UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

INFORMATION ENGINEERING BACHELOR DEGREE

"Executing and remapping GNU/Linux applications on the Android platform"

Supervisor: Prof. Carlo Fantozzi PhD

Student: Luca Fabbian

ACADEMIC YEAR 2020– 2021.
Graduation date 24 September 2021

Acknowledgements

I would like to thank my parents, who never failed to support and love me throughout the past months and the previous 21 years. Thank you for always being by my side and for helping me out when I was struggling.

I also want to express my gratitude to my university colleagues for the fun times and engaging conversations. In particular, I would like to thank Filippo Bragato, Paolo Bresolin, Filippo Simonetti, and Enrico Tosi, with whom I spent my last in-person classes, and my roommates and friends Gianluigi Galeotafiore and Diego Antignati.

I want to thank my high school mates Giosuè Sardo, Maxim Kovalkov, Antonio Napolitano, Filippo Casarin, Andrea Zigiotti, Lorenzo Botter, Gabriele Saracco, Matteo Secco, and the rest of the group with whom I participated in the "Zerorobotics" competition. This experience reignited my passion for computer science at a time when I was uncertain about my career path.

Furthermore, I would like to thank Dr. Emanuele Rocco, Dr. Andrea Saiani, undergraduate student Simone Nascivera, and other members of the Witted lab staff who supported me in multiple internship experiences. They played a crucial role in my personal and professional development, and I will always be grateful for their valuable advice and help, even outside of work.

Lastly, I would like to thank Professor Riccardo Colpi for helping me with my enrollment in Information Engineering and Professor Carlo Fantozzi for his constant dedication as my thesis advisor, following me closely and always demonstrating his competence, reliability, and punctuality.

Table of contents

Abstract

Many modern Android devices come with desktop-like computational power. Yet, you can't easily run on Android some software available on GNU/Linux operating systems, such as LibreOffice, Gcc, Python or Java.

After providing a comparison between Android and GNU/Linux OSes, spanning from kernel to high level libraries, this thesis will provide an overview of available solutions. I will consider apps such as IBochs, Limbo, Termux, AnLinux, Andronix e Debian noroot, and highlight issues with the state of the art. I will then develop a new, different approach based on web apps which could interact with a local server: through a powerful API, they will be able to talk to the Android system library and to access binary files of an Alpine Linux virtual environment.

To prove my point, I will develop a demo, named Unyw (https://github.com/unyw). This Android app will let users install and execute web apps wrapping GNU/Linux binaries with various degrees of abstraction. I will use the prototype to highlight strengths and weaknesses of the approach. While some application will be kept with little overhead and delivered with the same GUI, others will be completely remapped using web technologies (HTML/CSS/JavaScript/Svelte), all without compiling a single line of code.

To complete my project, I will develop some sample web apps, including those that allow interfacing with the package manager of Alpine Linux, the XTerm terminal emulator, the Chromium browser, and the file conversion function of LibreOffice. Additionally, I will provide libraries and scripts to facilitate the writing and distribution of other wrappers.

I have found that Unyw, the Android app I wrote, is excellent in terms of disk size (approximately 86Mb at the end of installation, a better result than all other solutions examined) and for the performance achieved (comparable or superior to all other solutions examined). It is also extremely easy to expand with both command-line programs and new screens with their own graphical interfaces. In conclusion, I will list the changes needed to make the prototype ready for distribution.

1. Introduction

Today's smartphones are getting computer-like capabilities. There even exists some smartphones with resources closer to the one of medium-high range desktops.

Since there is no hardware constraint, you may expect mobile devices to be able to run desktop apps. Android devices, in particular, are based on the same kernel of GNU/Linux desktop operating systems, so they should have little or no problem running desktop binaries, including:

- all GNU utilities, including the Gcc compiler, the Bash shell, and command line tools such as Gzip;
- Java Virtual Machine and Java executables;
- GUI based app such as LibreOffice, Firefox, Chromium, and Audacity.

Only few of the apps above have an Android porting, and usually is just a partial porting which lacks some key features: for example, Firefox for Android does not ship with developer tools. Thus, we are interested in having a general method for running the PC version of these applications. However, while there are good ways to run Android apps on desktops and they have been available for years (such as Anbox **[48]**), the inverse problem has fewer solutions and there are still open challenges.

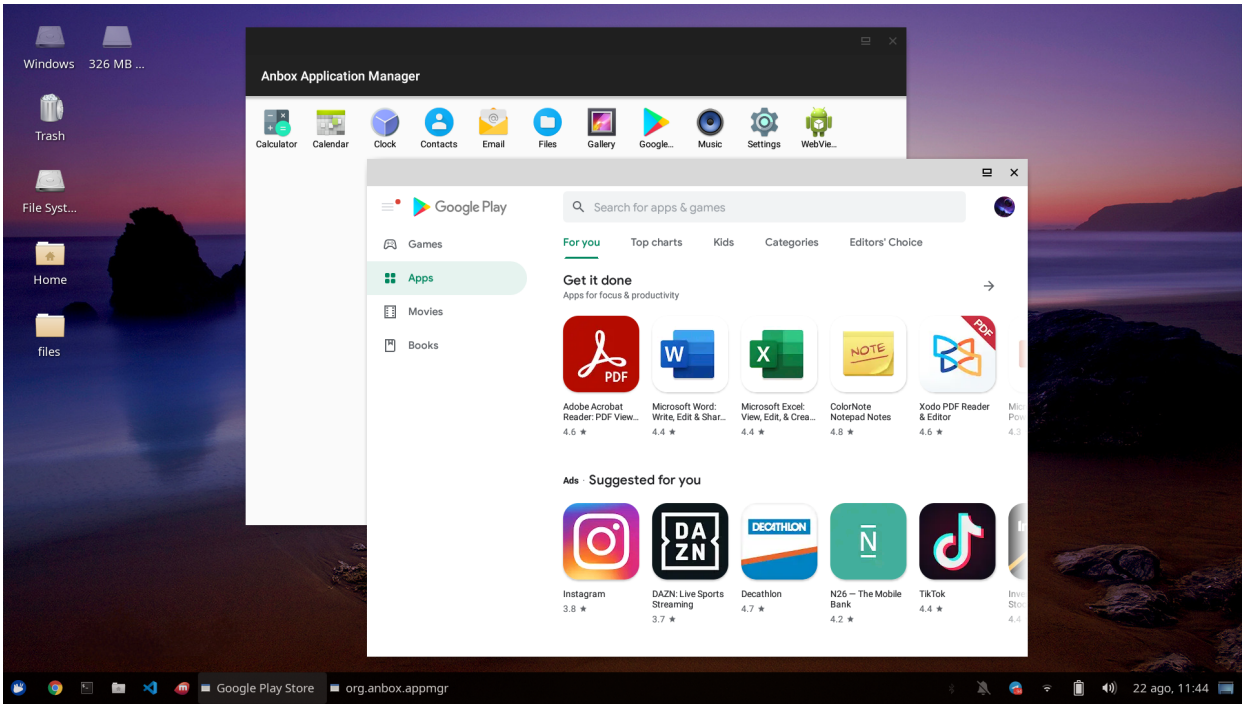

Figure 1: *Screenshot of Anbox seamlessly integrated into XUbuntu 20.04*

Android, in fact, has many architectural choices that are incompatible with GNU/Linux systems and imposes numerous security limitations that make it more difficult to design an emulator and integrate it properly with the system. Therefore, almost all existing solutions are limited to running a GNU/Linux system as it is, without offering any kind of enhanced control for mobile devices. Since mobile phones lack a keyboard and have a small screen, this makes some programs unusable. After analysing the issue, this thesis will therefore propose a new approach that can solve many of the issues encountered.

1.1 Distro and metrics considered

Since both Android and GNU/Linux are open-source at their core, there are numerous variants, usually called "distros," of these operating systems. The DistroWatch website counts more than 500 variants of GNU/Linux operating systems. Some distros are very similar to each other, while others have unusual characteristics and are in opposition to those of the most popular distros. It would be pointless and counterproductive to discuss all existing distros for the purposes of this thesis, as it would introduce numerous problems that do not arise with the vast majority of GNU/Linux software.

Therefore, unless said otherwise, I will only discuss the most popular GNU/Linux systems, such as Ubuntu **[49]**, Debian **[50]**, Arch Linux **[51]**, and Alpine Linux **[52]**, as representatives. Sometimes I will also mention Tinycore Linux **[53]** as an example of a "live" distro (i.e., one that does not need to be installed because it can work entirely from RAM) that can offer a graphical environment in just 12 Mb. For Android devices, I will consider only the features present on a generic smartphone and available to the user without tampering the device by unlocking the bootloader or root user.

Small changes in runtimes and performance are of no interest, as the performance of an executable is practically influenced by numerous factors, in most cases, thus I will focus more on the features present in the analysed solutions. However, in some cases, it will be necessary to argue the extreme inefficiency of some solutions compared to others; to do this, I will perform benchmarks using a real device. Since mobile devices often have limited memory, I will compare solutions based on the disk space they require.

1.2 Thesis topics and structure

In this thesis I will:

- Examine the incompatibilities between Android and GNU/Linux systems, from low-level to high-level issues, and suggest strategies to solve them (Chapter 2).
- Compare existing solutions, grouping them in hypervisor-based and chroot-based emulation mechanisms. I will list pros and cons for each solution, and conduct performance tests to measure their effectiveness (Chapter 3).
- Propose my own solution to the problem, a new approach based on previous considerations: an Android app that allows running web technology-based applications with a powerful API to integrate with both the Android system and an Alpine Linux system executed through chroot mechanisms. I will describe the app from a theoretical point of view, and provide a reference implementation to prove my point (Chapter 4).
- Analysing the obtained results. I will conclude with a comparison with the solutions mentioned in Chapter 3, and list some future improvements to my solution (Chapter 5).

2. Incompatibilities between Android and GNU/Linux OSes

2.1 Kernel

The Android and GNU/Linux systems share the same kernel base. However, the Android kernel is patched **[1]** with device-specific changes that aim to add features such as:

- General improvements and features developed in advance by the Android team. Typically, these additions become standard in the main Linux kernel within a few years. For example, this has happened with EAS **[2]** (Energy Aware System, a module that helps with energy saving), some networking modules, and some file system management modules like F2FS **[3]** (a file system developed by Samsung as a competitor to ext4).
- Features specifically designed for Android, such as a logging management module, Android Shared Memory, and a Wakelocks and alarm system.
- Third-party drivers, which are often included as pre-compiled binary bloatware and therefore are closed-source.

Many of these features are so low-level that only the operating system or kernel itself uses them. Therefore, they do not cause any incompatibility issues with applications, which actually benefit from them implicitly. For example, the presence of the F2FS module allows writing to F2FS file systems with the generic call to write a file, without the application being aware of calling the module internally or writing to an F2FS file system.

However, there is a substantial difference in how kernel modules are managed. Many GNU/Linux systems allow users to choose whether to install a completely open-source kernel (Linux-Libre) or one with some proprietary closed-source drivers. Users can also install additional modules at a later time.

In contrast, Android devices come with a pre-installed kernel that cannot be modified without administrative permissions and that contains binaries whose source code is unknown. This makes it impossible to run specific GNU/Linux programs that require the installation of kernel modules, such as v4l2loopback, a software to use a video stream as a camera.

One possible solution to solve all incompatibilities is to emulate the entire kernel, but this requires emulating the entire operating system, which comes at a significant performance cost. Solutions of this kind will be analysed in chapter 3.1.

A more refined solution consists of emulating only the necessary kernel modules, using the debugging tools available in the kernel itself. On Linux, a program can call another program and request it to intervene at certain points during its execution through the ptrace system call **[4]**. This function, initially designed to allow developers to debug and test programs, effectively allows remapping some instructions, including system calls usually handled by the kernel. Solutions of this kind will be analysed in chapter 3.2.

2.2 CPU architecture

Android **[5]** and GNU/Linux **[6]** are both compatible with x86/64 and ARM processors (including 64-bit versions). However, in practice, almost all Android devices are ARM-based, while most GNU/Linux systems were originally designed for desktop computers, which means that many programs are only released for x86 or x86/64 (such as the Zoom video conferencing software). This generally means that desktop binaries cannot be used without emulation. However, almost all open-source software (those written in portable languages that do not include specific assembly calls) can be recompiled to adapt to the new set of instructions.

Since the ARM architecture is popular beyond mobile devices, such as in embedded devices (like the Raspberry Pi) and servers, major GNU/Linux distributions (including Debian, Ubuntu, and Arch Linux) maintain an ARM port of the base system and all open-source software available in their repositories.

Additionally, modern Linux kernels include a feature called "binfmt_misc" that allows the operating system to register emulators, so that programs belonging to different architectures can coexist in the same operating system. When the user tries to run an executable that is not compatible with the processor architecture, the kernel will automatically recognize it and try to find a suitable emulator; this way, the emulator intervention is limited to the minimum necessary.

2.3 Dynamic libraries and Linker

For copyright and performance reasons **[7]**, Android systems use a C library implementation called Bionic, which is different from the GNU standard implementation, glibc. They have different headers, moreover Bionic does not fully implement the POSIX standard, nor does it include some experimental features found in glibc.

As a result, you cannot run executables dynamically linked to glibc (or any library that uses C library internally) as they are on Android. Additionally, while the linker is located in `/lib` in GNU/Linux, it is located in `/system/bin` in Android. This means you cannot invoke programs directly (e.g., `./myprogram`), you must call the linker (`/system/bin/linker` `` `pwd`/myprogram ``), otherwise you would raise a “Error: file not found”.

With the necessary adjustments, one may recompile dynamic GNU/Linux applications to make them work with Bionic and look for the linker in the correct location. Alternatively, one may distribute programs with a copy of glibc and other necessary libraries. Linux provides several tools, including shell variables LD_LIBRARY_PATH and LD_PRELOAD, to instruct the linker to use the provided libraries instead of the system's.

However, notice that you could not mix C library. Linking against glibc will prevent programs from linking to Android's system libraries, which works with Bionic. This includes closed-source Android drivers distributed as dynamic libraries. Some embedded GNU/Linux systems use a library called libhybris **[54]**, which, supported by a special kernel patch, let you mix Bionic and glibc libraries. However, as mentioned in 2.1, standard Android systems do not allow installing kernel module, so libhybris is outside the scope of this thesis.

2.4 OpenGL API

Typically, applications on Unix-like systems, including Apple devices, use the graphics card through a high-level API written in C called OpenGL. By default, both apps like Chrome, Firefox, and Blender, and system components like Compiz, the window animation manager in many GNU/Linux systems, use OpenGL.

There are other APIs such as Vulkan or Metal (maintained by Apple), which are more difficult to program but offer greater low-level control and therefore find use in the development of complex 3D games or other intensive applications.

Android does not implement the entire OpenGL API, but rather a subset called OpenGL ES. This makes it impossible to run GNU/Linux apps on Android by exploiting the graphics card. However, there is a library called gl4es **[55]** (successor to glshim) that can translate, possibly by emulating, OpenGL 2 calls on graphics cards that only have OpenGL ES 2 drivers. Unfortunately, there are no plans to update it to versions of OpenGL beyond 2, which has been obsolete since 2008.

A different solution, easier to implement, is to use software rendering, that is, implementing the OpenGL API using only the processor, without relying on the graphics card. This approach allows writing cross-platform software that is independent of GPU drivers. The main disadvantage is that the CPU is significantly slower at performing graphics operations compared to the GPU, sometimes even by orders of magnitude.

Among the solutions for implementing OpenGL through software rendering, is worth noting LLVMpipe **[8]**, an open-source driver available in many GNU/Linux distributions including Debian, Ubuntu, and Arch Linux.

2.5 Rendering

Android is based on the concept of "surfaces," which are areas where graphic elements can be rendered. These surfaces are managed by "SurfaceFlinger," which also acts as a window manager. You can interact with SurfaceFlinger through the HWUI library (which succeeded Skia), which provides an API for drawing 2D elements, or by using the OpenGL ES API.

On the other hand, GNU/Linux systems have a graphical server that implements the X protocol, the latest version of which, X11, dates back to 1987. The server receives requests from various applications, called "clients," and communicates with the system to render what it receives; at the same time, it also forwards input such as keyboard and mouse. Over the years, extensions to the base protocol have been consolidated, including Xkb (for keyboard layout management), Xinerama (for side-by-side screens), and GLX (for using OpenGL on X servers).

An interesting property of X11 is that clients and servers are independent: the same applications, in fact, work with any server that implements the protocol. In desktop computers, in addition to the reference implementation Xorg, there are also Xvesa (less performant, but with extremely small sizes, suitable for live systems like Tinycore Linux) and XQuartz (the implementation managed by Apple, currently installed on OS X devices).

The protocol is also network-transparent. Therefore, it is possible to run an application on one machine and have it rendered on another machine connected via the Internet or local network. In Unix, this is achieved simply by setting the shell variable

DISPLAY="SERVER IP:SERVER ID"

Usually, *SERVER ID = 0* for the main screen.

There are also more advanced solutions for streaming, such as the software Xpra, which can save windows in a virtual session and transmit them to other devices via SSH or on a browser via WebSocket.

In general, X servers do not include a window manager, which must be installed separately. Window managers manage how applications are positioned and, if necessary, the graphics of borders and windows. In GNU/Linux systems, window managers can be of two types **[9]** or a mixture of the two.

- Stacking/floating: that is, there are real windows, as in most desktop computers (including Windows), which can be moved around the screen and overlapped with each other. Among the most popular are GNOME, KDE, Xfce, Lxde, Openbox, Twm.
- Tiling: these window managers divide the space into rectangular areas, where applications can be displayed without overlapping. This allows you to manage windows more effectively even without using the mouse. Among the most popular are i3, bspwm, xmonad, and Awesome. This approach is similar to the one used in Android.

Although it is not possible to interface GNU/Linux applications directly with SurfaceFlinger, there are X11 servers for Android that are able to receive requests from X11 clients and render them using a surface as a base.

- X11 Server: a very lightweight server written purely in Java and designed specifically for Android. The original project has not received updates since 2014, but a fork **[10]** continues to keep the software alive by correcting various bugs and making it compatible with new versions of Android.
- XServer XSDL: a port of an X11 server for desktops based on Simple DirectMedia Layer, a cross-platform library for managing graphics, audio, and input devices. The app was programmed before an official port of SDL existed, so it uses an unofficial port made by the same programmer and called Commandergenius **[11]**.

Alternatively, one can use Xvfb (X virtual framebuffer), an X11 server created for testing that implements all graphical operations within RAM without rendering them to the screen. By using applications for remote desktop control, one may transmit the content of the screen emulated by Xvfb to the outside, displaying it on a surface. The standard protocol for streaming is the VNC (Virtual Network Computing) protocol **[12]**. A popular VNC server that enables streaming of an X screen is x11vnc. Numerous apps, including RealVNC (official, closed source) and bVNC (open source), exist for viewing VNC streaming on Android. Additionally, there is a client that can work entirely from a browser called noVNC.js **[70]**, although it requires a fairly recent version of x11vnc to support WebSocket.

As of today, there is no complete implementation of X11 with the GLX extension working on a generic Android. This is because GPU drivers are compiled with Bionic (see 2.3), Android implements only the OpenGL ES subset (see 2.4), and "Surface" objects in Android allow only one

instance of the "Context" object, whereas different windows require different contexts in GNU/Linux systems **[13]**.

For completeness, it should be noted that X11 is not the only protocol in the GNU/Linux landscape. Recent applications are also compiled to support the Wayland protocol, an alternative created in 2008. Wayland is less stable and has less support than X11, but due to its efficiency and security, it is set as the default in some systems, such as Ubuntu 21. I will not cover Wayland in this thesis, as X still enjoys greater compatibility and Wayland requires OpenGL already in the kernel code, making it more difficult to emulate.

2.6 Audio

The audio system used by GNU/Linux operating systems follows the same server/client philosophy seen for rendering in the previous section. The server in this case is called PulseAudio and communicates with a kernel module called ALSA, which in turn communicates with the drivers. Like X, PulseAudio is network-transparent and can be overridden using an environment variable (*export PULSE_SERVER*). However, unlike X, PulseAudio is organized into modules that are loaded at runtime. It is even possible to avoid running all the modules connected to ALSA and redirect the audio to a network socket, for example, by using the combination of the module-null-sink and module-simple-protocol-tcp modules. The data format in which the audio is sent is designed to be easily decodable, so much so that it could be played by Android apps written in less than 200 lines of Java code **[46]**.

2.7 Apps and software ecosystem

In GNU/Linux systems, executable files are compressed and grouped into "packages". A specific program, the "Package Manager", handles the installation and management of packages, often through the command line. Debian-based distros use dpkg+apt, while other options include yum and PacMan. There is no universal format for packages (apt uses the .deb format, yum uses .rpm, and PacMan uses .pkg).

The Package Manager allows for the installation of local packages, as well as access to repositories provided by the distro, which are collections of freely downloadable, pre-compiled, and tested packages. You can also connect to third-party repositories, such as for downloading proprietary or experimental packages.

Each package has its own "Manifest" file, which describes its contents, version, author, and other details. Typically, executable files are dynamically compiled. When they depend on libraries present in the repositories, rather than distributing a copy of the libraries, the dependency is indicated in the Manifest. During installation, the Package Manager will automatically download and install the necessary library packages. For interpreted languages such as Java, Python, and JavaScript/Node.js, you have to list the interpreter as a dependency.

If software is not distributed through the distro's package manager, one can still install it using alternative services such as Snap **[56]**, Flatpak **[57]**, or AppImage **[58]**, or by compiling and

manually copying it into system folders, using the appropriate folders with the "local" suffix. Interpreted languages often also have their own dependency management system, which, not being subject to approval from the distro's staff, allows for more rapid updates.

Packages containing an application with a graphical interface may include a .desktop file and an icon, so that they can be launched by clicking on the icon on the desktop.

In Android, the user does not have the ability to execute binary programs or install binary files directly. The only installable packages, "apk" packages, must have a Java bytecode file as their entry point. Therefore, it is necessary to write part of the app in Java or in one of the languages capable of producing pseudo-compiled Java code (currently, the official website recommends the Kotlin language).

You have to generate APKs using a set of programs called "Android SDK" (Software Development Kit). Every new version of Android comes with a new version of the SDK. APKs compiled with an SDK version are generally compatible with later Android devices. There is also a set of libraries called AppCompat to make APKs backwards compatible by providing an implementation of new functions through old functions.

To run C code inside an APK, you can compile it using the Android NDK (Native Development Kit) and use the Java Native Bridge to interface with it. However, it is typically not recommended to use binary code directly. Apps do not need to interact directly with C libraries because of a vast system library written in Java that provides many high-level bindings.

To execute code in interpreted languages other than Java, you need to include an interpreter in the app. For example, the React Native framework allows writing apps in JavaScript [14].

You may also execute binary files, whether downloaded from the internet or inserted as a resource inside the APK, using instructions like Runtime.getRuntime().exec("command") and the java.lang.Process class. However, as explained in section 2.8, this possibility has been severely reduced for all APKs compiled with SDK 29 (Android 10) or later. Therefore, APKs that intend to use this functionality need to be compiled with SDK 28.

For app development, an official editor called Android Studio is provided, which is already configured to support Kotlin and the Gradle build system.

Apps are generally managed and updated by the Google Play Store digital distribution service, which is a system app on devices, but they can also be installed manually or from other services like F-Droid. However, Google Play does not allow distributing APKs that execute or install arbitrary Java or binary code **[15]**; this clause means that all apps available on Google Play cannot auto-update because they would install arbitrary Java code.

Another clause of the Google Play Store also imposes severe limitations on the SDK version to use for compiling APKs, preventing publishing APKs compiled with SDK versions higher than 28. For these reasons, many of the APKs examined in chapter 3 will only be available through alternative services.

2.8 Permissions and resource access

All Linux systems (and more generally Unix-like systems) have multiple users, one of which, called "root," acts as the administrator. Users can belong to one or more groups, thereby obtaining the privileges associated with that group. For example, users belonging to the *sudoers* group have permission to use the "sudo" program, which allows them to temporarily become administrators.

Each file has an owner and a group owner. Each of the three read/write/execute operations, is associated with three bits that indicate whether that operation is allowed for all, the owner, or the group owner, for a total of nine bits.

In GNU/Linux systems, it is usually not necessary to request special permissions to access peripherals or write to user files. However, there are distributions and package managers, such as Snap **[56]**, where programs are executed in a secure environment and must declare the resources they need in the manifest.

Android uses a similar structure, reserving the root user for only system apps/programs. Costumers/device owners cannot log in as root unless they perform a procedure called "rooting" which, in addition to voiding the device's warranty, entails the complete formatting of the data inside (and therefore is beyond the scope of this thesis).

Normal apps are executed without administrator privileges and in a secure environment, where they only have access to certain resources. To perform some operations, such as reading conacts in the address book, using cameras, or connecting the device to a wifi network, app must declare their intentions in the app's Manifest. Starting from Android 6.0, some permissions, labelled as “dangerous”, must be also confirmed by the user during execution: the app has to request the system to display a pop-up window asking users if they accept that the app performs a certain operation, such as recording via the microphone.

Similarly, apps have limited capabilities to run processes in the background unless they inform the user of their presence using a priority notification **[16]**.

Each app has its own private space within the filesystem, where it can read, write, and execute without authorization. There is also a shared external space accessible to the user, where apps can write, provided they request a "dangerous" authorization. The files in this space do not have an owner or authorization bits like in traditional Linux systems: on the one hand, this means that they can be written and read by all apps with permission to access external storage, but on the other hand, they cannot be executed. This strictly requires that every GNU/Linux program must be installed in the internal space in order to be executed by the app. For security reasons, Android no longer allows binaries to be executed in the internal space of apk compiled with Android SDK 29 or later: this means that if you choose to compile apk with the current version of the SDK, there is no longer any way to install and execute binaries.

It is also worth noting that, starting from Android 11, apps have a folder where they can freely write even in the shared external space, and Google Play will become more selective in publishing apps that require global file access.

2.9 Inter-process communication between apps

Traditionally, Linux apps use a mechanism called Unix domain sockets to allow communication between different executables. Unix domain sockets are channels entirely managed by the kernel and identified by a path, just like files. It is possible to avoid creating a file by using special paths, which are not tied to the filesystem because the kernel directly manages them. In this case, it is called Abstract Namespace Unix sockets **[17]**.

Alternatively, many apps rely on network protocols used locally, such as TCP sockets. The Linux kernel efficiently manages these connections by remapping them before they reach the network, resulting in excellent performance comparable to that of Unix domain sockets (although slower **[18]**). The advantage of network protocols is that they are universal, more widely supported, and allow the program to interface with machines connected through a local network or the internet.

Android implements both Unix domain sockets and TCP sockets but adds new methods integrated with the system's Java library to facilitate communication between apps, such as:

- Intents, which are calls that apps make indicating an action to be delegated more or less generally. The operating system will find the most suitable app to perform the action. For example, if the app requests to open a PDF document, the operating system will choose the default PDF reader or ask the user which PDF reader they prefer. To receive an Intent, the app must declare in the Manifest that it is capable of handling it. You may direct an Intent to a specific app by specifying its ID. Typically, developers call Intents through Java system library, but there is also a program called "am" (Activity Manager) that launches Intents from the command line.
- Broadcasts, similar to Intents, but received by all compatible apps instead of a single app. The system itself uses broadcasts to communicate to apps that a particular event has occurred. For example, the CONNECTIVITY_CHANGE broadcast is sent every time the user changes the internet connection. As with Intents, it is possible to launch broadcasts using "am."
- Binding, an operation where two components create a bond and can call each other's Java functions. To enable binding between components of different apps, Android provides AIDL (Android Interface Definition Language) and the Messenger interface, whose role is similar to that of C library header files. AIDL offers more flexibility, but Messenger is easier to implement.

2.10 Aesthetics and user experience

GNU/Linux distributions do not have a precise aesthetic: Ubuntu alone is distributed in 7 different official flavors **[19]**, each with a different aesthetic. Among the graphic component libraries, GTK (open-source) and Qt (proprietary, but freely available for open-source projects) are popular choices. Typically, users expect that, like on Windows or Mac OS, there is an application bar on one of the four sides of the screen where they can see running apps and launch new ones. Apps are designed for medium or large screens, and it is assumed that users navigate with mouse and keyboard. As mentioned in 2.5, apps are usually managed by a window manager with windows. Users expect that each window has buttons to resize, maximize or close it.

In contrast to GNU/Linux systems, Android has official guidelines on how to create the graphic style of apps. Currently, the "Material Design" aesthetic is in use, with its evolution "Material Design You" focused on user customization, which includes, among other things, setting a background that modifies the system theme.

As mentioned in 1, current emulation solutions make no effort to adapt the aesthetic of GNU/Linux systems to Android, as this incompatibility is not a critical aspect. However, it is important to ensure that users are always able to input the same commands they would on a PC, where both a mouse and keyboard are always available, and where the keyboard also has keys such as Ctrl or Esc. This is typically done by providing a persistent on-screen keyboard or including a button to show/hide the on-screen keyboard and buttons with the keys to be integrated.

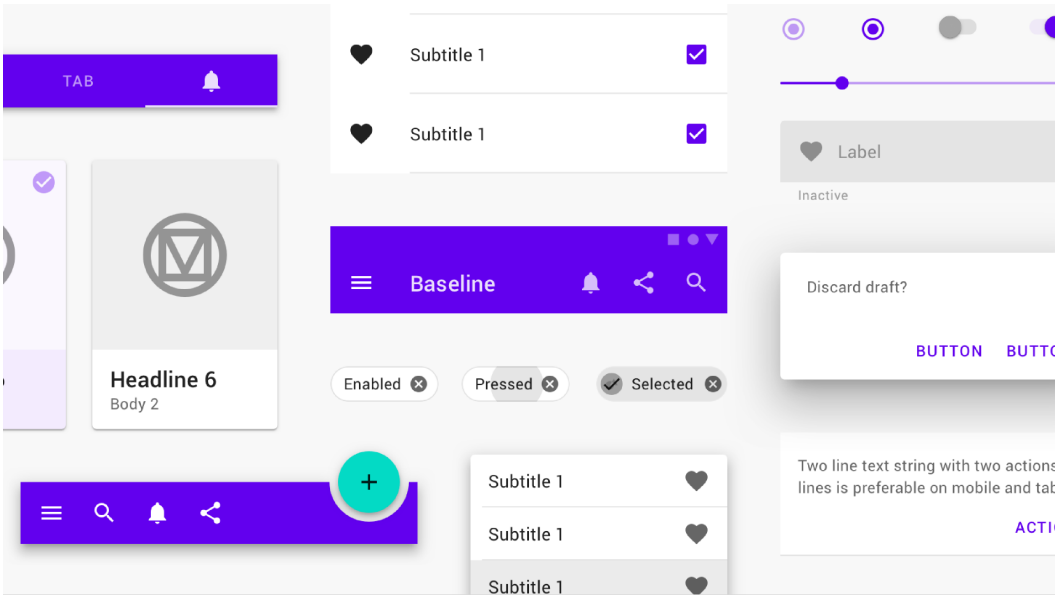

Figura 2: *app widgets following Material Design guidelines*

3. Legacy options

In this chapter, I will review some popular ways to run GNU/Linux software on Android, grouping them into two categories. I'll look at Android apps such as:

- IBochs, Limbo (based on an emulator/hosted hypervisor)
- Termux, AnLinux, Andronix, UserLAnd, and Debian noroot (based on native or chroot)

I'll discuss each app individually, exploring their strategies and how they tackle the issues mentioned in chapter 2. After that, I'll run tests to compare the apps in terms of quality and quantity, with a focus on performance and disk usage.

3.1 Emulator-based (hosted hypervisor)

In this section, I will discuss GNU/Linux software which is not directly executed by the processor, but instead, it is read by a program called "emulator" or "hypervisor" **[20]**, that is capable of interpreting machine code. The emulator executes the entire operating system, including the kernel, rather than just the desired apps. The emulated operating system can communicate with the outside world using virtual devices to perform low-level actions.

The main advantage of this approach is that the emulator has complete freedom to manage the running operating system. For example, when a program wants to write a file to disk, the emulator remaps the call so that it writes to a virtual disk file, without affecting the Android filesystem. Similarly, it is possible to run operating systems of any architecture (including x86), use different kernels, and have greater control over programs.

However, there are numerous disadvantages.

- Of course, the emulated program will have reduced performance, since the emulator must execute many instructions for each instruction in the program. This problem can be mitigated with techniques such as Just-In-Time compilation, but the results are usually unsatisfactory. To achieve performance levels similar to those of native systems, most processors include a hardware module capable of serving as a hypervisor (known as hardware hypervisor). This way, the most onerous operations are delegated to the electronics, and speeds comparable to those of native, unemulated systems can be achieved. The main limitation of this approach is that the processor can only serve as a hypervisor for its own architecture; therefore, it is not possible to use the hardware hypervisor for x86 systems that emulate ARM and vice versa.
- The emulator must be able to faithfully reproduce the architecture of the system it is running, which is not an easy task (only x86 processors have over 1500 **[21]** instructions).
- To emulate a program, it is necessary to start an entire kernel and operating system, often with very long boot times.
- To achieve better performance, emulators must allocate resources in advance. These is not a negligible cost, since it correspond to what you need to run an entire operating system.

3.1.1 IBochs (Bochs)

IBochs **[59]** is a proprietary port of the open-source emulator Bochs, written in pure C. The app lets you boot systems from both virtual disk files and .iso files (commonly used to distribute desktop systems as they can be burned directly onto a disk). It is available for both x86 and x86/64 architectures as well as ARM and ARM64, but it can only emulate x86 and x86/64 systems. I can emulate the presence of keyboards, mice, and various components, including a screen that can be displayed within the app. The official SDL porting is used to render the screen's content.

In addition to the screen, the graphical interface includes a complete virtual keyboard with all the keys, including those not typically found on a mobile device. However, I cannot use my personal keyboard.

There is also another port of Bochs **[22]**, but it will not be covered in this thesis because it was unable to properly boot any of the tested systems on the devices considered.

3.1.2 Limbo (Qemu)

Limbo PC Emulator **[60]** is a port of a part of the QEMU **[61]** (Quick EMUlator) emulator.

The real strength of Limbo is that it is based on QEMU, an extremely versatile software that can be used in numerous different scenarios, even outside of hypervisor emulation, as we will see in 3.2.

Like in 3.1.1, the app allows me to boot systems from both virtual disk files and .iso files, and allows me to emulate the presence of keyboards, mice, and other components.

Unlike IBochs, it supports the emulation of ARM and ARM64 systems (and many others) as well as x86 and x86/64. To avoid weighing down the apk file, the app has been split into several apk files, one for each available architecture.

In addition to the features available in IBochs, Limbo has multiple graphics engines and also allows for streaming of the virtual screen via the VNC protocol, allowing the user to view the emulated operating system through their preferred VNC client.

It should be noted that by installing a special kernel module (KVM **[62]**, kernel-based virtual machine), Limbo is able to access the hardware hypervisor, which allows for running operating systems (not only Linux, but also Windows) with very high performance, comparable to native performance. For the reasons stated in 2.1, QEMU+KVM will not be covered in this thesis; however, comparing the performance of QEMU and QEMU+KVM will be useful for estimating, in a first approximation, how much software emulation affects the performance of the operating system.

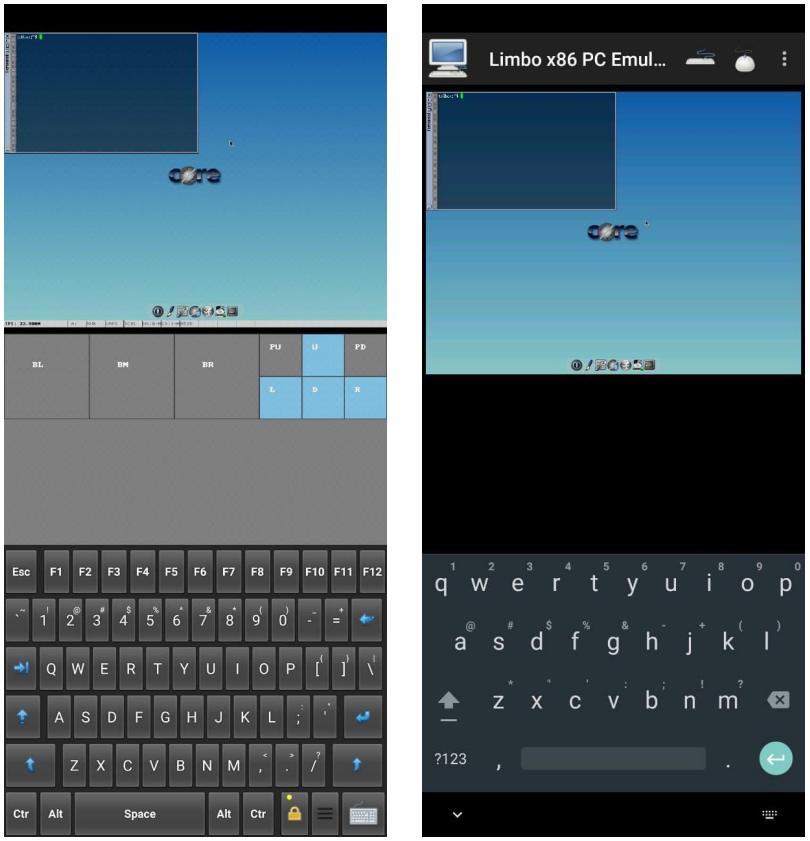

Figure 3: *Comparison between IBochs (left) and Limbo (right). While Limbo allows for the use of a personal keyboard, IBochs implements one within the screen.*

3.2 Native solutions or based on Chroot mechanisms (OS-level virtualization)

In these solutions, within the private folder of the app, a set of programs and libraries that make up a GNU/Linux distribution are downloaded while maintaining all or part of the filesystem structure (this set of files is generally called the root filesystem, abbreviated as "rootfs"). Then, through various mechanisms, all system calls are remapped so that they are compatible with Android and, instead of using Android system files, they use those in the app's private folder. This can be done either by intervening beforehand on the source code (and therefore the resulting binaries will be entirely native and will have no performance loss), or during runtime through kernel features. In these cases, it is improper to speak of "emulation," but the term "virtualization" is preferred **[23]**, as emulation implies that the behavior of a hardware component is being reproduced.

The Linux kernel includes a specific command for remapping calls at runtime: "chroot." Chroot allows programs to be run using a subfolder as a base instead of the system's base folder. For example, if you use `chroot /home/myfolder` and then start a program that writes a file to `/tmp/file.txt`, thanks to chroot, the file will be written to `/home/myfolder/tmp/file.txt` instead. Similarly, if the program tries to read the library `/lib/mylib.so`, it will instead load `/home/myfolder/lib/mylib.so`.

Files within a chroot environment (called a chroot jail) are isolated and cannot access the rest of the device. It is possible to make external folders available through the "mount" command.

The chroot command, as well as the mount command, require root privileges to be used directly (and therefore their direct use will not be discussed in this thesis), but there are alternatives that, using kernel debug tools (see 2.1), allow you to obtain the same functionalities without administrator permissions.

The most common tool for achieving chroot-like functionality on Android is the PRoot software **[24]**, which not only emulates chroot but also provides the functionality of the mount command and the binfmt_misc module (see 2.2). Unlike chroot, PRoot does not guarantee complete isolation of programs (a malicious program could make system calls to escape the PRoot jail), but this is not a problem because such programs are subject to numerous restrictions imposed by Android. PRoot also offers the ability to fake root permissions for the user, which is useful because some programs (such as the apt package manager) refuse to start unless executed with administrator privileges. Additionally, by using PRoot as binfmt_misc and installing the QEMU emulator, it is possible to run programs compiled for architectures different from that of the device, albeit with poorer performance.

In general, the advantage of using chroot-based systems is that they do not require emulation of a CPU or an entire kernel, nor do they require intervention on many of the executed instructions. For this reason, these approaches (called OS-Level Virtualization) are considered the fastest, even when compared to hardware hypervisors **[25]**. Moreover, the ability to execute individual binaries without the need to start the operating system (which consumes time and computational resources) makes these solutions useful for distributing single applications in perfectly reproducible environments. This has given rise to software such as Docker (which, however, requires root permissions and therefore cannot be used for the purposes of this thesis) to simplify and automate the distribution and management of applications executed in chroot.

The main disadvantage is that the lack of a hypervisor limits emulation possibilities. For example, it is not possible to start GNU/Linux systems that are incompatible with the Android kernel. Additionally, these solutions require binary files to be executed within the app's private space, which, as explained in 2.7 and 2.8, makes it impossible to compile them with the latest versions of the Android SDK and distribute them on the Google Play Store.

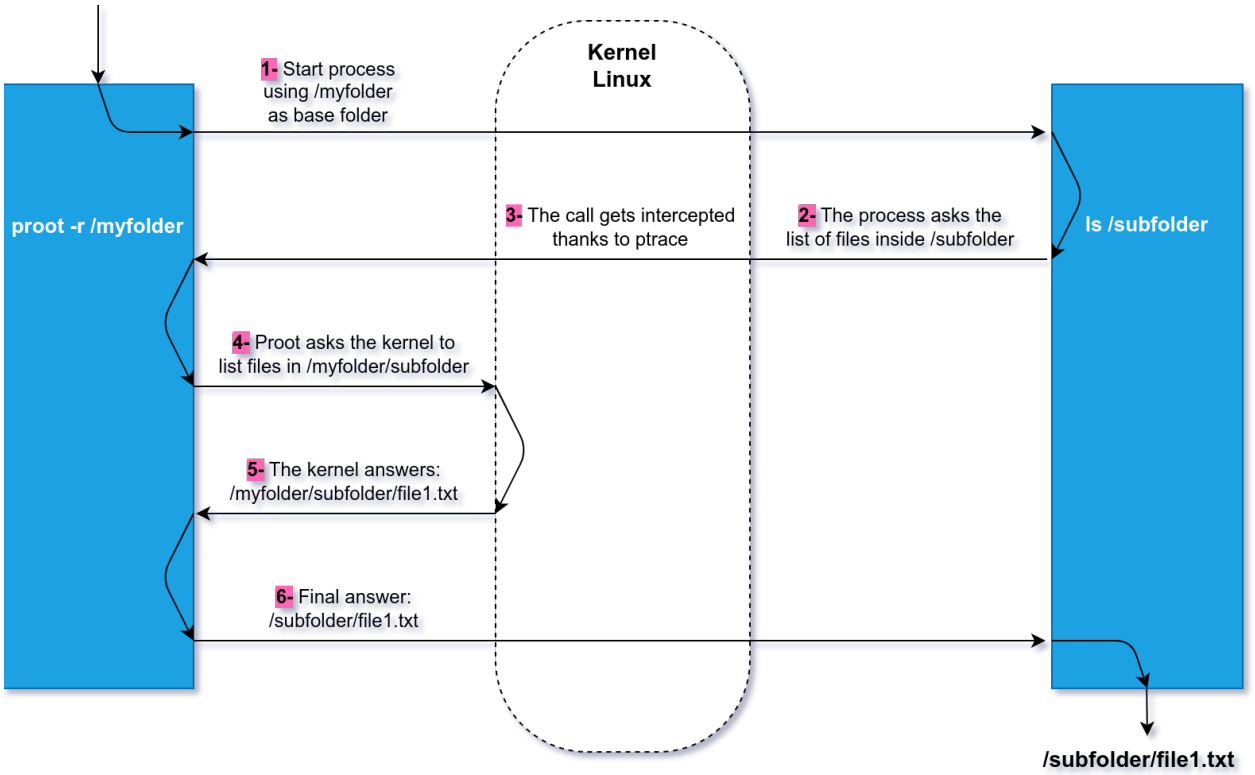

Figure 4: *Running ls (list file) using PRoot to remap the call.*

3.2.1 Termux

Termux **[63]** is an app that lets the user install and run ports of the main GNU/Linux command line applications via a terminal. It includes a package manager, pkg, based on apt **[26]**, and specific repositories created specifically for the app.

To work with Termux, programs are properly patched and recompiled to be compatible with Android. This patch allows binaries to:

- adapt to the Android file system, in particular limit their read and write to the private folder of Termux (see 2.8);
- work with only one user and without root privileges (see 2.8);
- search for the linker in /system/bin and be linked with Bionic instead of glibc (see 2.3).

The changes made remove the need to run binaries inside a chroot jail. In this sense, it can be said that rather than a complete GNU/Linux system, Termux is to be considered a "Linux Environment" that coexists natively with Android **[27]**.

By downloading an auxiliary app, the programs and executable scripts within Termux can interface with the Android system thanks to a set of executables called termux-api. These programs communicate with the app using the Abstract Namespace Unix Socket (the path is communicated to the app via a Broadcast launched by am, see 2.9) and allow operations such as checking the battery status, taking a photo, or reading received SMS.

In addition to various command-line utilities, Termux provides a range of graphical interface applications organized in a separate repository called x11-repo, which also includes Xvfb and x11vnc. Termux does not provide a way to display an X server within the app, so it is necessary to download an X server or a separate VNC client to interact with graphical interface programs.

Termux repositories contain only a reduced part of existing GNU/Linux software **[28]**, in particular, many graphical programs are missing, including LibreOffice, Audacity, Chromium, and Firefox. However, it is possible to install PRoot and start a chroot jail to use a system like Debian, Alpine, or Arch Linux. This can also be done with the help of the proot-distro package, which allows downloading proot and minimal rootfs (without a graphical interface) of the most popular distros. proot-distro **[64]** also provides a series of commands that help with updating and backing up such distros, but it does not include commands to simplify the installation of graphical environments.

Termux is available on both the Google Play Store and the alternative open-source software market F-Droid, but the version on Google Play is outdated (it can no longer be updated due to the restrictions listed in 2.7). In particular, the app on Google Play is set to an inactive repository; if the user does not intervene and modify it, they will not be able to install any packages **[29]**.

3.2.1.1 AnLinux and Andronix

In order to simplify the process of installing a distro's rootfs and PRoot inside Termux, there are some apps that guide the user step by step. The two most popular ones available are AnLinux and Andronix.

Both provide an interface where the desired distro can be chosen, both have ready-to-copy-and-paste commands to be entered in Termux, and both have their own mirror from which to take the rootfs base.

Both support the major Linux distros (Debian, Ubuntu, Arch Linux, and Alpine) and the ability to choose your own window manager.

AnLinux is open-source (and therefore also available on F-Droid **[30]**), while Andronix is only partially open-source **[31]**.

In addition to the basic distro versions, Andronix includes some purchasable closed-source versions that, according to the developers, have been optimized and tested to work better on various Android devices. These distros will not be covered in this thesis because it is unclear what modifications have been made: some preliminary tests have not been able to detect any significant performance differences compared to PRoot systems run by UserLAnd (see 3.2.2), and it has not been possible to conduct more accurate analyses due to the closed-source nature of the project.

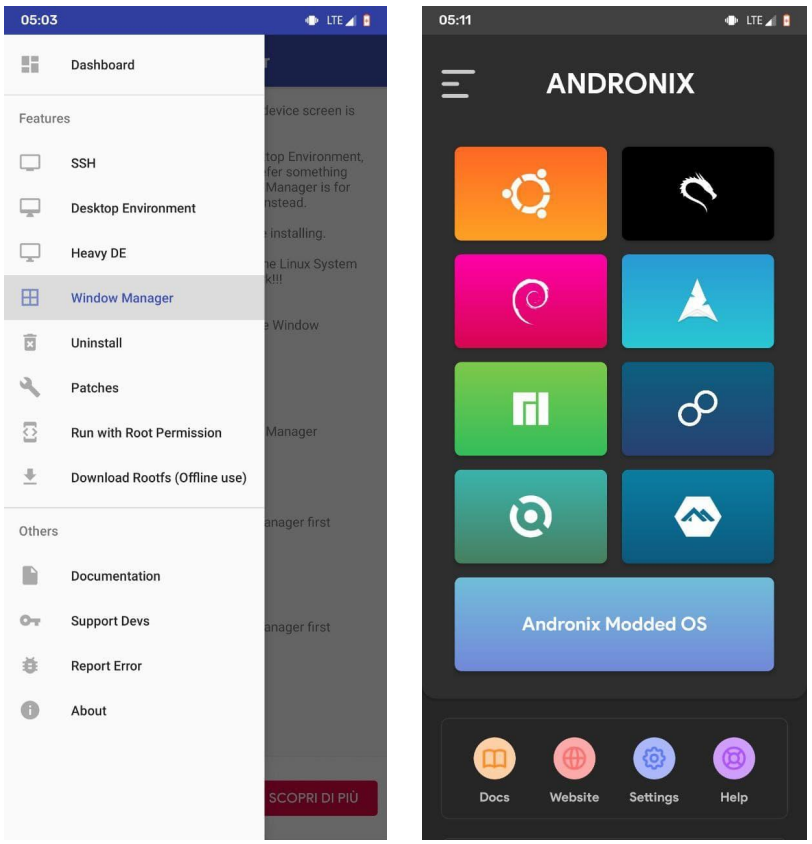

Figure 5: *Comparison between the opening page of AnLinux (left) and Andronix (right). Andronix has a significantly more polished graphical appearance.*

3.2.2 UserLAnd

UserLAnd **[65]** is an app that allows me to install the rootfs of the most popular distros (including Alpine, Debian, Ubuntu, and Arch Linux) and run them using PRoot. All distros are downloaded from a dedicated website and come preconfigured with Xvfb and x11vnc (for screen streaming via VNC protocol), the minimal window manager twm, and a terminal.

I can interact with each distro in three ways:

- Using a terminal interface included in the app, identical to that of Termux because it is based on its source code **[32]**;
- Launching it so that applications are rendered inside the XServer XSDL X server (this option is disabled for devices with Android 10 or later, as it no longer works);
- Viewing it through an external VNC client, not included in the app.

The app's advantages certainly include ease of use and extreme stability, as all distros are distributed preconfigured and tested. However, there are also some critical points. The app cannot handle multiple VNC sessions, and it presents bugs when multiple sessions are launched using the included terminal interface.

- Starting a VNC session is particularly cumbersome: first, I need to install an external app. Furthermore, to prevent unauthorized connections, the app protects the connection with a password that I have to enter every time I start a new session.
- The preinstalled twm window manager is extremely lightweight but also very inconvenient to use, especially with a touch screen. For example, it lacks a taskbar and requires holding down the right mouse button to show/hide screens.
- The app provides only one rootfs per distro, so I must download the VNC transmission packages even if I only want to use command-line functionality.

3.2.3 Debian noroot

Written by the same developer of XServer XSDL, Debian noroot **[66]** allows me to use a Debian distro (with Xfce desktop environment) executed through PRoot and rendered through XServer XSDL. Debian Noroot is an alternative to solutions based on Xvfb and x11vnc that potentially could achieve better results: it is the only app that integrates a real porting of the X server, allowing for excellent performance and numerous native functionalities, such as built-in audio support in the X server **[33]**. However, this strength is also its weakness: XServer-XSDL, due to its extensive codebase, requires a lot of effort to maintain and adapt to Android updates over time. Historically, Debian Noroot has been the best option for installing Debian, but nowadays, due to lack of support from the original author, it has become unusable on newer devices. The app presents numerous graphical glitches, often fails to handle keyboard inputs, and some programs do not start. Therefore, it will not be further discussed in this thesis.

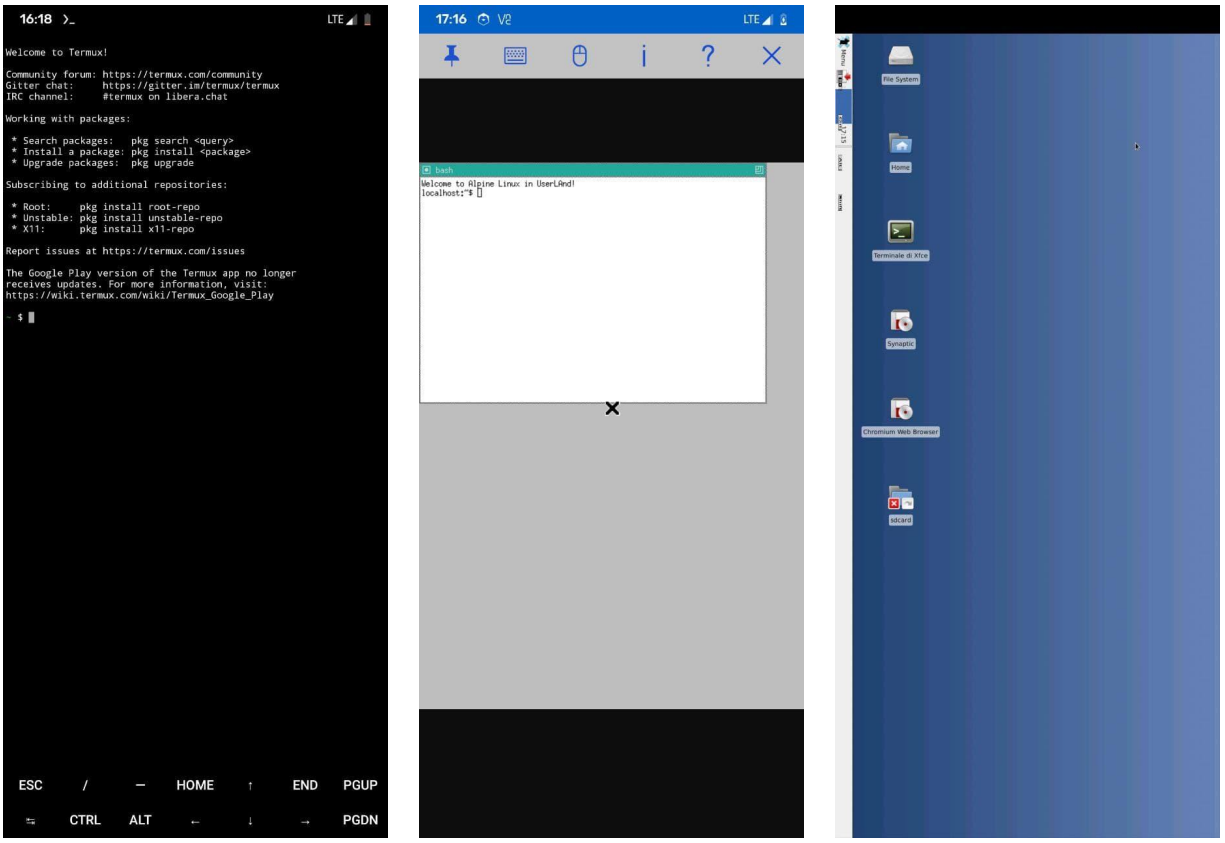

Figura 6: *Confronto tra tre diversi modi di visualizzare i programmi GNU/Linux: l'emulatore di terminale fornito da Termux (sinistra), una distro UserLAnd visualizzata usando RealVNC (centro) e il server X (XServer-XSDL) integrato in Debian noroot (destra).*

3.3 Comparative tests

In this section, I will report on the tests specifically conducted for this thesis. To facilitate comparison, I have chosen to present both the data collected from the apps analyzed in this chapter (sections 3.1 and 3.2) and the results achieved by my solution in chapter 4.

3.3.1 Available features

For the comparison, I considered the following features:

1. Works on recent versions of Android
2. Allows running GNU/Linux command line programs
3. Allows running simple graphical programs
4. Allows running complex graphical programs, such as the Chromium browser
5. Includes audio support
6. Includes hardware acceleration support
7. Runs at native or near-native speeds (without needing root access)
8. Allows the installation of complete GNU/Linux distros
9. Is compatible with .iso files (including the kernel)
10. Works and can display graphical programs without needing to install external apps/with 1 installed app/with 2 installed apps
11. Provides an API to interact with Android system libraries
12. Has an expandable graphical interface through external programs.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
IBochs	Yes	Yes	Yes	Yes	?*	No	No	Yes	Yes	0	No	No
Limbo	Yes	Yes	Yes	Yes	?*	No	No	Yes	Yes	0	No	No
Termux (without PRoot)	Yes	Yes	Yes	No	Yes	No	Yes	No	No	1	Yes	No
Termux (proot-distro)	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	1	No	No
Termux (AnLinux or Andronix)	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	2	No	No
UserLAnd	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	1	No	No
Debian noroot	No	Yes	Yes	Yes	Yes	No	Yes	Yes	No	0	No	No
Unyw (my solution)	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	0	Yes	Yes

* in both cases, there are options to adjust the audio, and the documentation suggests that support is present. However, despite numerous attempts, I was not able to activate it with any of the tested configurations on any device.

3.3.2 Performance benchmarks

In this paragraph, I have conducted some tests to quantitatively evaluate the performance of the analyzed solutions. Due to the nature of the problem, traditional benchmarking tools such as sysbench were not reliable in emulated environments, as they often provided inconsistent results, and sometimes their measurements were completely wrong. An example of this is discussed in an online forum **[67]**, where users trying to compare the performance of Termux and PRoot with sysbench obtained three results that contradicted each other, with one result being mathematically impossible.

Therefore, I chose to rely on practical tests specifically designed for the problem, in which actions were performed on a real device and the required time was measured (the device was filmed with an external device, and the duration of the footage was measured). This approach obviously provides less precise results, is more difficult to reproduce, and is more affected by unpredictable factors, but it ensures the correctness of the result. As evident from the data themselves, the results obtained were still consistent and did not show excessive discrepancies. All data obtained are reported in the minutes: seconds format. For each test, ten measurements and their average are reported. To limit the influence of errors, measurements were taken at different times and after a device restart.

The tests were conducted on a Redmi Note 7 mobile phone (4GB of RAM, Octa-Core Snapdragon 660 SoC processor, LineageOS 19) and an HP computer (8GB of RAM, AMD Ryzen 5 3500U processor, XUbuntu 20.04 LTS).

3.3.2.1 Booting of Tinycore Linux

In this test, I measured the time it takes to boot the Tinycore Linux 12 operating system from an .iso file.

PC - Qemu:

0:41	0:45	0:45	0:38	0:45	0:42	0:45	0:43	0:39	0:44	(Avg) 0:43

PC - Qemu+kvm:

0:07	0:08	0:09	0:07	0:07	0:08	0:08	0:07	0:08	0:07	(Avg) 0:08

Smartphone - IBochs: data was measured after activating the "Unlock speed" option, which allows for better metrics, and using 1024Mb of RAM.

2:11	2:12	2:53	2:05	2:15	2:09	1:59	2:07	2:10	2:06	(Avg) 2:13

Smartphone - Limbo: data was measured after activating the "High priority" option and using 1024Mb of RAM.

2:14	2:17	2:16	7:53	6:11	2:18	2:15	2:21	4:54	2:38	(Avg) 3:32

3.3.2.2 Execution of fibonacci.sh

This test runs a script (Code 1) that calculates the 20th number in the Fibonacci sequence using a particularly inefficient algorithm (it is even exponential). The test is particularly indicative in testing the effectiveness of the analyzed solutions in creating new processes, as it forces the system to create two for each sub-instance.

```
#!/bin/bash
fibonacci()
{
  if (( $1 < 3 ))
  then
      echo 1
  else
      echo $(( $(fibonacci $(( $1 - 1 ))) + $(fibonacci $(( $1 - 2 ))) ))
  fi
}
date
fibonacci 20
date
```

Code 1: *fibonacci.sh script*

PC - native:

0:10	0:11	0:11	0:10	0:10	0:10	0:10	0:10	0:09	0:10	(Avg) 0:10

PC - chroot command, Debian rootfs:

0:12	0:11	0:12	0:12	0:12	0:12	0:12	0:12	0:12	0:12	(Avg) 0:12

PC - PRoot, Debian rootfs:

0:22	0:20	0:21	0:20	0:23	0:22	0:22	0:21	0:22	0:23	(Avg) 0:22

Smartphone - Adb debug tool:

1:22	1:02	0:52	1:24	0:48	0:55	1:00	1:00	0:58	1:15	(Avg) 1:04

Smartphone - app IBochs: I measured the data within a Tinycore Linux terminal after activating the "Unlock speed" option, which allows for better metrics, and using 1024Mb of RAM

16:44	16:05	18:10	18:23	17:32	18:01	16:40	17:54	18:22	20:21	(Avg) 17:49

Smartphone - app Limbo: i dati sono stati misurati all'interno di un terminale di Tinycore Linux dopo aver attivato l'opzione "High priority" e utilizzando 1024Mb di RAM

25:36	27:12	21:11	24:12	21:31	27:13	26:18	22:40	24:39	21:31	(Avg) 24:12

Smartphone - app Termux:

1:34	1:37	1:23	1:34	1:26	1:34	1:40	1:25	1:28	1:36	(Avg) 1:32

Smartphone - app Andronix, rootfs Debian:

1:46	1:51	1:59	1:47	1:59	1:55	1:58	1:47	1:56	1:58	(Avg) 1:54

Smartphone - app Andronix, rootfs Alpine Linux:

2:11	2:15	2:11	2:11	2:05	2:09	2:05	2:11	2:08	2:07	(Avg) 2:09

Smartphone - app AnLinux, rootfs Debian:

1:48	1:49	1:57	1:49	1:55	1:57	1:48	1:55	1:55	1:53	(Avg) 1:53

Smartphone - app AnLinux. rootfs Alpine Linux:

2:12	2:14	2:09	2:13	2:08	2:09	2:03	2:14	2:08	2:10	(Avg) 2:10

Smartphone - app UserLAnd, rootfs Debian:

1:39	1:39	1:41	1:38	1:38	1:41	1:39	1:39	1:38	1:39	(Avg) 1:39

Smartphone - app UserLAnd, rootfs Alpine Linux:

2:01	1:58	1:58	1:59	1:58	2:02	1:58	2:01	1:59	1:59	(Avg) 1:59

Smartphone - app Unyw (my solution):

2:03	2:01	1:57	1:58	2:00	2:01	1:58	2:01	2:02	1:58	(Avg) 2:00

From the comparison, it is evident that the only solutions with unacceptable performance are those based on hosted hypervisors, which are about 10 times slower even than those based on PRoot.

PRoot-based solutions achieve slightly worse performance, but still comparable to native ones. Andronix and Anlinux show similar performance, as well as UserLAnd and my solution (which is exactly what one would expect, as they use the same binaries). It was not possible to measure Debian noroot's performance on the Redmi Note 7, but other measurements on devices where the app worked suggest that it performs similarly to UserLAnd. It is worth noting that, in general, solutions based on Alpine Linux perform slightly worse than their counterparts based on Debian. One possible explanation is that the Bash implementation present on Alpine is compiled with the musl C library, which is smaller in size but in some contexts performs worse than glibc **[45]**.

3.3.3 App size

3.3.3.1 Emulator-based solutions (hosted hypervisor)

IBochs takes up 26Mb of space, while Limbo x86 takes up 33Mb. These values do not include the size of the ISO images and the virtual disk space.

Generally, ISO images are heavier than rootfs because they include a copy of the kernel. For comparison, the ISO version of Alpine Linux specifically optimized for emulators weighs 44Mb on its own. Even in distros like Tinycore Linux, where the kernel has been compressed as much as possible, it still takes up 5Mb, approximately one-third of the total weight of the ISO.

3.3.3.2 Native solutions or based on Chroot mechanisms (OS-level virtualization)

In this section, I will report the space occupied by the analyzed apps in section 3.2 after the first launch. The data is reported as calculated by the Android operating system. The space is divided between the weight of the APK itself and the weight of the files within the folders reserved for the app, which includes both the internal private folder and the folder assigned to the app on the shared space, typically called "./Android/data/APP_ID".

App name	**Apk size (Mb)**	**User data (Mb)**	**Total (Mb)**
Termux (terminal)	110	75	185
UserLAnd (Alpine, terminal or VNC)	30	171	201
UserLAnd (Arch, terminal or VNC)	30	1220	1250
UserLAnd (Debian, terminal or VNC)	30	526	557
UserLAnd (Ubuntu, terminal or VNC)	30	430	461
Debian noroot	201	980	1181
Unyw (soluzione proposta, senza il supporto Pulseaudio)	10	48	58
Unyw (soluzione proposta, includendo Pulseaudio)	15	71	86

3.4 Final considerations

From the tests performed, it is clear that native approaches or OS-level virtualization-based approaches are always preferable when possible, both because they are more efficient and more integrated with the system, and because they can avoid a long boot time. Even in critical situations such as the one presented in 3.3.2, applications that use PRoot are still able to achieve execution times comparable to native ones, while hypervisor solutions prove to be significantly inefficient, up to 10-20 times slower.

The two hypervisor solutions considered have comparable performance (tests suggest that IBochs is slightly more performant, but in rare cases, the difference is less than 30%). Therefore, in situations where the considered distro is compatible with both apps, there is no real reason to use one over the other. IBochs is easier to install (can be downloaded from the Play Store) and has a cleaner graphical interface; however, Limbo offers many options that are absent in IBochs, such as the ability to use KVM or run ARM systems (although this latter feature requires downloading a separate app).

To run applications natively, the only existing solution, Termux, has proven to be practical and functional: most typical command-line programs of GNU/Linux systems are present in its repositories, and it is the only one among these solutions that presents an API, albeit limited, to interact with the Android system. The main disadvantage is that the app is oriented for use from the command line, and therefore lacks most of the graphical interface applications; it is also not easy to install and run the few graphical interface applications present.

For OS-level solutions based on PRoot, the best experience was obtained with UserLAnd. Both Andronix and AnLinux require several steps to configure the system; both also do not provide complete rootfs and therefore must perform part of the installation inside the device, which can be a long operation. It should also be noted that both Andronix and AnLinux are installable from the Google Play Store, and this is a serious problem because the unaware user may not know that they are incompatible with the version of Termux present in the Google Play Store (note: recently, Andronix partially solved the problem by adding a warning that explains the situation). The only advantage of using these apps is that the system is installed inside Termux; this allows for more control over the files and the ability to use Termux programs on them (for example, using the termux-job-scheduler API, it is possible to set an alarm to start PRoot at a certain time).

To sum up:

Category of the solution	**Are the performance levels acceptable?**	**Can it access the entire software catalog of GNU/Linux distributions?**
Software hypervisor	No	Yes
Native porting (Termux)	Yes	No
PRoot	Yes	Yes

- Overall, it is clear that PRoot-based solutions have the potential to be the most advantageous. However, there are some critical issues that are not resolved by any of the apps analyzed.
-
- Graphical GNU/Linux programs have the same interface as they had on PCs, which makes them unsuitable for use on mobile devices.
-
- The Android app startup process requires several steps, including opening an external app to view the screen transmitted with VNC.
-
- There is no real integration with the Android system. For example, consider a user who wants to convert a PDF file received via email using LibreOffice: performing this simple operation requires at least 4 different Android apps to be opened (the app from which the file originates, the app that runs PRoot, the app that displays VNC, a file manager that allows you to open the converted file), and there is no way to simplify the process.
-
- It is not possible to extend these solutions except by adding Linux programs within the PRoot environment; these are in any case limited extensions and difficult to share with other users.
-
-
-
-

4. My solution

In this chapter, I will describe a solution that, although based on PRoot virtualization, presents an alternative approach in managing programs. This solution will be described both theoretically and practically, as a concrete and functional implementation has been created to demonstrate its feasibility. The code for this implementation is freely available on the github.com/unyw repository (the name is a contraction of the words Unix and Anywhere). The reason the theoretical part and the implementation are not treated separately is that, in many cases, empirical considerations will be necessary to proceed. Similarly, sometimes the most sensible design choice will pose theoretical problems that would not otherwise need to be addressed in this thesis.

The goal is to maintain the advantages of PRoot virtualization highlighted in section 3.3, such as near-native speed and access to all the software of a distro, but provide new types of interfaces and functionality to existing programs, without requiring patching or recompiling. My solution will also aim to address some of the common criticisms of UserLand, AnLinux, and Andronix, such as making it easier to install and run GNU/Linux programs.

The approach will be based on the use of "wrappers," which are local programs written in HTML, CSS, and JavaScript that can encapsulate GNU/Debian programs (running, displaying, and remapping their functionality to make them more easily accessible from Android) and communicate with Android itself, becoming bridges that allow the two systems to interact with each other. To make it possible to install and run these web apps, an Android application (unyw.apk) will need to be programmed to manage them and allow them to communicate with the system libraries. Additionally, development tools will be provided to simplify the process of writing these web apps and publishing them. (Note: in the following text, the terms "wrapper" and "web app" will be essentially treated as synonyms.)

There are numerous advantages to using web technology programmed wrappers:

- Created interfaces can be easily and efficiently viewed within an Android app using the system library's "webview" component. This component is based on Chromium and is automatically updated in all modern versions of Android (Android 4.4 and later **[34]**).
- Web apps are composed of simple HTML, CSS, and JavaScript files. To install them, you only need to download and copy them into the private files of the Android application without the need for Android or special system permissions. This makes them easily manageable by any file package manager, including the distro run with PRoot, and can be distributed through the distro's repositories and easily synchronized with the programs they encapsulate (just indicate these programs as dependencies in the manifest).
- Web apps are generally quick to write because they require minimal post-processing (unlike Android apps, which as explained in 2.7, require pseudocompilation through Android SDK, a long and complex operation) and because web technologies are higher-level when compared to Java.
- In terms of disk size, these apps are also much lighter than Android apps and especially GNU/Linux graphical programs.
- It is possible to display a GNU/Linux program within a web interface using the "noVNC.js" VNC client **[70]**. This makes it possible to easily compose hybrid interfaces that display encapsulated programs but add controls and new graphical elements. It should be noted that noVNC.js is a complete implementation of the protocol and therefore allows for automatic

interaction with the screen being displayed, for example by sending a sequence of keys from code.
- Some well-known programs such as Visual Studio Code or Jupyter Notebook use a web interface by default. Using web interfaces within the solution means that such programs can be easily viewed practically without any graphical modifications.

In addition, websites are a universal format, which simplifies development and debugging because apps can be viewed and tested on PCs without the need for an emulator. However, the main disadvantage of the chosen approach is that web apps run in a protected and controlled environment (the webview, similar to a browser), which means they have access to fewer resources and, in general, have fewer possibilities to interact with the system. For example, a website cannot read and write files or have unrestricted access to processes or sensor data.

To solve this problem, it is possible to establish a channel that connects web apps with the Android application so that web apps can request the Android application to execute code outside the protected environment (in this case, we talk about "hybrid web apps"). Such a channel (which will be formalized in 4.2) can also provide a simple way for GNU/Linux programs to access Android functionalities, similar to what the Termux-api described in 3.2.1 does.

It should be noted that the created app, although technically usable, is not intended to be a product distributable to real users. It should be understood as a prototype created exclusively for this thesis, whose purpose is to demonstrate the potential of my solution. For example, the focus will not be on creating wrappers for numerous GNU/Linux programs but on creating some that exemplify the potential of the API and illustrate some of the design logics that can be used.

4.1 Development philosophy

In addition to the requirements imposed by the solution being sought, I decided to also design and develop according to additional criteria aimed at achieving a more organized, coherent, and easily manageable final code.

- **No compilation:** I chose to avoid inserting any code that required compilation. The resulting solution was therefore entirely developed with interpreted/pseudo-compiled languages including Ash shell, Kotlin, JavaScript/CSS/HTML/Svelte. All binaries present, such as PRoot and Busybox, were precompiled from external sources. This approach has a dual advantage: it simplifies the app generation process (as seen in 4.4.1), and it makes the final result more stable by delegating some critical parts to third-party projects with hours of work and debugging behind them.
- **Abstraction of the Android system:** where possible, I chose not to directly expose the wrappers to the complexity of the Android system, but to abstract the functionalities and resources provided by it. The opposite choice could have been made, and this possibility will be evaluated in 4.4.2.1 by explaining its advantages and disadvantages and why it was not adopted.
- **Single Android apk app:** I chose to produce a single apk file that works on all architectures, making the resulting app easier to distribute. However, this also means that the rootfs must be distributed separately from the main apk file, among other things-

4.2 Project structure

Overall, I have decided to divide the code I produce into three parts, each with a different Github repository:

- the rootfs, one for each supported CPU architecture (i.e., x86, x86_64, ARM, and ARM64) (repository: github.com/unyw/android-rootfs, will be discussed in 4.3);
- the Android application (repository: github.com/unyw/android-apk, will be discussed in 4.4);
- the wrappers (repository: github.com/unyw/repo-main, will be discussed in 4.5).

I have also created a repository called github.com/unyw/template-app, which can be cloned and used as a base for wrapper development.

To simplify the app distribution process, I have chosen to include both source code and a prepared distribution version in each repository. Github offers a service called "Github Pages" **[35]**, which allows configuring a repository so that its files can be downloaded via HTTPS: this makes it possible to use the repositories themselves for app distribution, without relying on external services.

The chosen division is a natural consequence of the fact that each group produces a resource that needs to be downloaded separately from the others. For instance:

- maintaining a single app valid for all CPU architectures means that it must download the rootfs separately; the alternative would be to include all possible rootfs architectures inside the apk file, making it extremely heavy and unnecessarily large.
- Including all wrappers and applications that encapsulate them within the rootfs would make them extremely heavy. Therefore, I have chosen to include only a minimal selection of wrappers/web apps in the rootfs and leave the user the possibility to install the rest.

This division also has the advantage of separating the Kotlin source code, contained entirely in the android-apk repository, and the web source code, contained mostly in the repo-main repository.

4.3 Distro choice and rootfs generation

After considering Debian, Ubuntu, Arch Linux, and Alpine, I chose Alpine as the distro, both because of its extremely small size and the speed of its package manager **[36]**. In hindsight, this choice turned out to be particularly fortunate, considering that the size of the resulting app is smaller than all the other solutions considered.

I then proceeded to create a script that automates the process of generating the rootfs. The script takes care of:

- Downloading a minimal rootfs from the official Alpine website and extracting it into a folder.
- Accessing the system using PRoot.
- Within the system, adding the github.com/unyw/repo-main/stable repository and downloading all the necessary packages for a minimal installation, including unyw-bridge-android, unyw-app-home, and unyw-app-xterm (see 4.5.2).
- Compressing the archive again (I used xz because it provided the best compression ratio).
- Generating a checksum of the resulting file.

This operation is repeated for each of the four supported architectures (armv7, aarch64, x86, x86_64), using qemu when necessary.

4.4 Android app

To enable the execution of GNU/Linux programs in the background even when the user is no longer actively using the Android app, the app must consist of at least two components:

- an *activity*, which is the active component responsible for displaying web apps using a webview;
- a *service*, which is a background component responsible for managing GNU/Linux processes even when the user closes the activity or views another app. As explained in 2.8, such a service cannot be completely in the background, but must inform the user of its presence through a priority notification (referred to as a "foreground service") or it risks being terminated by the Android system to free up resources.

Upon opening, the activity checks if the service is running and starts it if necessary. Then, a connection is established between the activity and the service so that they can perform inter-process communication. This is initially done through an Intent and then through "binding" (see 2.9). This connection persists until the activity is displayed on the screen and is revoked whenever the user closes or backgrounds the activity.

When the user reopens the activity, it checks if the service is already running and creates a connection with it again. This way, there is no need to restart processes like x11vnc, and web apps can connect to previously opened processes.

While the activity consists of only one component (which essentially starts the service and displays the webview), the service itself is divided into three parts, each with a specific function and each with its own thread.

The API server, which is responsible for making Android functionality and resources available to the wrappers and GNU/Linux programs. This API will be described in detail in 4.4.1.

The module that handles the main PRoot process; this module is responsible for installing the rootfs if it is missing, and then starting the processes that should always be active in the background, such as Xvfb and x11vnc. The installation and startup processes will be discussed in detail in 4.4.2.

The module that manages the other processes and terminates them. Two possible implementations will be discussed in 4.4.3.

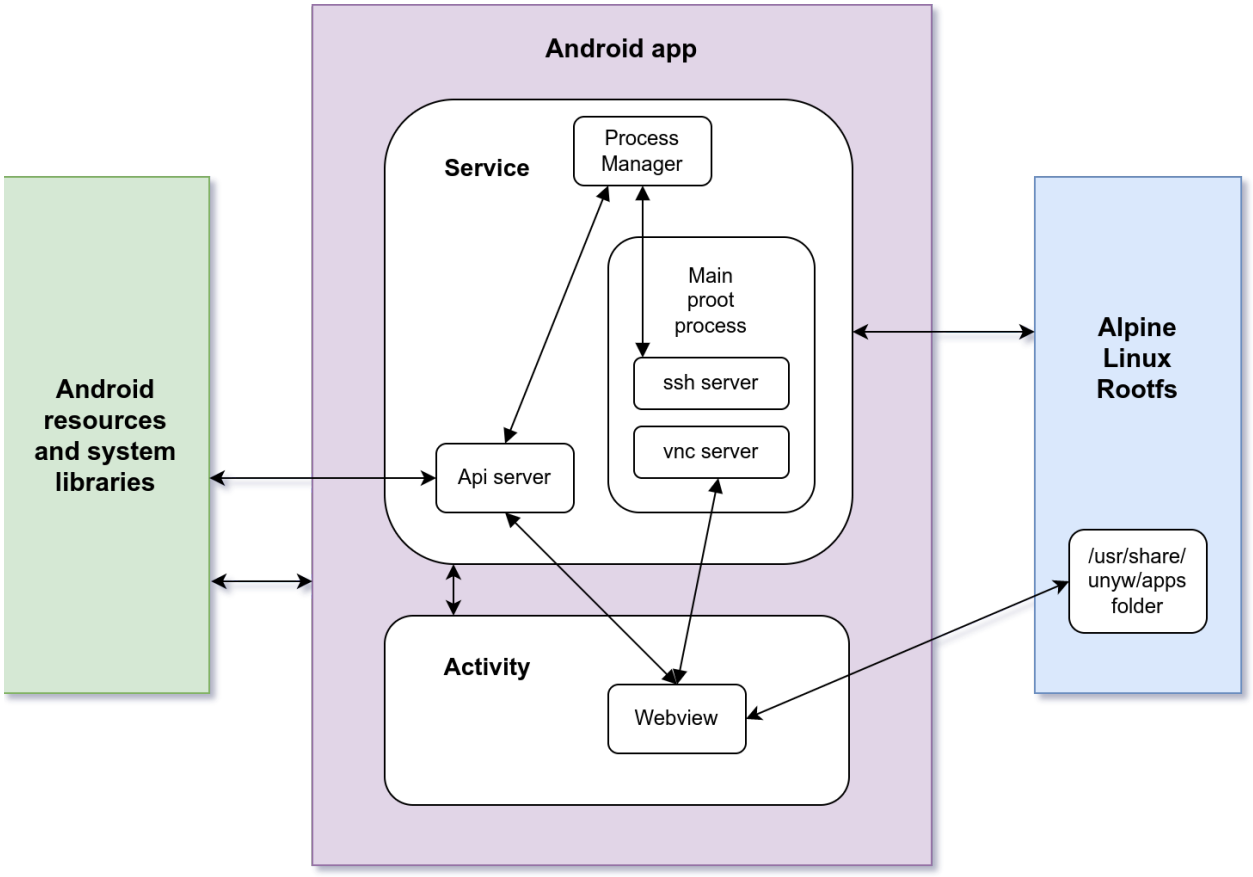

Figure 7: *Main components of the actual app*

4.4.1 Installation and launch of the app

In order for the app to function properly, PRoot and the rootfs must be installed correctly among its internal files.

As mentioned in 4.2, it is not possible to include the rootfs within the apk due to space limitations. Therefore, it must be downloaded from the github.com/unyw/android-rootfs repository.

The installation process is carried out by the service, as it needs to continue even when the app is in the background, and it consists of three phases:

- Some files are copied from the apk package's internal files to the apk's private file folder, including two scripts (install.sh and start.sh), Busybox (a collection of Linux command-line utilities), PRoot, and the dynamic libraries used by PRoot.
- The rootfs is downloaded using an Android system library.
- The "install.sh" script is called, which, by appropriately calling Busybox, decompresses the rootfs and completes the installation with other small necessary operations, such as generating a random identifier within the /etc/machine-id file.

It would be more elegant to perform the entire installation process using only Android libraries or only the script instead of mixing the two. However, in practice, this is not possible:

- Busybox does not include support for downloading using HTTPS, as it would require certificate verification, which would force the heavy OpenSSL library to be embedded in the binary.
- Android system libraries do not include a way to decompress xz files and, in general, do not provide a reliable way to recursively decompress archives while maintaining unchanged authorization bits.

To confirm the installation was successful, a file called "unyw_installed.txt" is created, which will also serve to indicate that the rootfs has already been installed in subsequent instances. If this file is not present at the end of the process, the installation will be repeated from the beginning. There is no system for recovering partially completed installations, as it would not make sense since the process lasts less than five minutes in practice.

Throughout the installation process, if the activity is opened, it displays a simple HTML page that is programmed directly into the Kotlin code. The page includes a generic progress bar and collects logs generated by the service. These logs are transmitted using the binding, but they use a different channel than the one provided by the API, so they are independent from it.

Once the installation process is completed, the app proceeds with a normal startup. Unless specified otherwise, the activity opens a special wrapper called **unyw-app-home**, which corresponds to the URL `/apps/home/index.html`. From this web app, it is possible to access all the other apps or install new ones.

However, it is possible that the app did not open as a result of a simple user click, but rather a more specific Intent, provided by another app or a shortcut. In this case:

- If the activity is launched with an Intent that has a URL parameter, that URL is opened instead of the default one.
- If the activity is launched with an Intent that involves opening a file, such as Intent.ACTION_SEND or Intent.ACTION_VIEW, the service takes care of resolving the

Intent and copying the file to a location accessible to the application. The user is then asked which web app among those installed they want to use to open the file. Finally, the relevant URL is opened, with a special URL parameter added of the type *?unyw=DATA*, which indicates to the web app that should open the file whose specifications are contained within DATA.

When the service is not active, the activity intentionally waits a few seconds before proceeding with the URL loading, to give the service time to execute a script called "start.sh", which starts some necessary background processes for correct operation, including:

- An SSH server, to remotely access the system, useful for debugging purposes and used by the API server.
- Xvfb, x11vnc, and the bspwm window manager (which will be discussed more thoroughly in 4.4.3).

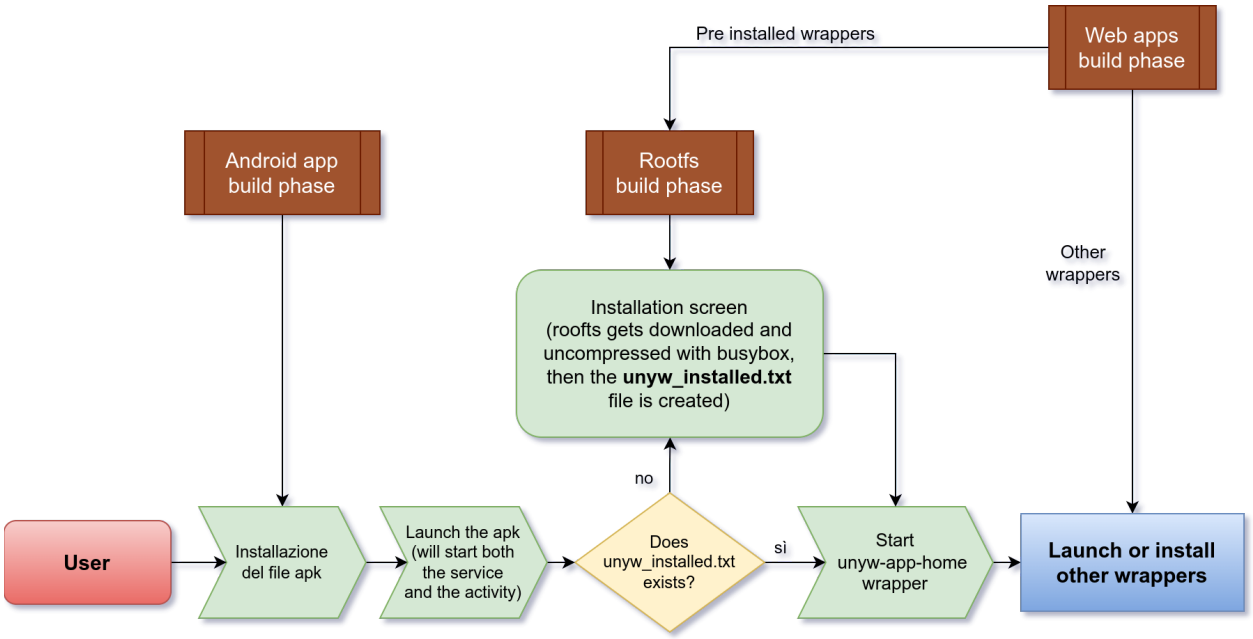

Figure 8: *Building, distributing, installing, and launching the app.*

4.4.2 Server API

As previously mentioned, in order to enable the use of Android functions, an API must be provided, which is a set of functions that can be called from outside using a predefined protocol. Since this API should be available to both wrappers and GNU/Linux programs, it was decided to implement it using the HTTP protocol. Specifically, a server was implemented using the Ktor library, which runs within a thread and is always reachable through port 12080.

The advantages of the HTTP protocol are that it is a universal request/reply protocol, stateless (does not require a connection to be established, each call is independent of the others), easily implementable, and monitorable.

The main disadvantages of this choice, however, can be summarized as follows.

- If external connections are not limited, an HTTP server can create a security hole in the device. This can be mitigated by only allowing connections from localhost and using an access token (see 4.6).
- The HTTP protocol, in general, adds heavy headers to communications, making it particularly inefficient for programs that need to make many calls **[37]**. This is not a problem in this case because none of the expected APIs need to be repeatedly called or require real-time response times. It should be noted, for completeness, that this protocol is still faster by an order of magnitude compared to the solution implemented by Termux-api: an HTTP call can take less than 120 milliseconds, while all Termux APIs require at least 3 or 4 seconds to execute, probably due to the call to am.
- The protocol is completely synchronous. This can be a problem: for example, it does not allow a wrapper to be notified through an event if the application is terminated. However, it should be considered that in this case, it would have been complicated to use an asynchronous protocol, both because my solution involves components that can be terminated and restarted continuously, and because not all contexts have the ability to properly handle asynchronous events (e.g., bash). In addition, some modules of the API itself will partially overcome this problem by providing callback mechanisms for events.

4.4.2.1 Approach based on predefined functions and modules

According to what was stated in 4.1, I decided to abstract access to Android libraries by creating an API with specific and predefined functions. For ease of implementation, these functions were grouped into modules, and all calls were structured with a fixed URL format:

http://localhost:12080/MODULE/FUNCTION?token=TOKEN

and a JSON formatted body (TOKEN is the access token, see 4.6). I chose to use only "post" type calls for simplicity. The choice of a single type of call is justified by the fact that this way, the function called is uniquely identified by only the names of the module and the function itself. The use of "post" instead of, for example, "get" is because of the absence of cache (which is absolutely detrimental in this case - we want each call to have an updated result) and the ability to send very long bodies.

I chose to always receive a response in JSON format, which will always contain a "status": "success" field if the request was successful. The choice of the JSON format has the intrinsic advantage of making the API robust to updates, allowing for the addition of fields in both requests and responses without breaking wrappers designed for previous versions. This format is not particularly suitable for use with GNU/Linux command-line programs, which usually prefer to use plain text. However, it is possible to use tools such as the jc command-line program to extract fields from a JSON with ease.

Modules can be grouped into two categories: modules that also work in the background ("unyw", "device", "file", "process", "job", and "notification") and modules that require the activity to be open and in the foreground ("webview" and "intent"). It is possible to use the functions of the "notification" module to alert the user that a background process has finished, through vibration, a toast message, or an actual notification, while the "webview" module allows GNU/Linux programs to interact with web apps if they are active.

```
POST REQUEST:
localhost:12080/process/run?token=debugapi
{
 "command": "echo Hello world!"
}

REPLY:
{
 "status": "success",
 "result": "Hello world!\n"
}
```

Code 2: *Request-reply example*

Modules and available endpoints
Unyw module: include functions related to Unyw itself /unyw/info: provides information about the app, such as a list of available APIs and a list of security tokens
Device module: includes functions for managing the device and its hardware /device/torch: to turn on/off the torch /device/battery: to get battery information
File module: includes functions for reading, writing, and listing files /file/read: to read a file as a string /file/write: to write a text file

/file/list: to get information about files inside a folder
Process module: functions for managing PRoot processes /process/run: to execute a process and get the execution result /process/dtach: to launch a process in the background /process/screen: to launch a process /process/list: to get a list of active background processes /process/kill: to terminate a background process
Job module: Manages actions scheduled by Android to be executed at a certain time or repeatedly /job/schedule: Allows creating a job and starting it /job/stop: Allows terminating a job. /job/list: Allows checking running jobs
Notification module: Includes functions to notify the user when the process is in the background /notification/toast: Displays a "toast" message (the black messages that disappear). /notification/vibrate:Vibrates the device /notification/show: Shows a notification in the application bar or modifies an existing notification /notification/list: Returns a list of created and currently displayed notifications
Webview module: Functions to interact with the webview; it only works if the activity is displayed on the screen /webview/info: Provides information about the webview, such as the running web app /webview/eval: Executes code inside the webview and returns the result /webview/event: Calls an event inside the webview (you may respond to received events using the standard window.addEventListener function)
Intent module: Includes a set of functions to interact with other apps installed within the device; it only works if the activity is displayed on the screen /intent/authenticate: Allows the user to authenticate with a PIN or fingerprint sensor /intent/photo: Opens the camera and asks to take a picture /intent/open: Opens a file present in the app's folders (e.g., a document) with the default system app. /intent/filepicker: asks the user to choose a file ***note:** technically, the authentication request is not classified as an "Intent" within the Android system library, due to Intents not being secure enough. However, since it essentially follows the same rules and requires the activity to be displayed to succeed, it seemed natural to include the function in this module and not in "device"

Discussing in detail the implementation, protocol, and specifications of the implemented functions would be redundant and beyond the scope of this thesis. The goal is to demonstrate how the API can coexist with GNU/Linux programs, not to showcase the potential of the API itself. Therefore, we will limit ourselves to making general considerations about why some architectural choices of interest were made. It should be noted that the presented application is a prototype designed to

demonstrate the potential of the expressed ideas, not to be fully functional for a user. For example, only a restricted subset of device modules has been developed; before an actual release of the app, it would be necessary to implement functions for managing sensors like accelerometer and GPS or for managing the camera in the background. Nevertheless, an attempt has been made to create a plausible grouping of modules even in view of a future distribution.

In most cases, we developed without providing redundant functions. However, there are some exceptions.

- The file module is technically unnecessary since such operations can be performed with Linux command-line programs invoked by the process module. However, in practice, file operations are frequent, so it makes sense to optimize them. Additionally, it is not always easy to perform them with GNU/Linux programs and obtain a useful result for a machine (e.g., ls does not have a standard output: it is a command useful for a human but risky if used within an automatic process).
- /webview/event can also be executed using /webview/eval. In this case, we wanted to encourage a specific pattern for the interaction between GNU/Linux programs and web apps.

All functions were also programmed to be "stateless" (not requiring previous calls) and robust to sudden closures of the calling entity. Most functions, for example, are exhausted with the call itself. In cases where the calls lead to the creation of resources that survive the end of the call (as happens, for example, with /process/dtach, which creates a background process, and /notification/show), these resources always have an identifying code that cannot be lost because it is always possible to request a list of active codes (for example, a web app that is closed and reopened can check if it had already launched a process in the past using /process/list).

To make it easier to call the API within Alpine Linux, a script called unyw-api (**Code 3**) has been created, which internally calls the Linux software wget to make an HTTP call.

```
#!/bin/sh
wget -O - -q --post-data="$2" \
 localhost:12080"$1"?token=`cat /tmp/UNYW_TOKEN_API`
```

Code 3: *unyw-api script for connecting to the API using wget*

4.4.2.2 Alternative approach based on Java Reflection

In this paragraph, I will describe an approach that is in contrast with the one presented in the previous paragraph. Instead of abstracting the functionality of the system library, it makes it available in its entirety as it is. This is possible thanks to the Java Reflection system library **[38]**, which allows for meta-programming and composing and executing arbitrary Java code during runtime. With Java Reflection, it is possible, for example, to verify which classes are present within a library, invoke a function by knowing its name as a string, import a library, and instantiate a class on-the-fly **[71]**.

Technically, by providing a way to interface with Java Reflection, access to all available Java functions on the device is provided, just as giving access to the eval function in JavaScript/Python provides a way to invoke any function of that language.

The use of Java Reflection for this purpose also has practical application in real-world scenarios, for example, it is the basis of the NativeScript framework **[39]** for developing Android apps using JavaScript.

This approach has two main advantages:

- the first is that it allows for a complete mapping of all available functionality within Android with minimal effort;
- the second is that this mapping automatically extends to functionalities released after the Android app's release, making it even unnecessary to update the APK and limiting the update to web apps only under appropriate conditions.

It is easy to argue that the aforementioned advantage is relative: a well-designed API should not require frequent updates. Additionally, there is no real benefit to updating web apps, which are numerous, compared to updating the Android app, which is just one. Furthermore, it is worth noting that apps distributed outside the Google Play Store can be programmed to self-update without external intervention.

On the contrary, this choice brings problems that are difficult to solve.

- First and foremost, it forces web app programmers to deal, directly or indirectly, with the complexity of Android, which often requires precise and laborious protocols to access resources. For example, requesting biometric authentication requires 30 lines of Kotlin code, while thanks to the abstraction of the API, web app programmers only need 2. Allowing the execution of arbitrary code generally eliminates any possibility of easily intervening on the code itself, such as filling the gaps in Android or smoothing out differences between different versions of the operating system.
- The environment provided by Reflection requires an Android device (real or emulated) to be executed or tested effectively; on the other hand, a more abstract API can be tested using alternative implementations. This means, for example, that much lighter automated tests can be designed. Moreover, the API could be fully reimplemented for platforms other than Android, allowing web apps to run on other operating systems, such as the Windows Subsystem for Linux.
- There is also a subtler problem with how to manage memory within the Java Virtual Machine. The Java language implements the object-oriented paradigm, which means that to perform actions, classes need to be instantiated, and these classes are generally not

serializable (they cannot be transferred to JavaScript) and need to be kept in memory for subsequent operations. The classical solution to this problem (which is also implemented by NativeScript) is the use of a "pool," a virtual container in which all these resources are placed, each with an identifying token. The web app can then request to create a resource, receive the token in return, and use that token to interface with the resource in the future. However, this way, the scope in which the objects were created is lost, and this prevents the garbage collector (the automatic memory management mechanism of Java) from eliminating them once they have fulfilled their function. Specifically, the loss of the token within the web app means that there is no way to interface with the resource, which remains active within the service even after the web app has been closed. Because of the large number of objects that Java programmers must reason with, this introduces a series of indiscriminate non-stateless elements into the API, which the web app must carefully manage, with the constant risk of clogging the pool and overloading memory or, conversely, attempting to access no longer existing resources (referred to as "dangling reference"). For example, consider a web app that checks the disk every second to see if files have been added to a folder. If object deletion is not implemented correctly, the user could quickly end up with tens of thousands of instances of the "File" object in memory and would be forced to restart the device. The situation is further complicated when considering that the API should be accessible to GNU/Linux background programs, and web apps may be terminated at any time, as the user could close the activity at any time. The solutions available in the literature are not applicable because they would require changes to the JavaScript interpreter (as NativeScript does, which, however, is in the simplistic condition of an API accessible from a single activity that does not have a service).

In addition, such an API would require many more calls than the current one and this, in theory, could highlight the performance limitations of the HTTP protocol. For these reasons, in the implementation provided, we have chosen to focus solely on the approach presented in paragraph 4.4.2. However, there is nothing preventing us from adopting a hybrid approach in the future by implementing a module within the API to access Android libraries via Java Reflection. This would allow us to maintain all the benefits of an abstract API while providing web app programmers with an alternative way to access Android libraries in the absence of an API module that implements that specific function. To solve the problem of memory objects, we could consider dividing the pool into "sessions" so that the service can eliminate inactive sessions. This is, in any case, a much broader problem (considering that the study of pointer management is the basis of entire branches of computer science **[68]**), which would deserve treatment in a separate thesis and, therefore, will not be further elaborated, remaining an open problem.

4.4.3 "Process" module and process manager

A crucial part of the presented project is the ability for web apps to interact with the GNU/Linux system by launching its programs. To achieve this, a process manager has been developed within the service. In an attempt to obtain the best possible performance, two interchangeable implementations have been created: in the first case, the Java ProcessBuilder class was used directly, while in the second, we connected to the internal SSH server using the JSch library. The advantage of using SSH is that it is not necessary to load PRoot every time, as shown in the figure. The first implementation seems to present occasional performance drops after the app has not been used for a few minutes, but we have not been able to consistently reproduce or measure this phenomenon. Apart from this, the results obtained with the two implementations are comparable.

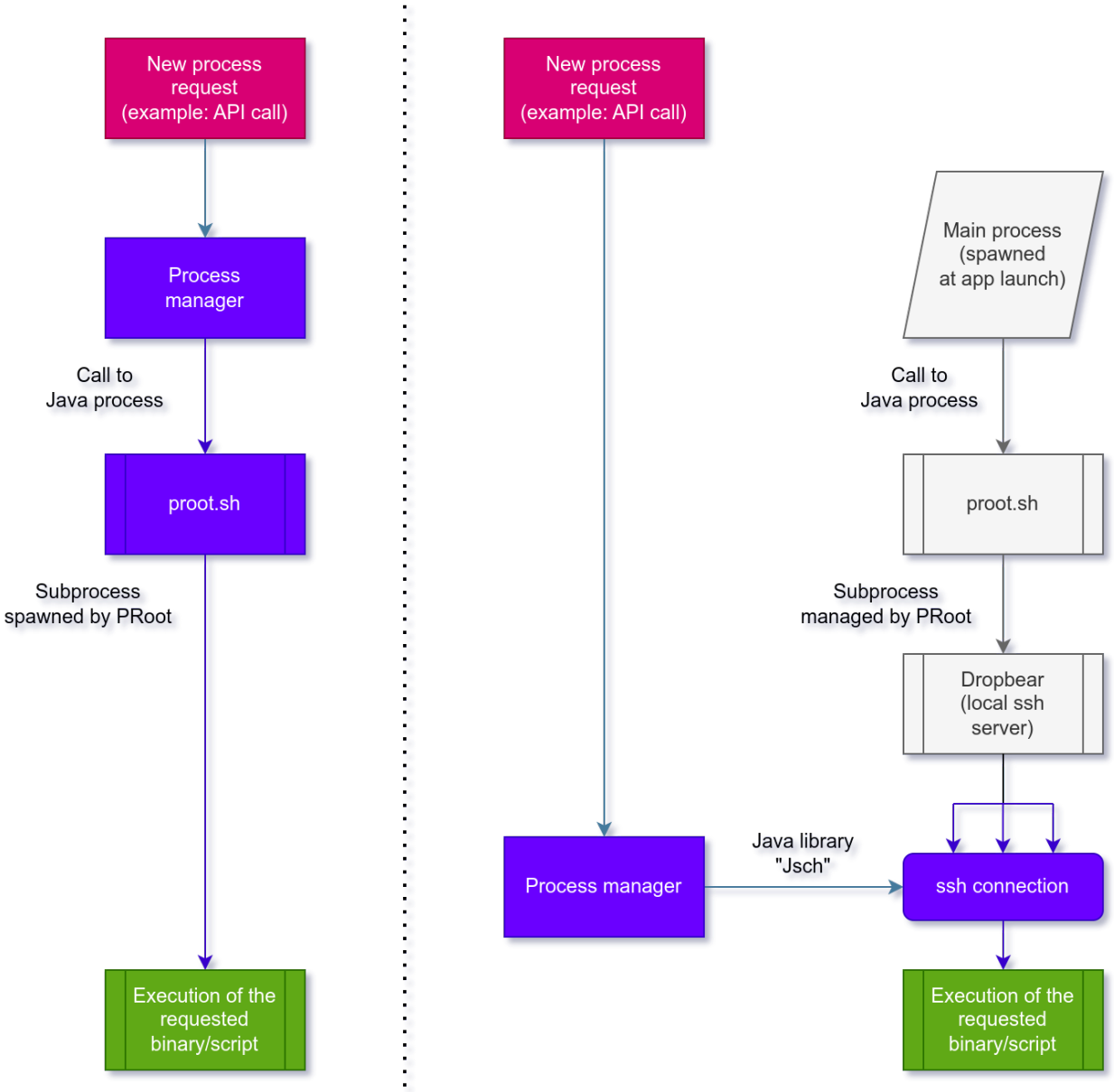

Figure 9: *Comparison between the two implementations of the process manager (on the right, the one using a local SSH server).*

To enable web apps to access the process manager, I created a "process" module within the API. I chose to provide two types of access to GNU/Linux processes:

- The first, obtainable using `/process/run`, is intended for quick processes that need to return a result quickly and therefore have a fixed timeout of 60 seconds to complete their execution, or they will be terminated by the system. For example, a web app could call `/process/run` with body `{"command": "ls /folder"}` to get the list of files in the `/folder` directory.
- The second, which can be invoked using `/process/dtach`, is intended for programs that, once launched, do not need to return a result but remain active in the background. These processes are instantiated using the Linux Dtach software **[40]** for running background programs, which allows them to be controlled in the future using a Unix socket. Dtach does not include a mechanism for terminating launched processes, so the API saves its process ID in a file in the same folder as the Unix socket each time it launches a process, allowing it to use the kill command later.

A `/process/screen` function is also provided, which allows you to start a process with Dtach and have it displayed in the foreground within the virtual screen provided by Xvfb, so that it can be viewed in the web app via noVNC.js.

For performance reasons, I chose to use a single instance of Xvfb (which is started with the service) and manage graphical processes with the bspwm window manager. I chose bspwm because it can be fully controlled from the command line. With the right configuration, bspwm allows multiple graphical interface programs to coexist, keeping the desired one as the only visible and full-screen process.

Different web apps may want to display apps within screens of different sizes; for example, a web app may decide to show a terminal so that it occupies half the screen. In theory, recent versions of Xvfb include an extension that allows the size of the emulated screen to be changed **[41]**. In practice, however, this operation has proven to be too slow. Therefore, I chose to keep the dimensions of Xvfb fixed and equal to the device's screen, and to adjust the available space for GNU/Linux programs using bspwm's "margin" function as needed.

4.5 Web app (“Wrappers”)

I decided not to impose any particular constraints on the structure of web apps. They are to be considered as normal websites and can be created using any existing technology/framework for building web apps. The API server itself is agnostic: as explained in section 4.4.2, it is sufficient to make an HTTP call to *localhost:12080* to access it.

As previously mentioned, I chose to distribute such apps using the package manager of Alpine Linux. This allows developers to specify among the package dependencies all the programs they want to be installed together with the web app (for example, a wrapper that wants to use LibreOffice can indicate "libreoffice" among its dependencies, and is guaranteed that LibreOffice will be available at runtime). I also decided to enforce the convention that wrappers be called **unyw-app-APPNAME** and be installed within the rootfs in the /usr/share/unyw/apps/APPNAME folder. Furthermore, each web app must have an entry point called "index.html" (as per the tradition of websites) and an icon called "icon.png".

In realizing the presented prototype, I chose to use the modern approach to writing web apps, which involves using Node.js and Npm (Node Package Manager) to manage JavaScript libraries, Rollup.js for code optimization, and a framework that simplifies the declaration of dynamic HTML interfaces. Specifically, I selected the Svelte framework, a minimal framework particularly suitable for writing single-page web apps (SPA) and which has gained a lot of popularity in recent years **[42]**. To facilitate the writing of wrappers, I created scripts and JavaScript libraries that automate some repetitive operations; these scripts are described in section 4.5.1, while the wrappers themselves are discussed in section 4.5.2.

4.5.1 Development environment

Creating packages for Alpine is particularly complex and poorly documented (for example, the official wiki contains outdated information and even spelling errors **[43]**). Therefore, I decided to create a script called "build.js" that would perform the necessary operations automatically for all web apps. This script takes care of:

- Downloading a rootfs to the computer if it is not running Alpine Linux and installing all necessary dependencies using PRoot. This is necessary because some software required to generate Alpine Linux packages (such as the one that digitally signs packages) is only available in Alpine Linux repositories.
- Generating a public/private key pair for digital package signing if it does not exist. The private key is stored locally on the computer, while the public key is inserted into the repository.
- Creating a manifest compatible with the Alpine Linux package manager for each wrapper from the information in the "package.json" file (the manifest that Node.js projects have).
- Creating and digitally signing packages. Alpine Linux requires that 4 versions of the packages be provided, one for each architecture, even if the packages do not contain architecture-specific binary software.
- Generating the APKINDEX.tar.gz files, which serve the Alpine package manager to list the packages available in the repository.

In addition, I created two JavaScript libraries, which can be installed and imported using Node.js + Rollup. These libraries make it possible to take advantage of the full potential of the API with just a few lines of code:

- **@unyw/api**, which makes it easier to make HTTP calls to the API server, in particular, it automates the token request operation (see 4.6). I chose to design this library to take advantage of all the features of the JavaScript language, especially JavaScript Promises, a model for managing asynchronous code introduced in the JavaScript ES2015 revision and currently a standard. This library also includes a mechanism useful for debugging, to connect to an API server outside of localhost: this allows you to launch and test a web app on your computer by connecting it to a remote device.
- **@unyw/ui**: a collection of Svelte graphic components created according to the principles of Material Design. In addition to generic components such as navigation bars, buttons, icons, ... components already integrated with @unyw/api have been created, including VncViewer, a component that allows you to display the screen transmitted by x11vnc via noVNC.js.

There is also a component called Keyboardbar, which includes a set of extra keys such as CTRL, ALT, TAB, ... which, when clicked, will be sent directly to the Xvfb screen via noVNC.js, thus expanding the user's keyboard.

The use of these libraries, combined with the efficiency of Svelte and JavaScript in general, makes the source code required to write web apps extremely concise. This allows you to create wrappers with very few lines of code, as shown by the code below (Code 4), which, in just 32 lines, contains the entire source code necessary to start chromium and display it with noVNC.js, including logic and styles.

```
<script>
 import {Keyboardbar, VncViewer} from '@unyw/ui'
 import Unyw from '@unyw/api'
  const socket = `chromium-app`

 Unyw().then( ({process}) => process.screen({
       socket,
       command: 'chromium-browser --no-sandbox'
   }))
</script>

<main>
 <VncViewer {socket}/>
 <Keyboardbar/>
</main>

<style>
 :global(body){
   margin: 0px;
   padding: 0px;
   font-family: 'Roboto', 'Arial', sans-serif;
   --color-primary: #000;
   -webkit-tap-highlight-color: transparent;
   overflow: hidden;
 }

 main {
   display: flex;
   flex-direction: column;
   height: 100%;
 }
</style>
```

***Code 4: unyw-app-chromium** wrapper*

4.5.2 List of developed demo wrappers

unyw-app-home: This web app allows users to launch other wrappers and interface with the package manager of Alpine Linux, enabling them to add external repositories and install new wrappers. It also lets them view information about the currently running unyw version and modify app settings.

unyw-app-xterm: This app allows users to launch and use the xterm terminal, which is displayed using noVNC.js. It can handle multiple instances of the program, allowing users to run several processes simultaneously.

unyw-app-chromium: This app allows users to launch and use the Chromium browser, which is displayed using noVNC.js. It was created as a minimal wrapper example.

unyw-app-vscode-server: This app launches the server version of the popular code editor VSCode, displaying the interface through the webview. Please note that currently, this web app does not work on ARM64 devices due to an internal package issue with code-server that is being resolved.

unyw-app-libreoffice-converter: This web app allows users to open a document using the Android file selector, convert it to a desired format using the converter provided by LibreOffice, and share the result by opening it with the preferred app. The goal is to demonstrate a more advanced interaction between GNU/Linux programs and the Android API.

4.6 Example wrapper development: **unyw-app-libreoffice-converter**

In this section, I will describe the development process of a wrapper, taking the example of the unyw-app-libreoffice-converter wrapper. The result will then be published in the existing repo-main repository.

As explained in section 4.5.2, the unyw-app-libreoffice-converter wrapper aims to provide a graphical interface to the "converter" function included in LibreOffice.

To start the project, it's convenient to begin with the base provided in the template-app repository, cloning it using degit **[69]**, a software distributed through Npm (Node Package Manager) that allows for easier cloning of template repositories.

```
# Use repo-main as working dir
cd repo-main

# Use degit to clone a copy of the repository
# It will crate a "unyw-app-libreoffice-converter" folder
# inside repo-main/packages
npx degit unyw/template-app packages/unyw-app-libreoffice-converter
```

Code 5: *setting up template-app*

Si modifica quindi il file manifest unyw-app-libreoffice-converter/package.json, inserendo il nome dell'app e aggiungendo LibreOffice fra le dipendenze.

```
{
 "name": "unyw-app-libreoffice-converter",
 ...
 "unyw-pkginfo": {
   "depend": [
     "unyw", "libreoffice"
   ]
 },
 ...
}
```

Codice 6: *Updating package.json*

I also modify other information, such as the web app's description and author, and add the icon file unyw-app-libreoffice/app/icon.png.

Next, I define the code and user interface within the file unyw-app-libreoffice-converter/src/App.svelte by adding various web app components:

- A "Open file" button that, when clicked, calls the API /intent/filepicker. This API asks the user to select a file from their device and saves the chosen file as /storage/unyw/libreoffice-converter/document.
- A "select" component that allows the user to choose the format to convert the file into.
- A "Convert" button that starts the xterm application with /process/screen and executes the script that converts the selected file and opens the converted file (see Code 7).
- A VNC client, created using noVNC.js, that displays xterm while the script is running and shows any errors. This component is imported from the @unyw/ui library, which is already included in the template.

```
# Perform the document conversion to pdf
libreoffice --headless --convert-to pdf                  \
  /storage/unyw/libreoffice-converter/document           \
  --outdir /storage/unyw/libreoffice-converter;

# Apre il risultato
unyw-api /intent/open \
 '{
    "file":"document.pdf",
    "folder": "libreoffice-converter",
    "mimetype":"application/pdf"
  }';
```

Code 7: *script to perform the document conversion PDF*

```
<script>
 import {Topbar, VncViewer} from '@unyw/ui'
 import Unyw from '@unyw/api'

 let format = 'pdf'
 const socket = `libreoffice-convert-app`

 const bashStr = str => `'${str.replaceAll(`'`, `'"'"'`)}'`

 const mimetypes = {
   "pdf": "application/pdf",
   "odt": "application/vnd.oasis.opendocument.text",
   "doc": "application/msword",
   "docx":
"application/vnd.openxmlformats-officedocument.wordprocessingml",
   "html": "text/html",
   "txt": "text/plain",
 }

</script>

<main>
 <Topbar>
   <h3>LibreOffice converter</h3>
 </Topbar>
 <div style="flex: 1 1 0; text-align:center; margin: 40px;">
   <div><button on:click={ () => Unyw().then( ({intent}) => {
     intent.filepicker({
       file: 'document',
       mimetype: 'application/*',
       folder: 'libreoffice-converter'
     })
   })}>Open file</button></div>
   <div><select bind:value={format}>
     <option value="pdf">pdf</option>
     <option value="odt">odt</option>
     <option value="doc">doc</option>
     <option value="docx">docx</option>
     <option value="html">html</option>
     <option value="txt">txt</option>
   </select></div>
   <div><button on:click={ () => {
     Unyw().then( async({process, intent}) => {
       await process.kill({
```

```
          socket
        })

        await process.screen({
          socket,
          command: `xterm -e  sh -c ${bashStr(`echo "Converting
document..."; libreoffice --headless --convert-to ${format}
/storage/unyw/libreoffice-converter/document --outdir
/storage/unyw/libreoffice-converter; `+
          `unyw-api /intent/open ${bashStr(`{"file":"document.${format}",
"folder": "libreoffice-converter", "mimetype":"${mimetypes[format]}"}`)};
echo ""; echo "Done!"; read`)}`
        })
      })
    }}>Convert</button></div>
 </div>
 <h3 style="margin: 4px 16px;"><b><u>Logs:</u></b></h3>
 <div style="height: 40vh; box-sizing: border-box; margin: 10px">
   <VncViewer {socket}/>
 </div>
</main>

<style>
 :global(body){
   --color-primary: rgb(22, 131, 0);
   overflow: hidden;
 }

 main {
   display: flex;
   flex-direction: column;
   height: 100%;
   align-content: center;
 }

 button {
   margin: 5px 0;
   padding: 10px 20px;
   border-radius: 20px;
 }
</style>
```

***Code 8: unyw-app-libreoffice-converter** wrapper*

Once I have finished writing the wrapper, I can prepare it for distribution by running the "build.js" script (see section 4.5.1) of repo-main, and then publishing it through Github Pages **[35]**, as shown in **Code 9**.

The wrapper will immediately appear on the Home screen of the Unyw app, and users can install it without any further steps.

```
# Use repo-main as working dir
cd repo-main

# Create the Alpine package starting from the wrapper
node build.js packages/unyw-app-libreoffice-converter

# Publish on Github
git add .
git commit -m "New libreoffice-converter package"
git push
```

Codice 9: *Preparing and publishing* ***unyw-app-libreoffice-converter***

4.7 Notes on app security

It is not the purpose of this thesis to carry out a thorough review of the security of my solution. Therefore, I will limit myself to providing general information on what the most important critical points are and propose a possible strategy to solve them.

All programs executed within the app are subject to Android security measures. Therefore, the execution of the programs itself does not pose a risk to the security of the system.

However, the app uses network protocols to communicate internally, in particular:

- the API is provided as an HTTP server on port 12080;
- Xvfb creates a screen accessible at port 6000;
- VNC and SSH are still based on the tcp network protocol, on ports 12081 and 12082 respectively

It should be noted that among these communications there is no mechanism that allows sending files to the Webview that need to be loaded into it: this mechanism is internal to Java and does not open network ports.

The presence of these open ports could expose the system to security vulnerabilities because a program outside the app could connect to the listed channels to take control of and/or damage the app and the resources it has access to. This is particularly serious because the app, as it needs to provide a diverse API, at installation must request authorization to access numerous resources.

To solve the problem, first of all, it is necessary to ensure that none of the listed ports are accessible outside of localhost. My solution can easily comply with this requirement because in all the cases listed, there is an option to limit connections to those coming from the device only ("localhost"); for example, for Xvbf, it is sufficient to start the process with --no-listen tcp. An advantage of localhost

connections is that they are entirely managed by the kernel, so it is not possible to intercept them if the device is not tampered with. This makes the transmission of sensitive data secure.

Avoiding external connections is not enough to guarantee security, as localhost channels could be improperly used if malicious apps are present within the device. For example, an app without authorization to write files to disk could exploit the API to do so (this is referred to as the Confused Deputy Problem **[44]**). Therefore, it is necessary for all protocols used for internal communication to be inaccessible to other apps on the device.

1) GNU/Linux programs executed by the app must ensure they communicate with the app's servers and not with another app on the device that may intend to steal the received information. This is easily achievable because all communications occur through IP protocol ports, which, once reserved, belong to a process until it releases them or is interrupted. It is sufficient for the app, at startup, to initialize the various services by requesting the use of fixed ports. If all ports are available, they will belong to the app until it is terminated. If some of these ports are not available (for example, because another app is attempting to intercept them or is using them), my app should avoid starting up, explaining the situation to the user. In this case, there would be no processes attempting to connect.
2) The various servers offered by the app must ensure that only requests from other components of the app itself are satisfied. This can be achieved through the use of tokens, i.e., randomly generated temporary passwords that allow access to a resource. In the prototype, at the service's startup, it generates a token and writes it inside the file /tmp/UNYW_TOKEN_API. As this file is within the app's private files, only the app's programs can access it (the webview does not have native access to it, but as it is an internal protocol in Java, it is possible to safely make the file available by remapping it to the localhost:12079/UNYW_TOKEN_API.json URL). At that point, the programs are asked to read the file and send its result as a parameter every time they need to make an API call. Once the API is secure, it is possible to use the API itself to distribute other tokens (such as VNC and SSH passwords). Xvfb does not allow the use of a password, but this is not a significant issue because the maximum achievable by embedding it is to start unnecessary graphical interface programs.

In practice, it is impossible to achieve better and more granular results without radically changing the app's structure. This is because, as previously mentioned, PRoot does not actually allow imposing permissions on the filesystem: a malicious web app could replace system programs with its infected version, thus rooting itself in the system.

4.8 Comparative tests

To validate my solution in terms of functionality and performance, I used the same tests described in section 3.3. The results obtained have also been reported in section 3.3 for easier comparison.

Available features (see 3.3.1):

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Unyw (my solution)	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	0	Yes	Yes

Therefore, my solution has the most features among those considered. Of the two missing features:

- feature 6 (support for hardware acceleration) is currently a problem that is practically impossible to solve;
- feature 9 (compatibility with .iso files, kernel included) cannot be obtained by OS-Level solutions like PRoot, which, as discussed in 3.2, have the prerogative of avoiding kernel emulation.

Execution of fibonacci.sh (see 3.3.2.2):

2:03	2:01	1:57	1:58	2:00	2:01	1:58	2:01	2:02	1:58	(Avg) 2:00

The unyw app achieves comparable or superior performance to all other solutions considered. In particular, the result is identical to that obtained by other solutions using PRoot and Alpine Linux.

App size (see 3.3.3 e 3.3.3.2):

App name	**Apk size (Mb)**	**User data (Mb)**	**Total (Mb)**
Unyw (my solution, without Pulseaudio support)	10	48	58
Unyw (my solution, Pulseaudio included)	15	71	86

Despite no optimization being made on the apk file size, especially regarding Pulseaudio, Unyw only requires 86Mb to be installed, making it lighter than all previously analyzed solutions. This can be explained both by the choice of Alpine Linux and by the use of wrappers combined with bspwm, which avoids the installation of a heavier window manager.

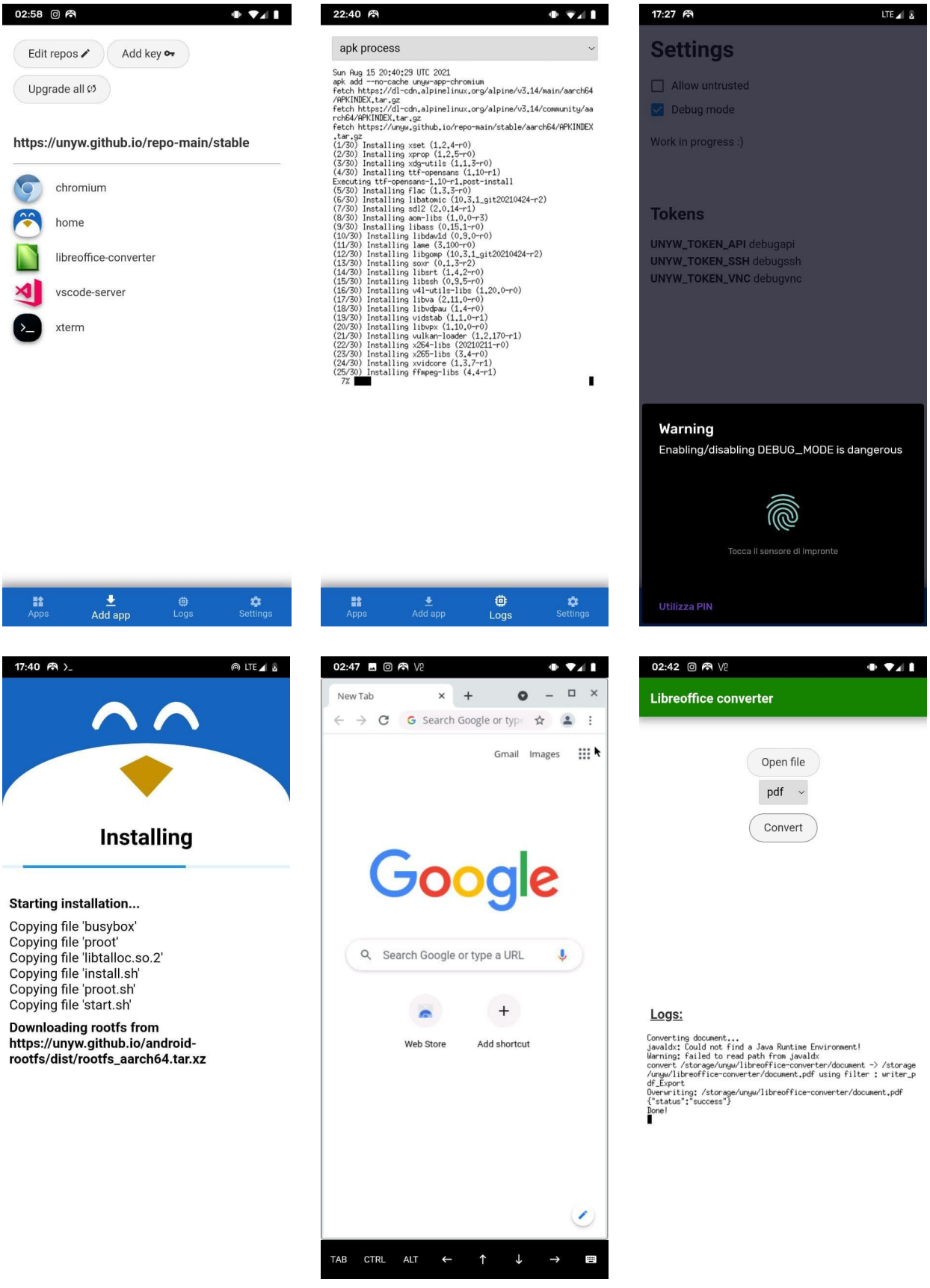

Figure 10: *Screenshots of Unyw:* ***unyw-app-home*** *(top), install page (bottom left)* ***unyw-app-chromium*** *(bottom center) and* ***unyw-app-libreoffice-converter*** *(bottom right)*

5. Conclusions

5.1 Results obtained

The purpose of this thesis was to provide a way to run GNU/Linux programs on the Android system. This goal has been achieved as I have managed to program a prototype that meets all the required specifications:

- It can be easily and quickly installed.
- It offers comparable performance to the most efficient solutions analyzed, requires the least amount of disk space, and has the highest number of features among those considered.
- It allows the execution of GNU/Linux programs both from the command line and through a graphical interface, without the need for external apps.
- It is expandable (and tools and libraries have already been created to assist in writing extensions, as seen in 4.5.1).
- It enables the creation of new interfaces to GNU/Linux programs with just a few lines of code.

Furthermore, it has been argued that in almost all cases, the unattained functionalities such as distribution through the Google Play Store and the implementation of hardware acceleration are practically impossible to achieve, with the exception of the approach presented in 4.4.2.1, which remains an open problem.

5.2 Future improvements

As I have mentioned before, the prototype developed is incomplete, as its only purpose is to demonstrate the concepts expressed in this thesis. Therefore, the project can be considered concluded only at a theoretical level. Before any eventual distribution, the following steps would be necessary:

Creating numerous wrappers in addition to those example ones presented in 4.5.2;

Formalizing a standard for the API and implementing all the missing functionalities, including an extensive mapping of the hardware components with which Android can interact and adding more extensive modules in general;

Creating a detailed documentation of the project, allowing third parties to implement their own web apps;

Testing the obtained app on various devices and Android versions, correcting any inconsistencies found. In particular, it is necessary to investigate more thoroughly how keyboard inputs are handled: the current solution exclusively uses the opportunities offered by HTML, whose standard, however, does not manage elements such as autocorrect in a unique way, generating different results from device to device. To ensure a homogeneous experience, it may be necessary to delegate the management of such events to the Kotlin code of the activity.

Resolving the issue presented in 4.4.2.1, in order to integrate a module that allows us to take advantage of Java Reflection.

Bibliography

[1] “Android Kernel Features - eLinux.org.” https://elinux.org/Android_Kernel_Features (accessed Aug. 22, 2021).

[2] A. Ltd., “Linux Kernel,” *Arm Developer*. https://developer.arm.com/tools-and-software/open-source-software/linux-kernel/energy-aware-scheduling (accessed Aug. 22, 2021).

[3] “Filesystems in the Linux kernel — The Linux Kernel documentation.” https://www.kernel.org/doc/html/latest/filesystems/index.html (accessed Aug. 22, 2021).

[4] Contributors to Wikimedia projects, “Ptrace,” *Wikipedia*, Jun. 08, 2021. https://en.wikipedia.org/wiki/Ptrace (accessed Aug. 22, 2021).

[5] “Android ABIs ,” *Android Developers*. https://developer.android.com/ndk/guides/abis (accessed Aug. 22, 2021).

[6] “SupportedArchitectures - Debian Wiki.” https://wiki.debian.org/SupportedArchitectures (accessed Aug. 22, 2021).

[7] Contributors to Wikimedia projects, “Bionic (software),” *Wikipedia*, May 31, 2021. https://en.wikipedia.org/wiki/Bionic_(software) (accessed Aug. 22, 2021).

[8] “LLVMpipe — The Mesa 3D Graphics Library latest documentation.” https://docs.mesa3d.org/drivers/llvmpipe.html (accessed Aug. 22, 2021).

[9] “Window manager - ArchWiki.” https://wiki.archlinux.org/title/window_manager (accessed Aug. 22, 2021).

[10] nwrkbiz, “GitHub - nwrkbiz/android-xserver: Maintaining the original project to make it work again with new Android versions.,” *GitHub*. https://github.com/nwrkbiz/android-xserver (accessed Aug. 22, 2021).

[11] pelya, “GitHub - pelya/commandergenius: Port of SDL library and several games to the Android OS.,” *GitHub*. https://github.com/pelya/commandergenius (accessed Aug. 22, 2021).

[12] Contributors to Wikimedia projects, “Virtual Network Computing,” *Wikipedia*, Apr. 22, 2021. https://en.wikipedia.org/wiki/Virtual_Network_Computing (accessed Aug. 22, 2021).

[13] pelya, “What about opengl support? · Issue #45 · pelya/xserver-xsdl,” *GitHub*. https://github.com/pelya/xserver-xsdl/issues/45#issuecomment-127371602 (accessed Aug. 22, 2021).

[14] “React Native Internals.” https://www.reactnative.guide/3-react-native-internals/3.1-react-native-internals.html (accessed Aug. 22, 2021).

[15] “Device and Network Abuse - Play Console Help.” https://support.google.com/googleplay/android-developer/answer/9888379 (accessed Aug. 22, 2021).

[16] “Background Execution Limits ,” *Android Developers*. https://developer.android.com/about/versions/oreo/background (accessed Aug. 22, 2021).

[17] golem, “What does the @ symbol denote in the beginning of a unix domain socket path in Linux?,” *Unix & Linux Stack Exchange*. https://unix.stackexchange.com/questions/206386/what-does-the-symbol-denote-in-the-beginning-of-a-unix-domain-socket-path-in-l (accessed Aug. 22, 2021).

[18] “TCP loopback connection vs Unix Domain Socket performance,” *NewbeDEV*. https://newbedev.com/tcp-loopback-connection-vs-unix-domain-socket-performance (accessed Aug. 22, 2021).

[19] “Ubuntu flavours,” *Ubuntu*. https://ubuntu.com/download/flavours (accessed Aug. 22, 2021).

[20] Contributors to Wikimedia projects, “Hypervisor,” *Wikipedia*, Jul. 23, 2021. https://en.wikipedia.org/wiki/Hypervisor (accessed Aug. 22, 2021).

[21] “How many x86 instructions are there?,” *The ryg blog*, Aug. 25, 2016. https://fgiesen.wordpress.com/2016/08/25/how-many-x86-instructions-are-there/ (accessed Aug. 22, 2021).

[22] L. Lisen, “Bochs.” https://play.google.com/store/apps/details?id=net.sourceforge.bochs (accessed Aug. 22, 2021).

[23] “Difference between Emulation and Virtualization - javatpoint,” *www.javatpoint.com*. https://www.javatpoint.com/emulation-vs-virtualization (accessed Aug. 22, 2021).

[24] “PRoot — chroot, mount --bind, and binfmt_misc without privilege/setup.” https://proot-me.github.io/ (accessed Aug. 22, 2021).

[25] J. M. Mbongue, D. T. Kwadjo, and C. Bobda, “Performance Exploration of Virtualization Systems,” *arXiv.org*, Mar. 12, 2021. https://arxiv.org/abs/2103.07092 (accessed Aug. 22, 2021).

[26] “Package Management - Termux Wiki.” https://wiki.termux.com/wiki/Package_Management (accessed Aug. 22, 2021).

[27] “Differences from Linux - Termux Wiki.” https://wiki.termux.com/wiki/Differences_from_Linux (accessed Aug. 22, 2021).

[28] termux, “x11-packages/packages at master · termux/x11-packages,” *GitHub*. https://github.com/termux/x11-packages/tree/master/packages (accessed Aug. 22, 2021).

[29] termux, “[README] Bintray repositories are down now,” *GitHub*. https://github.com/termux/termux-packages/issues/6726 (accessed Aug. 22, 2021).

[30] “AnLinux.” https://f-droid.org/en/packages/exa.lnx.a/ (accessed Aug. 22, 2021).

[31] AndronixApp, “GitHub - AndronixApp/AndronixOrigin” *GitHub*. https://github.com/AndronixApp/AndronixOrigin (accessed Aug. 22, 2021).

[32] CypherpunkArmory, “GitHub - CypherpunkArmory/UserLAnd: Main UserLAnd Repository,” *GitHub*. https://github.com/CypherpunkArmory/UserLAnd (accessed Aug. 23, 2021).

[33] pelya, “No audio · Issue #35 · pelya/debian-noroot,” *GitHub*. https://github.com/pelya/debian-noroot/issues/35 (accessed Aug. 23, 2021).

[34] “Migrating to WebView in Android 4.4 ,” *Android Developers*. https://developer.android.com/guide/webapps/migrating (accessed Aug. 23, 2021).

[35] “GitHub Pages,” *GitHub Pages*. https://pages.github.com/ (accessed Aug. 23, 2021).

[36] M. Stapelberg, “Linux package managers are slow (2019) - Michael Stapelberg.” https://michael.stapelberg.ch/posts/2019-08-17-linux-package-managers-are-slow/ (accessed Aug. 23, 2021).

[37] “Analysis of HTTP Performance Problems.” https://www.w3.org/Protocols/HTTP-NG/http-prob.html (accessed Aug. 23, 2021).

[38] “Using Java Reflection.” https://www.oracle.com/technical-resources/articles/java/javareflection.html (accessed Aug. 23, 2021).

[39] Contributors to Wikimedia projects, “NativeScript,” *Wikipedia*, Aug. 06, 2021. https://en.wikipedia.org/wiki/NativeScript (accessed Aug. 23, 2021).

[40] “dtach(1) - Linux man page.” https://linux.die.net/man/1/dtach (accessed Aug. 23, 2021).

[41] Erik, “Resizing an Xvfb display,” *Stack Overflow*. https://stackoverflow.com/questions/6356169/resizing-an-xvfb-display (accessed Aug. 23, 2021).

[42] “Stack Overflow Developer Survey 2021,” *Stack Overflow*. https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks (accessed Aug. 23, 2021).

[43] “Alpine package format - Alpine Linux.” https://wiki.alpinelinux.org/wiki/Alpine_package_format (accessed Aug. 23, 2021).

[44] Contributors to Wikimedia projects, “Confused deputy problem,” *Wikipedia*, Jul. 04, 2021. https://en.wikipedia.org/wiki/Confused_deputy_problem (accessed Aug. 23, 2021).

[45] “Comparison of C/POSIX standard library implementations for Linux.” http://www.etalabs.net/compare_libcs.html (accessed Aug. 23, 2021).

[46] dront78, “GitHub - dront78/PulseDroid: Android network audio player,” *GitHub*. https://github.com/dront78/PulseDroid (accessed Aug. 29, 2021).

[47] “Modules – PulseAudio.” https://www.freedesktop.org/wiki/Software/PulseAudio/Documentation/User/Modules/ (accessed Aug. 29, 2021).

[48] Anbox, “Anbox,” *Android in a Box*. https://anbox.io/ (accessed Sep. 16, 2021).

[49] “Enterprise Open Source and Linux,” *Ubuntu*. https://ubuntu.com/ (accessed Sep. 16, 2021).

[50] “Debian -- The Universal Operating System.” https://www.debian.org/ (accessed Sep. 16, 2021).

[51] “Arch Linux.” https://archlinux.org/ (accessed Sep. 16, 2021).

[52] “index,” *Alpine Linux*. https://www.alpinelinux.org/ (accessed Sep. 16, 2021).

[53] “Tiny Core Linux, Micro Core Linux, 12MB Linux GUI Desktop, Live, Frugal, Extendable.” http://tinycorelinux.net/ (accessed Sep. 16, 2021).

[54] libhybris, “GitHub - libhybris/libhybris: Hybris is a solution that commits hybris, by allowing us to use bionic-based HW adaptations in glibc systems,” *GitHub*. https://github.com/libhybris/libhybris (accessed Sep. 17, 2021).

[55] “GL4ES - The OpenGL driver for GLES Hardware,” *GL4ES - The OpenGL driver for GLES Hardware*. http://ptitseb.github.io/gl4es/ (accessed Sep. 17, 2021).

[56]“Snapcraft - Snaps are universal Linux packages,” *Snapcraft*. https://snapcraft.io/ (accessed Sep. 17, 2021).

[57] “Flatpak—the future of application distribution.” https://flatpak.org/ (accessed Sep. 17, 2021).

[58] “AppImage,” *Linux apps that run anywhere*. https://appimage.org/ (accessed Sep. 17, 2021).

[59] L. B., “lBochs PC Emulator,” *Apps on Google Play*. https://play.google.com/store/apps/details?id=lb.myapp.lbochs (accessed Sep. 17, 2021).

[60] limboemu, “Home · limboemu/limbo Wiki,” *GitHub*. https://github.com/limboemu/limbo/wiki (accessed Sep. 17, 2021).

[61] “QEMU.” https://www.qemu.org/ (accessed Sep. 17, 2021).

[62] “KVM.” https://www.linux-kvm.org/page/Main_Page (accessed Sep. 17, 2021).

[63] “Termux.” https://termux.com/ (accessed Sep. 17, 2021).

[64] termux, “GitHub - termux/proot-distro: An utility for managing installations of the Linux distributions in Termux.,” *GitHub*. https://github.com/termux/proot-distro (accessed Sep. 17, 2021).

[65] CypherpunkArmory, “GitHub - CypherpunkArmory/UserLAnd: Main UserLAnd Repository,” *GitHub*. https://github.com/CypherpunkArmory/UserLAnd (accessed Sep. 17, 2021).

[66] pelya, “GitHub - pelya/debian-noroot: Debian running on Android, without rooting your phone,” *GitHub*. https://github.com/pelya/debian-noroot (accessed Sep. 17, 2021).

[67] “r/termux - CPU Battle : Termux vs Linux proot,” *reddit*. https://www.reddit.com/r/termux/comments/hvzp9x/cpu_battle_termux_vs_linux_proot/ (accessed Sep. 17, 2021).

[68] Contributors to Wikimedia projects, “Garbage collection (computer science),” *Wikipedia*, Jun. 29, 2021. https://en.wikipedia.org/wiki/Garbage_collection_(computer_science) (accessed Sep. 17, 2021).

[69] Rich-Harris, “GitHub - Rich-Harris/degit: Straightforward project scaffolding,” GitHub. https://github.com/Rich-Harris/degit (accessed Sep. 18, 2021).

[70] “noVNC.” https://novnc.com/info.html (accessed Sep. 18, 2021).

[71] C. ai progetti Wikimedia, “Live coding,” Wikipedia, Sep. 07, 2021. https://it.wikipedia.org/wiki/Live_coding (accessed Sep. 18, 2021).