a Wayland-based solution should be added. Third, currently network access is unrestricted. Isolating network access through network namespaces should be considered. The challenge here is to strike a balance between full and no access – many applications use localhost communication extensively.

# REFERENCES


Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. (2020). Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA. USENIX Association.

Anderson, R. (2020). *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons.

Android Open Source Project (2021). Android compatibility definition document. https://source.android.com/compatibility/cdd.

Anjali, Caraza-Harter, T., and Swift, M. M. (2020). Blending containers and virtual machines: A study of firecracker and gvisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 101–113, New York, NY, USA. Association for Computing Machinery.

Antoine Martin and others (2021). Xpra readme. https://github.com/Xpra-org/xpra/blob/master/README.md.

Badger, L., Sterne, D. F., Sherman, D. L., Walker, K. M., and Haghighat, S. A. (1996). A domain and type enforcement unix prototype. *Computing Systems*, 9(1):47–83.

Bélair, M., Laniepce, S., and Menaud, J.-M. (2019). Leveraging kernel security mechanisms to improve container security: A survey. In *Proc. of the 14th Int. Conf. on Availability, Reliability and Security*. ACM.

Bell, D. E. (1975). Secure computer systems: Mathematical foundations and model. *Mitre Corp. Report*, pages 74–244.

Biba, K. (1975). Integrity considerations for secure computing systems. *Mitre Report MTR-3153, Mitre Corporation, Bedford, MA*.

Brown, P., Blandford, J., Taylor, O., Untz, V., Bastian, W., Lortie, A., Faure, D., and Thompson, W. (2020). *Desktop Entry Specification*.

Canonical Ltd. (2022). Snap documentation — snapcraft documentation. https://snapcraft.io/docs.

Containers Community (2016). Bubblewrap source code. https://github.com/containers/bubblewrap.

ENCRYPTED SUPPORT LP (2022). Whonix. https://www.whonix.org/.

Firejail Contributers (2020). Firejail source code. https://github.com/netblue30/firejail.

Flatpak Team (2018). Flatpak's documentation. https://docs.flatpak.org/en/latest/#.

Jana, S., Porter, D. E., and Shmatikov, V. (2011). TxBox: Building secure, efficient sandboxes with system transactions. In *2011 IEEE Symposium on Security and Privacy*. IEEE.

Khalimov, A., Benahmed, S., Hussain, R., Kazmi, S. A., Oracevic, A., Hussain, F., Ahmad, F., and Kerrache, C. A. (2019). Container-based sandboxes for malware analysis: A compromise worth considering. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, UCC'19, page 219–227, New York, NY, USA. Association for Computing Machinery.

Linux Manpage Team (2021). *Linux manual page*.

Mayrhofer, R., Stoep, J. V., Brubaker, C., and Kralevich, N. (2021). The android platform security model. *ACM Transactions on Privacy and Security*, 24(3):1–35.

Open Container Initiative (2022). Runc source code. https://github.com/opencontainers/runc.

Subgraph (2014). Subgraph os. https://subgraph.com/.

The Qubes OS Project and others (2022). Architecture — qubes os. https://www.qubes-os.org/doc/architecture/.

Waterson, D. (2020). Managing endpoints, the weakest link in the security chain. *Network Security*, 2020(8):9–13.

Yosifovic, P., Ionescu, A., Russinovich, M. E., and Solomon, D. A. (2017). *Windows Internals Seventh Edition Part 1: System architecture, processes, threads, memory management, and more, Seventh Edition*. O'Reilly.