Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Oliveira, Lucas Cardoso Coelho Alves de.
Comparative study of techniques for detecting emulators on Android devices / Lucas Cardoso Coelho Alves de Oliveira. - Recife, 2022.
24 : il., tab.

Orientador(a): Leopoldo Motta Teixeira
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2022.

1. Emuladores. 2. Android. 3. Fraude. 4. Detecção. 5. Mobile. I. Teixeira, Leopoldo Motta. (Orientação). II. Título.

000 CDD (22.ed.)

Comparative study of techniques for detecting emulators on Android devices

Lucas Cardoso[1], **Leopoldo Teixeira**[1]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Recife – PE – Brazil

lccao@cin.ufpe.br, lmt@cin.ufpe.br

Abstract. *The growth in the mobile ecosystem provide a fertile ground for malicious activities. Malicious users might use mobile emulators to commit fraudulent acts. Such practice has already caused serious damage to app service providers. Therefore, the need for effectively detecting whether an app is running in an emulator must be considered. There is a set of proposed techniques by open source projects and academic papers. In this work, we perform a comparison analysis of some of the existing techniques. Our results show that none of the proposed techniques is effective to all emulators, with some being unable to detect the emulators they were designed to detect. We have also found that the current Nox emulator version for macOS is not detected by any technique.*

1. Introduction

The mobile ecosystem has been steadily growing in the last decades, turning into a huge industry. The combined user spending in the App Store and Google Play is set to reach almost 233 billion U.S. dollars by 2026 [Statista 2022]. This provides a fertile environment for malicious activities [Dimjašević et al. 2016]. These activities incur in major costs to this growing industry and to society overall. For instance, fraud is a major problem, with American consumers losing $5.8 billion to it in 2021 [Iacurci 2022].

Malicious users might use mobile emulators to commit fraudulent acts. Such practice has already caused serious damage to app service providers, being linked to ”Mobile ad fraud” or ”Mobile Click Fraud Attack (MCFA)”, this fraud happens because the fees paid by advertisers are only effective when users click on the advertising page, while malicious users use simulators to simulate multiple terminals to click in order to defraud advertisers' fees [Lin et al. 2019]. Contrary to what happens on PCs, where practical use cases have developed, virtualization is not broadly available on consumer mobile platforms [Vidas and Christin 2014].

Another problem present on emulators are the potential threats this unique architecture may expose its users to. There are some flaws on communication channel authentication, permission control, and open interfaces, that attackers could exploit to bypass Android security mechanisms. Some of the found vulnerabilities include: the abuse of the emulators input method (IME) text channel, that allows a malicious Android app located in the same emulator to inject any text to Android; a race condition attack that hijacks the app installation process and finally install an arbitrary app; and the possibility of a malicious app gaining system privilege (e.g., root) without user consent, which will

change the app into “God Mode” and enable the adversary to do nearly anything within the emulator [Xu et al. 2021].

Therefore, the need for effectively detecting whether an app is running in the emulator must be considered [Lin et al. 2019]. A few detection techniques have been previously developed. Some are based on differences of: behavior, performance, hardware and software components [Vidas and Christin 2014], memory usage, context switching and vectorization [Jang et al. 2019], and even machine learning [Guerra-Manzanares et al. 2019]. This work seeks to make a comparison analysis between different techniques, taking into consideration the changes in Android’s policies regarding permissions and the current version of popular emulators.

We have also analyzed open source repositories found on GitHub. Those repository tend to focus on techniques that observe differences in behavior, specially on Android’s API return values that have fixed values on emulators. Three were identified as containing relevant detection techniques [Framgia 2016, Gingo 2017, Arakawa 2019].

The app developed for data collection, the collected data from emulators, and the Jupyter notebook used in this paper, are available on a GitHub repository [Cardoso 2022]. The emulators used for testing were Android Studio’s emulator, Bluestacks, Genymotion and Nox, the rationale this selection is explained in Section 2.2.

The results found indicate that there is no silver bullet capable of detecting all emulators. As a matter of fact, none of the proposed techniques were able to detect one of the tested emulators. The techniques that were able to detect more emulators were Android Build Characteristics, described in Section 2.3.1, File Detection, described in Section 2.3.2 and OpenGL, described in Section 2.3.9, all being capable to detect 2 of the tested emulators. Of those techniques, Android Build Characteristics had a high number of false positives, therefore is not an advised method. Vectorization technique described in Section 2.3.10 has shown great promise when dealing with real devices of supported architectures but did not support any of the tested emulators.

2. Background

This section provides a brief explanation of the Android ecosystem. Moreover, it explains and classifies the methods proposed in previous works, as well as in the open source repositories, describing known problems and limitations.

2.1. Android

Android is an open source operating system for mobile devices and a corresponding open source project led by Google [AOSP 2022]. Figure 1 shows its overall architecture. It consists of the following components: the application framework, referring to the set of frameworks available to app developers; the Binder Inter-Process Communication (IPC), responsible for enabling the application framework to cross process boundaries and call into the Android system services code; System services, responsible for bridging application framework APIs with the underlying hardware; Hardware abstraction layer (HAL), a standard interface for hardware vendors to implement, which enables Android to be agnostic about lower-level driver implementations; Linux kernel, the underlying kernel it is a version of the Linux kernel with a few special additions such as Low Memory Killer [AOSP-Architecture 2022].

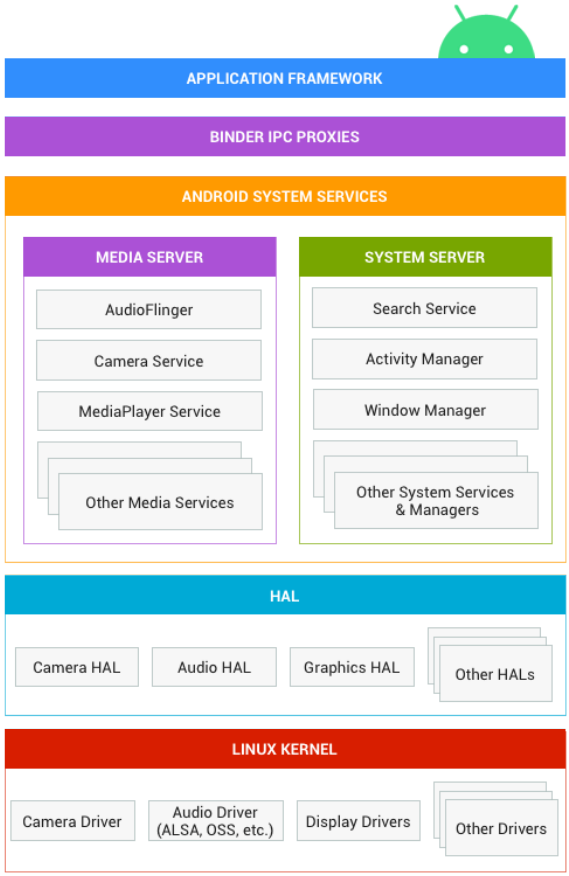

Figure 1. Android architecture [AOSP-Architecture 2022]

Each application runs on it's own sandbox, since Android takes advantage of the Linux user-based protection to identify and isolate app resources. This isolates apps from each other and protects apps and the system from malicious apps. To do this, Android assigns a unique user ID (UID) to each Android application and runs it in its own process. The kernel enforces security between apps and the system at the process level through standard Linux facilities such as user and group IDs that are assigned to apps. By default, apps can't interact with each other and have limited access to the OS. If app A tries to do something malicious, such as read application B's data or dial the phone without permission, it's prevented from doing so because it doesn't have the appropriate default user privileges. The sandbox is simple, auditable, and based on decades-old UNIX-style user separation of processes and file permissions [Android 2022d].

Android apps can be written using Kotlin, Java, and C++ languages. The Android SDK tools compile the code along with any data and resource files into an APK or an Android App Bundle. An *Android package*, which is an archive file with an `.apk` suffix, contains the contents of an Android app that are required at runtime and it is the file that Android-powered devices use to install the app [Android 2022c]. Every app project must have an `AndroidManifest.xml` file (with precisely that name) at the root of the project source set. The manifest file describes essential information about the app to the Android build tools, the Android operating system, and Google Play. It contains, among other things, some components of the app, the permissions, hardware and software

features the app needs [Android 2022b].

2.1.1. Permissions on Android

App permissions are an Android feature for supporting the user privacy by protecting access to sensitive data, such as system state or contact information. It also serves to restrict some actions that can impact privacy or energy consumption, like connecting to a paired device and recording audio. Android categorizes permissions into different types, including install-time permissions, runtime permissions, and special permissions [Android 2022n].

Install-time permissions give the app limited access to restricted data, allowing it to perform restricted actions that minimally affect the system or other apps. These permissions must be declared by the app developer and are automatically granted upon app installation. These permissions are displayed to the user by the app store [Android 2022n].

Runtime permissions, also known as dangerous permissions, allow apps to have additional access to restricted data. They might also allow apps to perform restricted actions that more substantially affect the system and other apps. As the name suggests, these permissions must be requested and granted during runtime, besides being declared in the app manifest file. Requesting such a permission evokes a system prompt that the user must interact with, granting or revoking the permission. These permissions are often tied to accessing sensitive and private data such as location and contact information [Android 2022n].

Special permissions are related to particular app operations, only the Android platform and Original Equipment Manufacturers (OEMs) have access to such permissions [Android 2022n]. Since these are not available to regular Android apps, these are not very relevant to this work.

2.2. Android Emulators

The Android emulators can be divided into two categories, depending on the focus of its users, either developers or end-users [Xu et al. 2021]. In this work we seek to make a general assessment on those two categories, so two emulators of each category were chosen.

With regards to emulators geared toward developers, we use Genymotion and the Android Studio emulator. This type of emulator is mainly leveraged to assist app development or perform automated app testing [Xu et al. 2021]. They were chosen due to their availability and ease of use. Different from the developer-oriented emulators, the end-user emulators aim to enhance the user experience with Android apps [Xu et al. 2021]. The chosen emulators to analyze were Bluestacks and Nox due to their popularity and availability on macOS [Xu et al. 2021].

Android Studio and Nox emulator are QEMU emulators [Lin et al. 2019]. QEMU is a generic open source machine emulator and virtualizer, it supports the use of Kernel-based Virtual Machine (KVM) modules [QEMU 2020], Lin et al. found that the KVM used for Nox is VirtualBox [Lin et al. 2019]. The virtual machine used by Bluestacks

is also VirtualBox [Xu et al. 2021]. With respect to Genymotion, it is not clear but its documentation implies that it uses them as well [Genymotion 2022]. All emulators were executed with macOS as the host OS.

2.3. Identified Techniques

Since there is a broad range of techniques, this work focuses mainly on the proposed methods that were implemented by existing open source projects and some selected papers regarding difference in behavior detection. The open source projects selected were developed by Arakawa [Arakawa 2019], Framgia [Framgia 2016] and Gingo [Gingo 2017].Vidas and Christin [Vidas and Christin 2014] behavior detection and vectorization proposed by Jang et al. [Jang et al. 2019] were selected from papers. They were further divided into 10 distinct strategies. The track record and performance hit of every technique were further analysed and scrutinized on Section 4.

The first selected heuristic is Android Build Characteristics, which is related to a set of constants system properties common on Android Build. Section 2.3.1 further explains the heuristic and the values used. Section 2.3.2 describes the second selected technique, File Detection. This strategy consists of finding known emulator files within the app sandbox.

The third heuristic consists of accessing Android's Telephony Characteristic, checking for known emulator characteristics. This techniques are further scrutinized on Section 2.3.3. Another selected approach is about the detection of QEMU drivers. It relies on the behavior of a common open source emulator known as QEMU, it is described on Section 2.3.4.

IP detection is about using common binaries to check the current device IP connection, but, as described in Section 2.3.5, it seems to have been deprecated. Section 2.3.6, named Package Names, describes the check on installed packages for known emulator packages.

Section 2.3.7 refers to a method that looks for Available Activities. Section 2.3.8 describes the technique that searches Services. OpenGL Render detection is detailed on Section 2.3.9. Finally, Section 2.3.10 describes the Vectorization strategy proposed by Jang et al. [Jang et al. 2019].

2.3.1. Android Build Characteristics

These are system properties that contain information about the current build of the Android device like it's model (Motorola G5, Samsung S22, etc), manufacturer (Motorola, Samsung, etc) and other static characteristics. They are set automatically during the build phase of the Android image, with information set on a `.sysprop` file [Android 2022g]. They are usually generated by manufacturers, but on emulators, since they have to build their own image, the properties are often set either with default values or with custom specific values.

Since they are set on the ROM image, and have a low performance hit. All open source projects use it in some way. Vidas and Christin also provided a few

values [Framgia 2016, Gingo 2017, Arakawa 2019, Vidas and Christin 2014]. They use Android's helper class `Build`, that access the underlying set system properties statically.

The first heuristic considered checks for specific values in Build values [Framgia 2016]. The value check is usually for default values, like `golfish`, `generic` and `google_sdk`. They also check for usual VirtualBox virtual machine, Nox, Genymotion and Droid4x emulators. The values can be found on Table 1.

Table 1. Build Heuristic 1

Build Characteristic	Relation	Value
FINGERPRINT	*starts with*	generic
MODEL	*contains*	google_sdk, Emulator, Android SDK built for x86
MODEL	*lowercase contains*	droid4x
MANUFACTURER	*contains*	Genymotion
HARDWARE	*equals*	goldfish, vbox86
HARDWARE	*lowercase contains*	nox
PRODUCT	*equals*	sdk, google_sdk, sdk_x86, vbox86p
PRODUCT	*lowercase contains*	nox
BOARD	*lowercase contains*	nox
SERIAL	*lowercase contains*	nox
BRAND	*starts with*	generic
DEVICE	*starts with*	generic

The second heuristic adds to the first by considering three specific build characteristics [Arakawa 2019, Gingo 2017]. They have similar checks to the ones proposed by Framgia [Framgia 2016]. But add support for Andy and TianTian emulators. Since they also only mark the emulator with 3 or more evidences, they have a significantly higher number of checks on many different properties. These values can be found on Table 2.

Finally, also similar to what is conducted by Framgia [Framgia 2016], a third technique lists heuristics that mark them as emulators proposed by Vidas and Christin [Vidas and Christin 2014]. They seem to be focused solely on default values and the ones from Android Studio emulator, like `generic`, `sdk`, `unkwnown` and `goldfish`. Table 3 shows those values.

2.3.2. File Detection

This heuristic checks the existence of known files present on emulators. They simply try to see if a file for a given path exists in the app sandbox using java's `File` class, for example checking for the existence of a `x86.prop` file. Both Framgia and Arakawa [Framgia 2016, Arakawa 2019] implemented a method of this detection with the same files paths. Their detection differs only on the listed x86 files, on them, Framgia [Framgia 2016] has a stricter approach, marking it has an emulator only if five or more system properties associated with QEMU are found. It is not clear why that is the

Table 2. Build Heuristic 2

Build Characteristic	Relation	Value
FINGERPRINT	*contains*	generic/sdk/generic, generic_x86/sdk_x86/generic_x86, Andy, ttVM_Hdragon, generic_x86_64, generic/google_sdk/generic, vbox86p, generic/vbox86p/vbox86p
MODEL	*equals*	sdk, google_sdk, Android SDK built for x86_64, Android SDK built for x86
MODEL	*contains*	Droid4X, TiantianVM, Andy
MANUFACTURER	*equals*	unknown, Genymotion
MANUFACTURER	*contains*	Andy, MIT, nox, TiantianVM
HARDWARE	*equals*	goldfish, vbox86
HARDWARE	*contains*	nox, ttVM_x86
PRODUCT	*contains*	sdk, Andy, ttVM_Hdragon, google_sdk, Droid4X, nox, sdk_x86, sdk_google, vbox86p
BRAND	*equals*	generic, generic_x86, TTVM
BRAND	*contains*	Andy
DEVICE	*contains*	generic, generic_x86, Andy, ttVM_Hdragon, Droid4X, nox, generic_x86_64, vbox86p

case. For each emulator there is a set of known files that are checked, Table 4 contains a set of the checked emulator files.

Gingo only has one check for a Bluestacks folder, it is calculated during runtime, using the set external folder ending with `windows/BstSharedFolder`.

2.3.3. Telephony Characteristics

These characteristics refer to the behavior of the telephony services and states. Since Android devices are not require to provide this functionality, it is necessary to check if this feature is available for utilization. Android provides a easy way to check for it using the `PackageManager` class, but this restricts this check to the availability of this feature.

In addition to the feature check, Framgia [Framgia 2016] also checks for the `READ_PHONE_STATE` permission, but there are a few caveats in that. Starting on Android API level 23, dangerous permissions must be requested at runtime [Android 2022o] and this particular permission is considered dangerous [Android 2022j]. One of the used data is the phone number, and starting on Android API version 26, it requires the `READ_PHONE_NUMBERS` dangerous permission [Android 2022j]. But this highlights a limitation on the approach from Framgia, since it only collects data with the `READ_PHONE_STATE` permission, but not all data collected requires these permissions, unnecessarily curbing the other check. Table 6 contains the permission related to each method used in the heuristic.

The values checked are on Table 5. The values are: `line1Number`, that

Table 3. Build Heuristic 3

Build Characteristic	Relation	Value
FINGERPRINT	*starts with*	generic
MODEL	*equals*	sdk
MANUFACTURER	*equals*	unknown
HARDWARE	*equals*	goldfish
PRODUCT	*equals*	sdk
BRAND	*equals*	generic
DEVICE	*equals*	generic
BOARD	*equals*	unknown
ID	*equals*	FRF91
RADIO	*equals*	unknown
SERIAL	*equals*	null
TAGS	*equals*	test-keys
USER	*equals*	android-build

Table 4. Emulator Files checks

Emulator	Files
Genymotion	*/dev/socket/genyd, /dev/socket/baseband_genyd*
Nox	*fstab.nox, init.nox.rc, ueventd.nox.rc*
Andy	*fstab.andy, ueventd.andy.rc*
x86	*ueventd.android_x86.rc,x86.prop, ueventd.ttVM_x86.rc, init.ttVM_x86.rc, fstab.ttVM_x86, fstab.vbox86, init.vbox86.rc, ueventd.vbox86.rc*
QEMU	*/dev/socket/qemud, /dev/qemu_pipe*

represents the device phone number; `deviceId`, that is an unique device ID; `subscriberId` an unique subscriber ID, for example, the IMSI for a GSM phone; and `networkOperatorName`, the alphabetic name of current registered operator [Android 2022q].

Vidas and Christin [Vidas and Christin 2014] also provided a few characteristics, but they were less assertive about the values provided, because some of them were very situational. For example, checking for a T-Mobile carrier, which is a valid carrier in the USA, nevertheless, when found outside the USA, it may be a good indicative of emulator usage [Vidas and Christin 2014]. Because of this circumstances, only the values they considered guaranteed to be emulators were taken into account. And those were limited to evaluating the equality of the subscriber id to `310260000000000` and the voice mail number to `15552175049`.

2.3.4. QEMU drivers

Framgia and Arakawa [Framgia 2016, Arakawa 2019] have a identical approach, checking for QEMU drivers. They check the content on `/proc/tty/drivers` and `/proc/cpuinfo`, looking specifically for the `goldfish` substring. Goldfish

Table 5. Telephony Characteristics Values

TelephonyManager Field	Values
line1Number	*15555215554, 15555215556, 15555215558, 15555215560, 15555215562, 15555215564, 15555215566, 15555215568, 15555215570, 15555215572, 15555215574, 15555215576, 15555215578, 15555215580, 15555215582, 15555215584*
deviceId	*000000000000000, e21833235b6eef10, 012345678912345*
subscriberId	*310260000000000*
networkOperatorName	*android*

Table 6. Telephony Related Permissions

TelephonyManager method	Permissions Required
line1Number	*READ_SMS, READ_PHONE_NUMBERS*
deviceId	*Deprecated after Android API Level 26*
networkOperatorName	*None*
subscriberId	*READ_PRIVILEGED_PHONE_STATE*
voiceMailNumber	*READ_PHONE_STATE*

is a virtual hardware platform used to run some emulated Android systems under QEMU [Turner 2014].

About those paths, the tty abbreviation came from an old abbreviation of teletypewriter and was originally associated only with the physical or virtual terminal connection to a Unix machine. Nowadays a tty device means any serial port style device. The `/proc/tty/drivers` file determines what kind of tty drivers are currently loaded in the kernel and which tty devices are currently present [Corbet et al. 2005]. The `proc/cpuinfo` file is a virtual file that identifies the type of processor used by the system [RedHat 2022].

2.3.5. IP

This method was proposed by Framgia [Framgia 2016], and consists of using the `netcfg` Linux network connection tool [ArchLinux 2022], but it has been deprecated at least since Android API version 23 [Google 2016]. This data does not seem to be relevant anymore since less than 3% devices are below this Android version [statcounter 2022].

2.3.6. Package Names

The Android installed packages in a device are available through the `PackageManager` class [Android 2022m]. Arakawa [Arakawa 2019] implemented a heuristic checking the strings of this list of installed packages names for a set of prefixes, and a single specific package. But there is a privacy issue, since the list of installed packages is

considered a personal and sensitive data. When the app targets Android API level 30 or higher, the system filters the apps, being necessary to either add a query for the desired packages on the Android Manifest or to add the permission `QUERY_ALL_PACKAGES` [Android 2022l]. This is becoming more relevant as Google is set to enforce this as the minimum target of all existing apps on the Google Play Store from November 2022 onwards, and every new app or update is already required [Google 2022]. Table 7 contains the checks used by this heuristics.

Table 7. Package Name Checks

Package Relation	Package Name
starts with	*com.vphone.*
starts with	*com.bignox.*
starts with	*com.nox.mopen.app*
starts with	*me.haima.*
starts with	*com.bluestacks*
starts with	*cn.itools.**
starts with	*com.kop.*
starts with	*com.kaopu.*
starts with	*com.microvirt.*
starts with	*com.bignox.app*
equals	*com.google.android.launcher.layouts.genymotion*

*This heuristic also checks if *Build.PRODUCT* starts with *iToolsAVM*

2.3.7. Available Activities

The `Activity` class is a main component of Android Apps. They are design to accommodate the necessity apps may face to have different entry points to the application. Most apps contain multiple screens, which means they are comprised of multiple activities. These activities work together to form a cohesive experience to the user. They also have the ability to call for other applications activities, one of the ways to do that is to use an `Intent` [Android 2022i].

An `Intent` is a messaging object you can use to request an action from another *app component* [Android 2022h]. In this context, an app component refers to an entry point through which the system or a user can enter your app [Android 2022c]. There are two types of `Intent`, *explicit* and *implicit*. Explicit ones must provide a specific package or component name, whereas implicit specify an action and corresponding data [Android 2022h].

An explanation on how an implicit intent is delivered through the system to start another activity can be found on Figure 2. [1] Activity A creates an `Intent` with an action description and passes it to `startActivity()`. [2] The Android System searches all apps for an intent filter that matches the intent. When a match is found, [3] the system starts the matching activity (Activity B) by invoking its `onCreate()` method and passing the `Intent` object. [Android 2022h]

In this heuristic, proposed by Arakawa, the detection consists of querying the

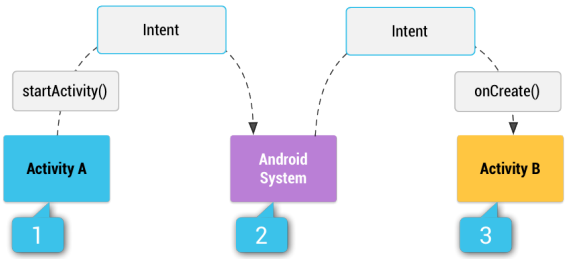

Figure 2. Launching Activity using Intent [Android 2022h]

system for action *ACTION_MAIN* and category *CATEGORY_LAUNCHER*, and checking if any of the activities start with `com.bluestacks.`. What this does is searching for the possible entry points of every application installed in the app.

2.3.8. Services

A `Service` is an application component that can perform long-running operations in the background [Android 2022p]. Arakawa proposes the collection of 20 running services and checks for service names starting with `com.bluestacks.`. The problem with this detection is that, as of Android 26, it is not possible to collect other application services [Android 2022a], making the effectiveness of this technique very limited.

2.3.9. OpenGL Render

Android includes support for high performance 2D and 3D graphics with the Open Graphics Library (OpenGL), specifically, the OpenGL ES (GLES) API [Android 2022k]. It works as an state machine, so applications can call its functions to set its state, which in turn determines the final appearance of its primitive types in a framebuffer [Martz 2006]. To create GLES contexts and provide a windowing system for GLES renderings, Android uses the EGL library. GLES calls render textured polygons, while EGL calls put renderings on screens [Android 2022f].

Gingo proposes a technique that checks the render for substrings *Bluestacks* and *Translator*. But there is a problem in Gingo implementation, it assumes there is a running context, which is necessary to access the OpenGL render. There is no documentation or indication whether this is due to that context being present on an emulator or it is necessary to create one before this check, so both ways were tested.

2.3.10. Vectorization

Vectorization is a CPU technology that supports calculating multiple data with a single instruction. Memory access alignment issue stems from the incapability of earlier (and current) CPUs accessing the cache with byte granularity. For example, some 32-bit CPU architectures have a 30-bit addressing line for fetching the memory. Due to the lack of 2 bits, such CPU can access the memory only if the target memory address is multiple of 4 (100 in binary). For such reason, if the CPU wants to access a memory address that is not

a multiple of 4, the CPU fetches memory twice and re-assembles the memory contents. Vectorization instructions of Intel and ARM do not support unaligned memory access at the hardware level (even with kernel modification for handling alignment feature), thus raising an application-aware fault. [Jang et al. 2019]

In general, the kernel is capable of handling unaligned memory access. However, high-performance vectorization operations such as Intel SIMD and ARM NEON are guaranteed to raise fault upon unaligned access regardless of kernel configuration. Software emulators, on the other hand, do not have to suffer from such issues because any memory access is ultimately reconstructed with multiple combinations of operations at a software level. [Jang et al. 2019]

Jang et al. [Jang et al. 2019] proposed a technique in which first it is installed a fault-handler to catch the hardware fault signal induced by unaligned vectorization, then it deliberately triggers a vectors misalignment. If the running environment is emulated, nothing happens upon such memory access. However, in the real hardware-based environment, the CPU immediately raises the fault signal and invokes the callback handler. [Jang et al. 2019]

Since this implementation interacts directly with the system using low level languages, it is necessary to use a Java Native Interface (JNI) for the C++ and assembly codes, being available through a Native Development Kit (NDK). The native code contained in such libraries runs outside the Java virtual machine directly on the processor of the smartphone or emulator [Spreitzenbarth et al. 2013]. An issue with the provided code is that to trigger the misaligned vectorization it is necessary to use machine code in assembly. Therefore, it is necessary to provide code for every supported architecture, and there is no support on the code for x86 instructions and the used emulators tested ran on this architecture, so it did not provide a solid detection method as it is.

3. Methodology

In this work, we evaluate existing techniques for detecting emulator usage in Android. Figure 3 shows the methodology we followed. The first step consisted of a literature analysis, trying to find related works in detecting emulators within an app during runtime. The main work we found related to that is the work of Vidas and Christin [Vidas and Christin 2014], which compares software and hardware analysis.

Jang et al. [Jang et al. 2019] presented some ways to rethink those detection techniques. They proposed three possible techniques based on context switching, translation block cache, and vectorization. They argue that the latter was the superior approach due to its correctness with better performance. So, this was chosen as the measured approach.

Then, we began evaluating open source solutions, which were not yet fully exploited by the literature. As mentioned, we searched through GitHub repositories, and found three popular projects Arakawa [Arakawa 2019], Framgia [Framgia 2016] and Gingo [Gingo 2017]. In the vast majority they provided methods focused on the Android API, making their approach easier to incorporate into apps or SDKs and easy for mobile developers to understand, thus the popularity.

The heuristics selection focused in a blend of techniques from the open source

projects and research papers. The code provided by the open source solutions provides the easiness to implement, though it required a lot of adaptation and tests due to changes in Android and lack of documentation. On the other hand the research paper techniques provided a good benchmark, since they were already peer reviewed. For [Vidas and Christin 2014] it was used its software detection, since a comprehensive hardware analysis would require a vast number of real devices that were not available. The vectorization solution from Jang et al. [Jang et al. 2019] was the chosen due to it being the superior solution provided. The rest of the heuristics came from open source projects with occasional minor tweaks explained on earlier sections.

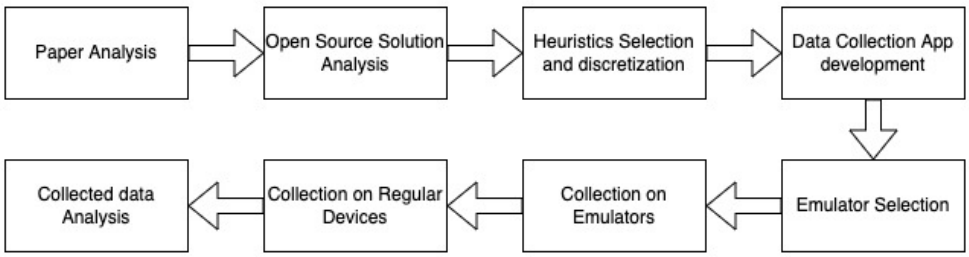

Figure 3. Metodology

An application was developed so that data collection and assertion was closer to a regular application. It also was thought as being easier to collect real device data, and, since this project relied on real device collection, easier for people to use. As soon as the app is initialized it asks for the required Android permissions (it also works without them), and it has a button that zips the collected data and sends it via a `SEND_ACTION Intent`. This enables the data collected to be easily shared. Figure 4 is the app screen, and as it can be seen, there is also a checkbox, as a way to provide everyone to whom the app was shared with a easy way to confirm it did or did not installed it on an emulator.

Figure 5 illustrates how data is collected within the app. The major component is the Collection Block, that is an abstraction containing all the data collection necessary for an assertion and the assertion logic. This is all encapsulated by an measurement in milliseconds of the time required to run this code block. First the UI related data collection blocks are executed, with the only representative of this mode being the two OpenGL implementation as described in Section 2.3.9. Then the app runs on background all the other collection modes. In the end, it provides a zip file with everything.

After the app development, the app was executed 5 times in all emulators described in Section 2.2, to minimize possible outliers and have a more precise analysis. The app was also sent to other people in order to have a benchmark with real devices. The resulting data was analysed for accuracy and performance (the time in milliseconds). A Jupyter notebook was developed and is available on the same repository as the app.

4. Results

We were able to collect data from fifteen real devices, and as mentioned, we ran the data collection five times for each of the evaluated emulators (Bluestacks, Android Studio, Nox, Genymotion). The real devices general characteristics with OS version,

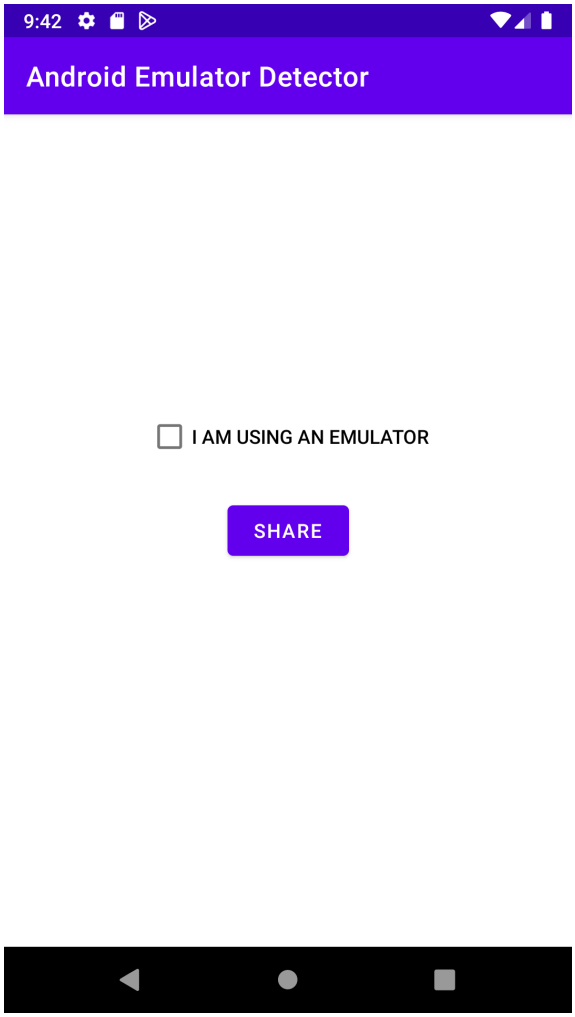

Figure 4. Collection App Screen

manufacturer and model can be seen on Table 8. In following subsections, we detail the result by each method, then we make general considerations of each emulator detection and a technique analysis taking into consideration its intersections.

4.1. Android Build Characteristics

On our first run, we found that all of the Build characteristics techniques proposed by Vidas and Christin [Vidas and Christin 2014] marked every device as an emulator. Upon further investigation, it was concluded that the **Radio** feature had been the reason, due to the fact that it was deprecated. Thus, the current behavior was returning the string marked as suspicious by one of the heuristics [Android 2022e]. Because of that, this variable was removed from the analysis.

Even with the removal, Vidas and Christin [Vidas and Christin 2014] heuristics still had false positives. This happened only on Xiaomi devices, and the reason that triggered the false classification was the **Board** property. The true positives of this heuristic were Android Studio and Genymotion emulators.

The Arakawa and Framgia techniques [Arakawa 2019, Framgia 2016] were able to detect Genymotion emulators, and had no false positives. Gingo [Gingo 2017] was not

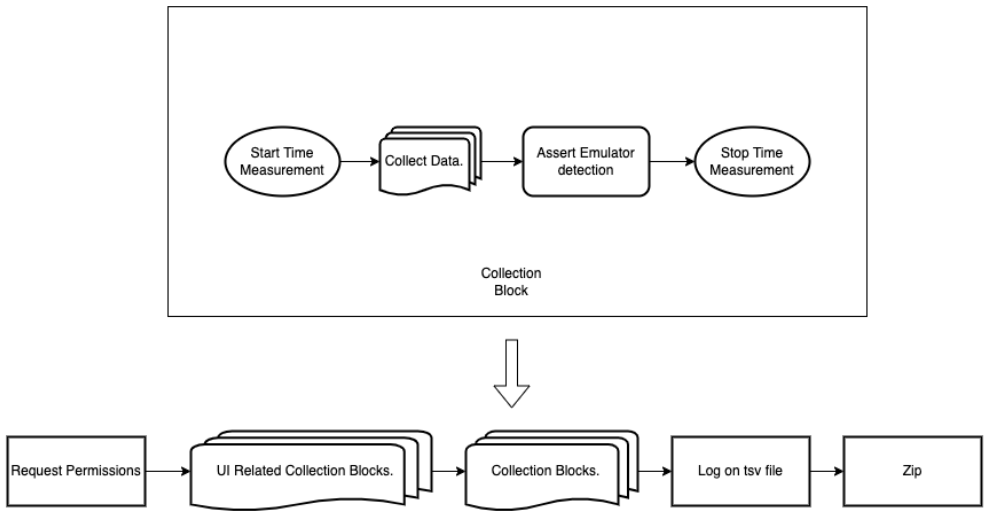

Figure 5. Data collection on App

able to detect any of the tested emulators. Figure 6 contains the overall Build confusion matrix.

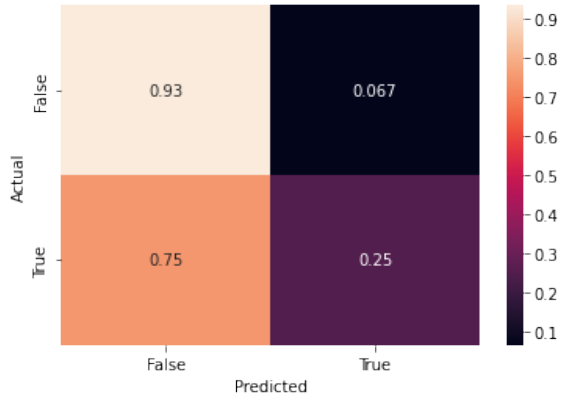

Figure 6. Build Characteristics Confusion Matrix

Overall, the time to perform detection was very low. The maximum time it took to collect the whole data was 33 milliseconds, with a mean value of 2.09 milliseconds and a median value of 1 millisecond.

4.2. File Detection

Gingo [Gingo 2017] was the only one able to detect the Bluestacks emulator. Framgia and Arakawa [Framgia 2016, Arakawa 2019] were able to detect Android Studio emulators. They were not able to detect any other emulator and had no false positives. Their confusion matrix can be seen in Figure 7.

Overall the time needed to perform detection was significantly higher than Build one. The maximum time it took to collect the whole data was 268 milliseconds, with a mean value of 17.74 milliseconds and a median value of 4 milliseconds.

Table 8. Real Devices

Device	OS Version	Manufacturer	Model
1	31	samsung	SM-G980F
2	31	samsung	SM-G780G
3	29	Xiaomi	Redmi Note 8
4	30	motorola	motorola one fusion
5	27	Xiaomi	MI PLAY
6	30	motorola	moto g(8) power
7	27	Xiaomi	MI PLAY
8	30	motorola	moto g(8) power
9	28	motorola	moto e6 play
10	29	samsung	SM-G960U1
11	33	Google	Pixel 6a
12	31	samsung	SM-G985F
13	31	samsung	SM-G780G
14	31	samsung	SM-G998B
15	31	samsung	SM-S906E

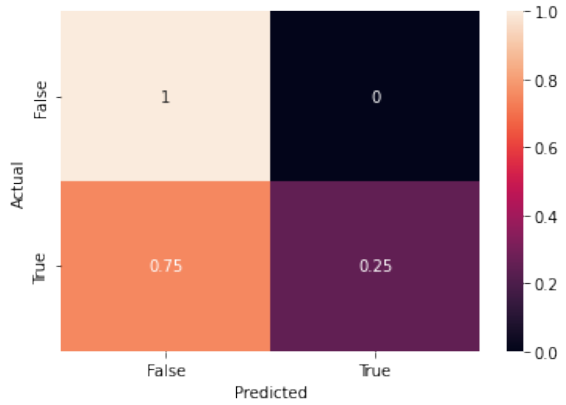

Figure 7. File Detection Confusion Matrix

4.3. Telephony Characteristics

Both Framgia and the simplified proposed heuristic were only able to identify Android Studio and Genymotion emulators, with no false positives. Their confusion matrix can be seen in Figure 8.

For this heuristic, the time to perform detection was significantly higher than for File Detection. This is probably due to its interaction with the system controlled component `TelephonyManager`. The maximum time it took to collect the whole data was 2217 milliseconds, with a mean value of 222.47 milliseconds and a median value of 50.5 milliseconds.

4.4. QEMU drivers

This detection technique was not able to detect any of the tested emulators. Its performance hit was very small with a highest value of 44 milliseconds. The mean value of 9 milliseconds and a median value of 2 millisecond.

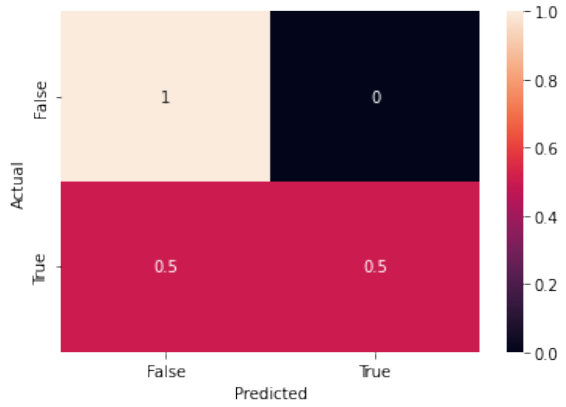

Figure 8. Telephony Characteristics Confusion Matrix

4.5. IP

This detection technique was also not able to detect an emulator. Their impact was also small, but this is due to the fact that it throws an error due to the deprecation of its method, as described in Section 2.3.5.

4.6. Package Names

The only emulator this method was able to detect was Genymotion. It had a relatively high detection time, as it scales accordingly to the quantity of installed packages. Therefore, it is a detection method to be careful with when executing it. But the main device that was responsible to worsening its performance was Android Studio emulator, the overall metrics being: maximum value of 837 milliseconds, mean value of 107.89 milliseconds and median value of 42 milliseconds. The confusion matrix can be seen in Figure 8.

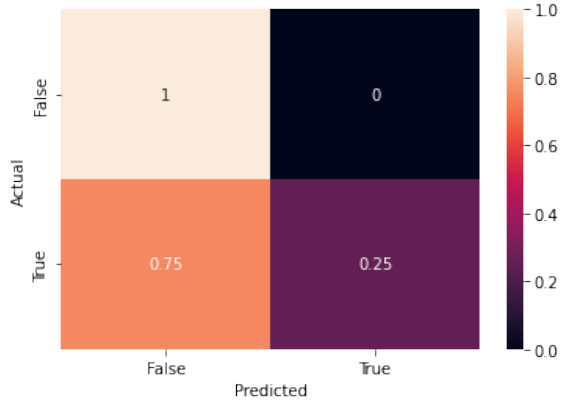

Figure 9. Packages Names Confusion Matrix

4.7. Available Activities

This technique was not able to identify any emulator and had no false positives. The overall performance hit was not high with maximum time of 160 milliseconds, mean value of 26.96 milliseconds and a median value of 16.5 millisecond.

4.8. Services

This approach was also not able to identify any emulator and also had no false positives. Its performance metrics indicates a very high performance hit, due to its maximum value being 2175 milliseconds. But this was an outlier, and their median value was 2 milliseconds.

4.9. OpenGL Render

Due to the issues described in Section 2.3.9, Gingo [Gingo 2017] does not seems to work, not being able to collect any useful data on emulators nor real devices. It was then necessary to instantiate and initialize an OpenGL context in order to correctly obtain any useful data. Those minor alterations were able to detect Android Studio and Genymotion emulators. The confusion matrix considering only the solution that works can be seen in Figure 10.

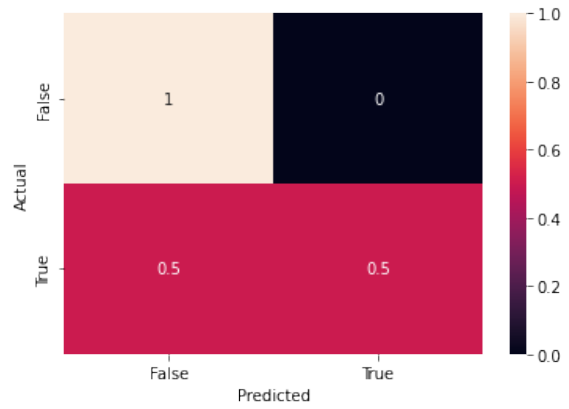

Figure 10. OpenGL Confusion Matrix

This detection technique must interact with the UI thread, since it revolves around loading an OpenGL context. Therefore, its performance is the most critical, since it can directly impact user experience, due to the possibility of causing visible stuttering due to locking the thread for too long. The maximum value found was 345 milliseconds, with mean value of 88 milliseconds and median of 34 milliseconds. The general numbers seem high, but they seem to be pulled up by the Android Studio and Genymotion emulators.

4.10. Vectorization

This method was not able to detect any emulator. As a matter of fact, it did not support any of the emulators ABIs (Application Binary Interfaces) as described in Section 2.3.10. However, it was able to run on all real devices.

Though this heuristic was able to correctly assess all devices running on the *arm64-v8a* ABI, there was an issue that resulted on a false positive on the *armeabi-v7a/NEON* ABI. The support for this architecture should be removed, since this would result in false positives, but the results for *arm64-v8a* devices validation were very promising.

Excluding the problematic result found for the *armeabi-v7a/NEON* ABI, it could be used as a cross check for the build technique proposed by Vidas and

Christin [Vidas and Christin 2014]. This method seems to only be able to serve to validate a real device, instead of acting as an emulator detection technique, since valid devices can have architectures different than the supported ones and the complexity of reliably adding new supported ABIs.

The performance metric in this method had 179 milliseconds as its maximum value. Interestingly, this happened in an unsupported architecture on a emulator. It's mean value was 21.29ms, with median of 5 milliseconds. Its confusion matrix can be seen in Figure 11.

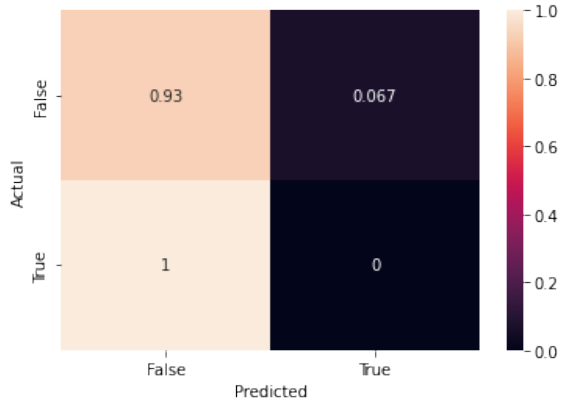

Figure 11. Vectorization Confusion Matrix

4.11. General Considerations

Bluestacks, Android Studio and Genymotion emulators were able to be detected by the proposed techniques, and only Nox was not detected by any technique. Table 9 contains the results consolidation.

Table 9. Results Consolidation

Detection Technique	**Results**
Android Build Characteristics	Detected: Android Studio, Genymotion, False Positives
File Detection	Detected: Android Studio, Bluestacks
Telephony Characteristics	Detected: Android Studio, Genymotion
QEMU drivers	No emulator detected
IP	No emulator detected
Package Names	Detected: Genymotion
Available Activities	No emulator detected
Services	No emulator detected
OpenGL Render	Detected: Android Studio, Genymotion
Vectorization	Detected: False Positives

It's important to note that Bluestacks emulators were only identified by Gingo [Gingo 2017] File Detection heuristic. The heuristics that discovered the Android Studio emulator were: Arakawa and Framgia [Arakawa 2019, Framgia 2016] File Detection, Android Build Characteristic by Vidas and Christin [Vidas and Christin 2014] , Telephony Characteristics by Framgia [Framgia 2016] and OpenGL Render by

Gingo [Gingo 2017] . Finally, Genymotion was detected by the most number of approaches, with them being: Android Build Characteristics by Vidas and Christin [Vidas and Christin 2014] and the same heuristic by Framgia [Framgia 2016] , Telephony Characteristic by Framgia [Framgia 2016] , Package Names by Arakawa [Arakawa 2019] and OpenGL Render by Gingo [Gingo 2017] .

We can see that open source solutions had a lot of good methods that were able to detect emulators, but had also a lot of them that failed in doing so, like Available Activities and Services, and some that aren't even supported anymore, like IP detection. The Android Build Characteristics solution proposed by Vidas and Christin [Vidas and Christin 2014] contained a high number of false positives, even with the fix of the `Radio` value, and is not advised, unless some tweaks are made. Jang et al. [Jang et al. 2019] vectorization approach was very promising on supported *arm64-v8a* architecture, detecting all real devices on it, but either wasn't supported or misidentified real devices on other ones.

The most interesting from all the heuristics were File Detection, being able to detect Bluestacks and Android Studio, OpenGL and Telephony Characteristics, that were capable of detecting Android Studio and Genymotion. There are a few caveats though. First, File detection heuristic actually consists of merging Arakawa's and Framgia [Arakawa 2019, Framgia 2016] heuristics with the one proposed by Gingo [Gingo 2017], with the latter being the only one able to detect the Bluestacks emulator. Second, though OpenGL Render detection proposed by Gingo [Gingo 2017] performance wasn't so bad, it is of very sensitive nature, since it runs on Android's UI thread and delays may impact user experience, due to stuttering and frame drops, and further investigation may be necessary. Third, the Telephony Characteristic requires some sensitive permission to be able to detect, this permission may not be natural by the app to ask for, and can cause strangeness for the user. Because of these latter two problems, a simpler solution that only detects Genymotion may be considered, like Package Names by Arakawa [Arakawa 2019], along with the File detection pair.

5. Conclusions

The majority of the tested emulators had a technique that was able to detect them, with Nox being undetected by any method. But our evaluation shows that, so far, there was no single approach that is able to identify the rest of them. Because of that, a set of techniques must be used. File detection proposed by Arakawa, Framgia and Gingo [Arakawa 2019, Framgia 2016, Gingo 2017], Telephony Characteristics by Framgia [Framgia 2016] and OpenGL proposed by Gingo [Gingo 2017] and were the best in detecting emulators, but the best subset achieving the highest number of detections with the least performance and user experience impact are the combination of the File Detection techniques and the Package Names proposed by Arakawa [Arakawa 2019].

The vectorization technique proposed by Jang et al. [Jang et al. 2019] is currently not developed enough, but can be used to validate real devices that use the *arm64-v8a* ABI. It does also have the potential to be expanded upon adding more ABIs support, but need further tweaking and tests.

5.1. Similar works

In his PhD thesis, Rashid [Rashid 2018] has made an analysis on emulator detection and bypassing. His analysis does not focus on performance nor it specifies the detection efficiency in each method, focusing instead on an external dynamic analysis.

Vidas and Christin [Vidas and Christin 2014] made an analysis similar to the one proposed here. However, open source proposed techniques by Framgia [Framgia 2016], Gingo [Gingo 2017] and Arakawa [Arakawa 2019] added more checks to the Android Build Characteristics and Telephony Characteristics. Other than that, Android has changed a lot since the time of publishing of the article, specifically regarding it's permissions that have become stricter due to the stronger public emphasis on privacy and control over device data.

5.2. Future works

This comparison can be extended for other emulators on other operating systems, to validate the results found here. The vectorization technique does have potential to be a good detection technique, but the lack of support for important architectures tampers its potential. Adding support to `x86` and `x86_64` would provide such possibility, so it could be used for emulation detection instead of real device validation. Since no heuristic tested here was able to detect the Nox emulator, further research is necessary to its architecture and inner workings.

A deeper investigation on the performance hit of OpenGL detection may also be useful, since this detection showed a good track record on Android Studio and Genymotion emulators. The values returned by the render on the Bluestacks and Nox emulators were also not found on real devices, and this can also be investigated.

Another possible future work is using Frida to bypass emulator detection and bypass detection. Frida is a dynamic instrumentation toolkit that supports Windows, macOS, GNU/Linux, iOS, Android, and QNX. Its widely available for these platforms and allows injecting snippets of JavaScript or your own library into native apps on supported platforms. Frida also provides simple tools built on top of its API. These can be used as-is, tweaked to your needs, or serve as examples for how to use the API. It injects Googles V8 engine into the target processes, where your JavaScript code gets executed with full access to memory, hooking functions and even calling native functions inside the process [Rashid 2018]. This is a powerful tool that has the ability to bypass security checks, so its detection is also an important research subject.

References

Android (2022a). Activity manager. `https://developer.android.com/reference/android/app/ActivityManager`. Accessed: 2022-09-18.

Android (2022b). App manifest overview. `https://developer.android.com/guide/topics/manifest/manifest-intro`. Accessed: 2022-10-03.

Android (2022c). Application fundamentals. `https://developer.android.com/guide/components/fundamentals`. Accessed: 2022-09-17.

Android (2022d). Application sandbox. `https://source.android.com/docs/security/app-sandbox`. Accessed: 2022-10-02.

Android (2022e). Build. https://developer.android.com/reference/android/os/Build. Accessed: 2022-09-25.

Android (2022f). Eglsurfaces and opengl es. https://source.android.com/docs/core/graphics/arch-egl-opengl. Accessed: 2022-09-18.

Android (2022g). Implementing system properties as apis. https://source.android.com/docs/core/architecture/sysprops-apis. Accessed: 2022-10-01.

Android (2022h). Intents and intent filters. https://developer.android.com/guide/components/intents-filters. Accessed: 2022-09-17.

Android (2022i). Introduction to activities. https://developer.android.com/guide/components/activities/intro-activities. Accessed: 2022-09-17.

Android (2022j). Manifest permissions. https://developer.android.com/reference/android/Manifest.permission. Accessed: 2022-09-17.

Android (2022k). Opengl es. https://developer.android.com/develop/ui/views/graphics/opengl/about-opengl. Accessed: 2022-09-18.

Android (2022l). Package visibility filtering on android. https://developer.android.com/training/package-visibility. Accessed: 2022-09-17.

Android (2022m). Packagemanager. https://developer.android.com/reference/android/content/pm/PackageManager. Accessed: 2022-10-03.

Android (2022n). Permissions on android. https://developer.android.com/guide/topics/permissions/overview. Accessed: 2022-09-29.

Android (2022o). Request app permissions. https://developer.android.com/training/permissions/requesting. Accessed: 2022-09-17.

Android (2022p). Services overview. https://developer.android.com/guide/components/services. Accessed: 2022-09-18.

Android (2022q). Telephony manager. https://developer.android.com/reference/android/telephony/TelephonyManager. Accessed: 2022-10-01.

AOSP (2022). Android open source project. https://source.android.com. Accessed: 2022-09-11.

AOSP-Architecture (2022). Android architecture. https://source.android.com/docs/core/architecture. Accessed: 2022-09-11.

Arakawa, Y. (2019). Emulatordetector. https://github.com/mofneko/EmulatorDetector. Accessed: 2022-09-07.

ArchLinux (2022). netcfg. https://archlinux.org/netcfg/. Accessed: 2022-09-17.

Cardoso, L. (2022). Android emulator detector. https://github.com/Lucas-CardosoO/android-emulator-detector. Accessed: 2022-10-02.

Corbet, J., Rubini, A., and Kroah-Hartman, G. (2005). *Linux Device Drivers*. O'Reilly Media, Inc, 3rd edition.

Dimjašević, M., Atzeni, S., Ugrina, I., and Rakamaric, Z. (2016). Evaluation of android malware detection based on system calls. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, IWSPA '16, page 1–8, New York, NY, USA. Association for Computing Machinery.

Framgia (2016). Android emulator detector. `https://github.com/framgia/android-emulator-detector`. Accessed: 2022-09-07.

Genymotion (2022). Genymotion. `https://docs.genymotion.com/desktop/Get_started/012_Macos_install/#virtualbox`. Accessed: 2022-09-12.

Gingo (2017). Android emulatordetector. `https://github.com/gingo/android-emulator-detector`. Accessed: 2022-09-07.

Google (2016). netcfg command not found. `https://issuetracker.google.com/issues/37081688`. Accessed: 2022-09-17.

Google (2022). Target api level requirements for google play apps. `https://support.google.com/googleplay/android-developer/answer/11926878`. Accessed: 2022-09-17.

Guerra-Manzanares, A., Bahsi, H., and Nõmm, S. (2019). Differences in android behavior between real device and emulator: A malware detection perspective. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 399–404.

Iacurci, G. (2022). Consumers lost $5.8 billion to fraud last year — up 70% over 2020. *CNBC*. Accessed: 2022-09-10.

Jang, D., Jeong, Y., Lee, S., Park, M., Kwak, K., Kim, D., and Kang, B. B. (2019). Rethinking anti-emulation techniques for large-scale software deployment. *Computers Security*, 83:182–200.

Lin, J., Liu, C., and Fang, B. (2019). Out-of-domain characteristic based hierarchical emulator detection for mobile. In *Proceedings of the 2nd International Conference on Information Technologies and Electrical Engineering*, ICITEE-2019, New York, NY, USA. Association for Computing Machinery.

Martz, P. (2006). *OpenGL Distilled*. OpenGL. Pearson Education.

QEMU (2020). Qemu-wiki. `https://wiki.qemu.org/Main_Page`. Accessed: 2022-09-12.

Rashid, W. (2018). *Automatic Android Malware Analysis*. PhD thesis.

RedHat (2022). E.2.3. /proc/cpuinfo. `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/deployment_guide/s2-proc-cpuinfo`. Accessed: 2022-09-17.

Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., and Hoffmann, J. (2013). Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, page 1808–1815, New York, NY, USA. Association for Computing Machinery.

statcounter (2022). Mobile tablet android version market share worldwide. https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide. Accessed: 2022-09-17.

Statista (2022). Mobile app consumer spending worldwide from 2021 to 2026, by store. https://www.statista.com/statistics/747489/annual-consumer-spend-mobile-app-by-store/#statisticContainer. Accessed: 2022-09-10.

Turner, D. (2014). Goldfish virtual hardware. https://android.googlesource.com/platform/external/qemu/+/emu-master-dev/android/docs/GOLDFISH-VIRTUAL-HARDWARE.TXT. Accessed: 2022-09-13.

Vidas, T. and Christin, N. (2014). Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, page 447–458, New York, NY, USA. Association for Computing Machinery.

Xu, F., Shen, S., Diao, W., Li, Z., Chen, Y., Li, R., and Zhang, K. (2021). Android on pc: On the security of end-user android emulators. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1566–1580, New York, NY, USA. Association for Computing Machinery.