ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica - Scienza e Ingegneria

Corso di Laurea in Ingegneria e Scienze Informatiche

A study of dynamic analysis use cases in Android applications for third party libraries detection

Relatore
Gabriele D'Angelo

Presentata da:
Giacomo Boschi

III Sessione di Laurea Novembre
Anno Accademico 2024-2025

Abstract

The Android operative system is the most used for mobile devices, which cover a large part of daily life. Because of this, many branches of security research on Android applications have emerged, with one of them being the detection of Third party libraries (TPLs). Studying the properties of the applications code, that is performing static analysis techniques, is the go-to approach when it comes to this branch of research, resulting in dynamic analysis driven approaches being relatively understudied. The following thesis explains the state of the art for TPL detection and discusses possible types of dyanmic analysis approaches, which study the run-time behavior of the target applications, making confrontations with static analysis and showing how even very simple solutions in terms of costs and implementation provide complementary information for state of the art methodologies.

I would like to thank my supervisor Gabriele D'Angelo for all the help and support provided in the creation of this thesis.

Contents

List of Figures

1. Introduction

It is estimated that there are more than 5.7 billion people using a smartphone [6], a device that is increasingly present in daily life, working as an access point to sensible information through specific applications. For example, studies estimate that more than 60% of the global population uses home banking applications [7], and more than 5 billion people use social media [8].

A smartphone's resources are managed by an operating system (OS), with Android being the most used for mobile devices and having a market share of more than 70% for said devices [9]. Android is currently open source and offers the possibility to distribute applications through the official App Store (Google Play Store), but also allows a developer to use the APK (Android App Package) file that can be shared for installation, giving the possibility of distribution through a personal medium (like a website, for example), which could be more desirable for some developers.

It is worth pointing out that many limitations will probably be placed on this system, as at the time of writing this paper, Google recently announced a developer verification process [10] which would require developers to identify themselves (using information like legal name and government ID) and register any of their applications, as certified Android devices will only install registered applications. Regardless, Android currently remains the most used OS for mobile devices and still allows for installation of apps from external sources.

Many applications have common needs and requirements (for example, being able to communicate with a remote server, or communicating with appropriate cryptographic protocols, or managing specific file types), so instead of implementing every functionality from zero, it is very common practice for the development of Android applications to use **libraries**, sets of software functionalities that can be easily imported into a project through the Gradle build system. By calling the appropriate functions, it is possible to execute the desired operations without knowing the implementation behind the functions that are being called. While some libraries are a direct creation of the Android development team, and are constantly updated and maintained by the team itself, some are the work of individuals or groups outside the Android team, and are called Third Party Libraries (TPL). An example of TPL for Android is OkHttp [11], a library used for making HTTP and HTTPS requests to external servers.

The usage of TPLs is very common in Android, so much that it is estimated that more than 60% of code in Android applications is composed by TPLs [12]. While using TPLs offers many advantages and allows for faster app development, a TPL can also be a vector for vulnerabilities, which can compromise the device and the user. Being capable of identifying which TPLs have been used and correctly determining their version becomes an important part of security-related tasks such as penetration testing of applications and malware detection. It is a very good practice to keep libraries updated and to not use obsolete versions. In recent years, a lot of research has been done to tackle this problem, resulting in the development of software and methods for TPL detection and versioning, with a success rate that keeps improving, but is still far from perfect. The objective of this thesis is to show a working environment with various methods for improving already existing literature and addressing the problem, by also presenting possible solutions based on unusual techniques.

2. Understainding Android applications

Android applications are built on 4 basic components [13]:

- **Activities**: these are the entry points for user interaction, as they represent screens with graphical interfaces. Each activity is independent of the others, although their combination usually defines a user experience with the given application.
- **Services**: these are used for running application processes in background (like playing music for example). Services can be run until they are done executing their work (Started Services) or could be requested by an external process or application (Bounded Services).
- **Broadcast receivers**: these entries allow the system to communicate events to applications without normal user interaction. They can be used for events handling without having the application being constantly open. This allows, for example, to send notifications after a specific event has occurred.
- **Content providers**: they manage shared sets of application data and regulate interactions with them. With content providers it is possible to manage data on the file system that more than one application operates on.

Android applications are usually written in Java or Kotlin, but it's also possible to write code using native libraries in C++, with Android Studio being the official integrated development environment (IDE) [14]. Both Java and Kotlin have their own classes to manage the explained components, so when analyzing the code of any application, it will be very common to encounter files that define an activity or a service through the proper class. For simplicity moving forward, when referring to an Android project or an Android application, it is assumed that Android Studio is being used for development.

2.1 APK files

APK is the file format for installing an Android application on a device, and it gives a very large amount of information regarding the application being installed. An

APK is a ZIP archive [15] that contains different files and directories, including the compiled source code of the application. More specifically, it contains the following files and folders [16]:

- **The manifest file**: this is one of the most important files in any Android application: it's a xml file used for defining the basic components of the program (Activities, Services, Broadcast Receivers, Content Providers) and defining basic information like app version, required permissions, API and device compatibility and much more.
- **Compiled resources**: this is a folder (usually called **resources.arsc**) containing information like strings, colors and styles used by the app.
- **Native libraries (lib)**: this is a folder that contains the machine code of the native libraries for the different architectures.
- **Assets**: folder that contains files used by the application. The files present in this folder are in a binary format, like audio files and fonts.
- **Resources**: this folder contains files required for defining the user interface, like xml files containing the interface layout and images.
- **classes.dex**: this folder contains the compiled bytecode in the DEX (Dalvik Executable) format, which has to be translated to machine code by a proper Virtual Machine.
- **META-INF**: this folder contains metadata about the APK file, including necessary signatures and certificates information.

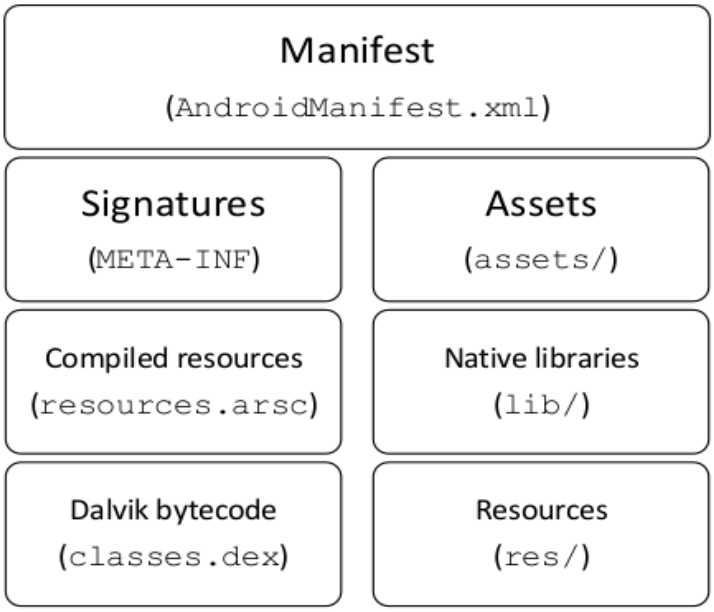


Figure 2.1: Structure of an APK file [1]

Since APKs act as archives, they can be decompressed and the files of the application can be read and analyzed. This means that having the APK of an Android application allows for the study of the application through static analysis, which is the practice of understanding how a program acts not by how it behaves during execution, but by studying its ”non-changing” parts, mainly by reconstructing the source code of the application by disassembly of the bytecode.

2.1.1 Obtaining the APK of an application

From what has been explained so far, it is simple to understand that possessing the APK for the application that is being analyzed gives a lot of important information: anybody desiring to perform any sort of security assessment of an Android app will probably want to obtain the associated APK. The immediate problem that emerges is that not only does installing applications through the official Google Play Store not show the APK file, but there is not an official method for retrieving the APK of an application. A common way of obtaining APKs is to rely on a third party with their own medium for distributing APKs, especially in scenarios where conducting studies requires a large test set of applications, with Androzoo being a well-known example in academic environments [17]. Another possibility is the extraction of APK files of applications already installed on a device.

2.2 Android Applications at runtime

Now that it has been explained what constitutes an Android application, it is worth understanding how Android applications are run and managed.

2.2.1 Executing an application

For executing an application's instructions, Android's approach is very similar to that of a Java Virtual Machine (JVM): the idea is to make applications cross-platform by converting files to bytecode that is equivalent across all device architectures (in the case of Java, this results in .class files), and then having a virtual machine that translates the bytecode to executable machine code for the specific architecture [18]. Android adapted this method to the limited resources of its devices: the first versions used a **Dalvik Virtual Machine** (DVM), which translated Dalvik Bytecode, contained in **classes.dex** file, to machine code, and used JIT compilation [19] (Just In Time compilation), which translated bytecode to instructions during the execution of the program. This was useful as JIT compilation did not require saving machine code on devices that had very limited storage capacity. From version 5.0 of Android (Lollipop), this was replaced with **Android Runtime** (ART) [20] which uses AOT compilation (Ahead Of Time compilation): while a DVM must translate the .dex file every time the application is run, ART will translate the DEX bytecode once during app installation, saving machine code files that can be directly executed without having to translate every time the app is being run.

2.2.2 Managing applications

The Android OS is built upon the Linux OS, with each application being executed in its own isolated process, which has its own ART instance executing the application code [21]. Each process is a fork of the **Zygote** process [22], which acts as the root of all Android processes, and it's started when the OS itself is initialized. When Zygote is forked for starting a new application, a new instance of the Virtual Machine is requested (ART or DVM based on version) through an interface called Berkeley Socket (BSD) [23]. Processes will be placed in a hierarchy list based on importance, so that Android can terminate processes with the lowest importance to reclaim memory if necessary.

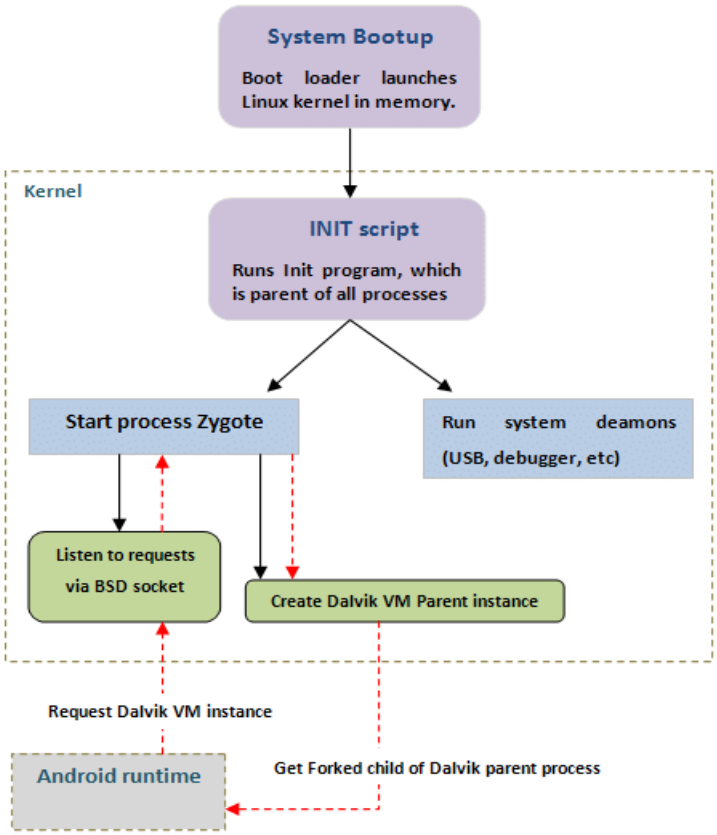


Figure 2.2: How Zygote works [2]

2.3 TPLs and vulnerabilities

The build system for Android development is **Gradle** [24], which allows for a relatively easy importing of wanted third party libraries, by inserting dependencies in an specific file, (usually called *build.gradle*, but naming may vary). For managing versions, a *libs.versions.toml* file is used which keeps track of versions and assigns aliases to libraries. Any vulnerability found in software that is caused by a TPL's code is particularly dangerous as any application using the TPL is subject to the consequences of the vulnerability, so almost in any scenario, finding a vulnerability in a TPL ends in the developer of the library patching the exploit and releasing a new version of the TPL. This means that detection of outdated TPL versions is fundamental for finding vulnerabilities.

2.3.1 Classifying vulnerabilities

To record vulnerabilities found in software and keep a historical record of them, multiple large databases with their own standards for storing and assessing the dan-

Listing 2.1: Example of Gradle build file with toml versioning

```
//libs.versions file
[versions]
okhttp = "5.1.0"
picasso = "2.8"

[libraries]
okhttp = {
    module = "com.squareup.okhttp3:okhttp",
    version.ref = "okhttp"
}

picasso = {
    module = "com.squareup.picasso:picasso",
    version.ref = "picasso"
}

//gradle.build file
implementation(libs.okhttp)
implementation(libs.picasso)
```

gers of registered vulnerabilities. This thesis will use the **Common Vulnerabilities and Exposure (CVE)** database [25], a large vulnerability database that registers and classifies reported vulnerabilities. When a new vulnerability is discovered, it is reported to a CVE partner, which assigns a CVE ID defined by the CNA (CVE Number Assignment) rules. After that, the partner submits the details of the vulnerability and publishes the CVE record. A score from 0 to 10 is assigned to the record through the Common Vulnerability Scoring System (CVSS) which defines the severity of the vulnerability based on different metrics (base, temporal and environmental) [26]. In simpler terms, the objective is to identify versions of TPLs used in an application and to check for the existence of associated CVE reports.

3. Static analysis of applications for TPL detection

As mentioned briefly in Chapter 2, static analysis is a form of reverse engineering based on observing the elements that compose an application, mainly the source code and the assets used by it, without actually executing the program. In the case of an Android application, the APK doesn't contain the source code of the program (which might be written in Kotlin, Java etc.), but the DEX file with the bytecode, which isn't very readable for humans. **Decompilation** can be considered the opposite process of compilation: a compiled file is passed to a program (decompiler), which produces one or more files in a language that is more readable for humans and easier to analyze. In the case of Android, DEX files can be decompiled to obtain files in the **smali** assembly language, which is designed to be a version of the DEX bytecode that is more readable by a human agent [27].

3.1 Obfuscation techniques

Obfuscation techniques are one of the biggest roadblocks for TPL detection. Obfuscation is an intermediate step in the creation of compiled files that attempts to "pollute" the code by removing or adding information, making the code behavior and properties harder to understand. Obfuscation is performed by specialized programs called obfuscators. In the specific case of Android, Proguard and R8 are well-known examples and R8 is also integrated in Android Studio [28]. Obfuscation techniques include [29][30]:

- **Identifier renaming**: most identifiers, such as variables and class names, are replaced by randomly generated names that have no meaning.
- **String encryption**: String literals are encrypted and are not directly readable from the bytecode. They are decrypted during runtime.
- **Reflection**: this is a feature of Java that developers can use for flexible interactions with the program by, for example, dynamically creating objects and invoking methods. In the context of obfuscation, this feature is used to hide implementations and call functions implicitly.

- **Control flow manipulation**: this technique attempts to complicate the control flow of a program by modifying its conditions. Different techniques can be applied; examples are adding junk conditions that are always true or false, merging conditional blocks, or splitting them.
- **Dead code removal and addition**: non functional code (junk code) is added to the flow of the application, complicating the reversing process, or it performs optimizations that make the code less understandable. There are multiple ways of performing this; conditional statements and loops might be modified, and unnecessary calls to functions might be made.

While these are the most prevalent techniques, other possibilities exist: manifest files might be modified, the packages might have their structure changed, and the code might be disassembled and reassembled to change the order of elements in the APK etc. These practices are part of a process called **repackaging**, which is common enough in obfuscation, especially for hiding malicious payloads [30]. In general, the operations performed depend on the obfuscators being used, but they will all usually cause more imprecise decompilation results, making it harder for a third party to understand the code.

3.2 Methods for TPL detection

There are 3 main approaches for detecting TPLs in applications through the use of static analysis: using a whitelist, using machine-learning algorithms, and using signatures created through a process of feature extraction [31][32]. The whitelist method is the simplest to implement: it tries to identify TPLs by checking package names in the application and finding matches with a list of packages related to TPLs (the whitelist). While this method is the easiest to implement, it becomes ineffective when obfuscation techniques are introduced. Considering that it is estimated that more than 25% of Android applications use obfuscation (and when restricting studies to groups of more popular applications, this percentage increases to 50%) [33], this method is not very reliable. Even if obfuscation wasn't a problem, whitelisting requires a constantly updated list of TPLs, which isn't very practical given the scale of the marketplace at hand. Methods based on machine learning algorithms usually work on classification problems or clustering problems. Classification techniques try to identify patterns for identifying specific types of libraries, and it is mostly used for identifying ad related libraries, while clustering techniques try to group together libraries that are similar to one another. Using features is by far the most popular method for TPL detection tools, and it's going to be the focus for the rest of this chapter.

3.2.1 Methods based on feature extraction

The basic concept of feature extraction is that TPLs have unique, identifiable characteristics (features) that can be used to recognize one TPL from another. While tools that are based on these methods work in different ways, there are usually 4 common steps identifiable in TPL detection processes of this type [34][35]:

- **Preprocessing**: a large enough set of applications is gathered, and their APKs are decompiled using proper tools. Apktool and Androguard are famous examples for decompilers [36][37].
- **Library Istance Construction**: the goal of this step is to separate the main module of the application, containing the code written by the developers, from the TPLs code, producing as a result a set of TPL candidates. This step is not always taken since, depending on how features are extracted, tools might be able to differentiate TPLs from the host application without needing this intermediate step.
- **Feature Extraction**: features that can uniquely identify TPLs are extracted, the combination of these features can be considered a fingerprint that can be used to match libraries.
- **Library Identification**: this final step uses all the data obtained in the previous steps to actually identify TPLs in applications with a certain grade of precision.

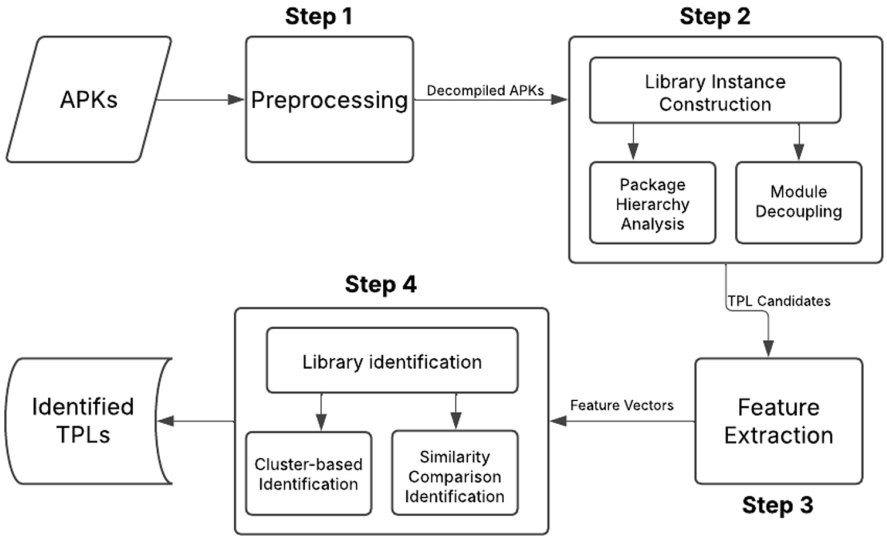


Figure 3.1: Basic work flow for TPL detection [3]

The identification of libraries usually happens with either clustering based methods or similarity based methods [38]. In the first one, after separating the host module with the app code from the other modules, modules with similar features are grouped (clustered) together. All modules contained in the same cluster are considered a TPL. In the second one, all app modules are treated as TPLs candidates and the features extracted from the app modules are compared to the features extracted from TPLs collected beforehand (this means that not only APKs of applications must go through preprocessing but also the target TPLs).

There aren't strict rules for what constitutes a feature, but they are usually choosen so that they are resilient to obfuscation and provide a good level of uniqueness for identifying libraries. Examples of features are [34]:

- **Class dependencies**: the relationship between classes, like inheritance, fields of the classes and methods of the classes.
- **API calls**: the set of Android APIs called by the modules.
- **Methods signature**: the signature of a method is all the charateristics that uniquely describe that method (name of the method, return type of the method, types and names of all the fields of the method). This feature is based on collecting signatures.
- **Control flow graph (CFG) blocks**: this feature is defined by basic blocks of instructions that can be identified inside the code of the modules.

The representation of the fingerprint is usually done with one or multiple hash functions. Sometimes entire specialized data structures are used, for example Libscout uses a Merkle tree for representation [39].

3.3 The state of the art

A lot of study has been conducted in recent years about TPL detection, which brought to life numerous tools with their own methodologies for each step of the process. Table 3.1 shows and compares a sample of tools developed in recent years [34][40][41].

Table 3.1: TPL Detection Tools Comparison

Tool	**Year**	**Library Identification**	**Features Extracted**
LibScout	2016	Similarity comparison	Method signatures
LibRadar	2016	Clustering	API calls
LibD	2017	Clustering	CFG blocks
LibPecker	2018	Similarity comparison	Class dependencies
ORLIS	2018	Similarity comparison	Method signatures
LibID	2019	Similarity comparison	Class dependencies
LIBLOOM	2023	Similarity comparison	Packages and classes
Libscan	2023	Similarity comparison	Classes signatures

While tools are getting more precise in modern times, obfuscation techniques still pose a huge limitation to the precision of TPLs detection. Also using static analysis alone misses a lot of useful information that can be obtained during the runtime of an application. The following chapters will explain how dynamic analysis aspects can be introduced in a study for TPL detection, showing part of the tools and literature already existing on the topic.

4. Using dynamic analysis for TPL detection

Dynamic analysis is the study of a program while it is being executed. This type of analysis finds uses in studies related to malware detection and privacy leakage (which is the unplanned exposition of private information) inside applications, but its use in the detection of third party libraries has been extremely limited. An intersection of the studies is when researchers try to detect malicious TPLs which contain dangerous payloads, but the problem with this type of study is that it usually has binary classification as its end goal (that is, identifying libraries as either benign or malicious, without any sort of versioning or sometimes even clear identification of the actual TPL).

4.1 Dynamic analysis techniques

Many techniques exist in the field of dynamic analysis for Android, so a valuable first step is to identify which are the most usable for the scope of the study. A recent literature review of dynamic analysis techniques has categorized methods into 7 groups [42]:

- **Fuzzing**: this technique tests the target program with large amounts of input to look for unexpected behaviors.
- **Network Traffic analysis**: As the name suggests, this technique attempts to intercept and study the content of network traffic coming from and to the target application.
- **Taint analysis**: the execution path of a program (flow) is studied and mapped.
- **Tracing and hooking**: calls to functions,Android APIs and system calls are tracked.
- **Log-based analysis**: Logs released by applications through logging systems are collected for information. Usually, Logcat [43] is the standard log system for Android developement.

- **Memory decomposition**: The memory content of applications is analyzed to gather information.
- **Visualization assistance**: tools for visualizing data are used for a better understanding of the gathered information.

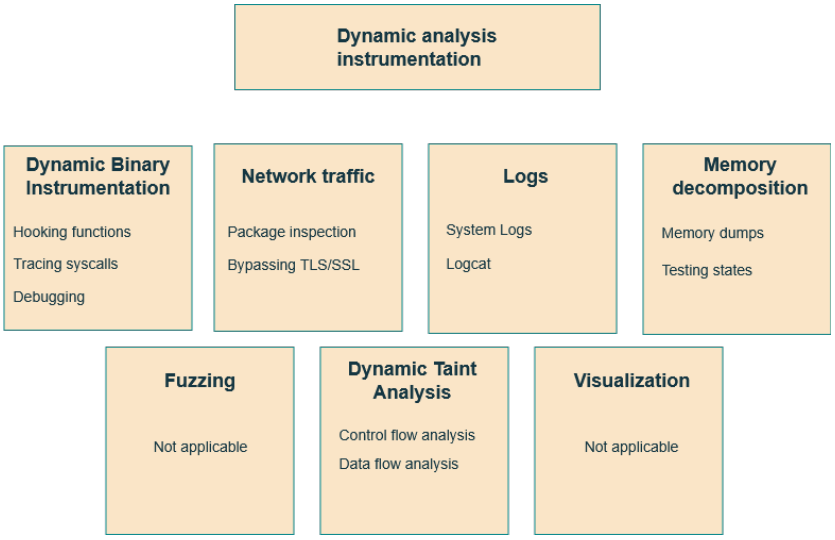


Figure 4.1: Cathegorization of dynamic analysis techniques

4.1.1 Tracing and hooking

Tracing consists in the tracking of system calls (syscalls for short) and signals made by a process, through the use of a specialized program called tracer. Debuggers and performances analyzers use tracing. A very simple way to trace a process is using the Linux command **strace** [44], which tracks syscalls made by a process. Function calls can also be intercepted during tracing, for example Linux can do so by using the command **ltrace** [45]. Tracing can be supplemented by frameworks called **Dynamic Binary Instrumentation (DBI)** frameworks [42], which allow for optimization and avoid the recompilation of code. The most important aspect of a DBI framework is the possibility of performing **hooking**, which is the manipulation of the control flow of an application by the injection of instructions. There are different ways in which hooking is implemented in Android, some examples are:

- Replacing the Zygote process with a custom process or modifying the existing Zygote: the Xposed framework for example will modify the $/system/bin/app\_process$ file creating an extended process [46][47].
- Tampering with the virtual memory of ART runtime, which contatins the representation of Java classes and methods [48][49].

- Injecting jump instructions to the beginning of functions during program execution (inline hooking) [50].
- Modifying the Global Offset Table (GOT), which in a Linux executable contains the offsets used to determine where functions start [51].

4.1.2 Observing network traffic

The content of network communications coming from an Application can contain plenty of useful information to identify a library: for example, most popular TPLs for HTTPS communication have their own user agent. Instrumentation for traffic interception is well known and easily accessible: tools like Wireshark [52] have been in circulation for years, and many DBI frameworks possess functionalities for performing this type of analysis themselves. To be considered is also the use of hooking for intercepting the functions sending packets. **Tracedroid** is an actual example of this [4]: it's a tool that analyzes HTTP and HTTPS requests to identify cases of privacy leakage caused by third party libraries.

4.2 Using dynamic analysis for TPL detection

The objective of this study is to observe whether dynamic analysis techniques can identify TPLs used inside an application, but to do that a strategy must first be defined. I decided to observe the feasability of a similarity based methodology by extracting features, like in static analysis. The concept of feature extraction isn't new in dynamic analysis: in fact, many malware detection studies and tools define features to uniquely identify malware inside applications. This means information collected during dynamic analysis using the techniques explained in Section 4.1 could be used to attempt to create features that can uniquely identify third party libraries. A small initial set of candidates is shown in the table below, but possibilities can easily be expanded and more research can be performed.

Table 4.1: Examples of candidates for creating features

Feature	**Technique**	**Observations**
Syscalls	Tracing	Results can be confronted with static analysis tools.
Loaded classes	Hooking	Features used by similarity comparison tools could be extracted. Can be confronted with static analysis results.
Network data	Hooking, Network traffic	Information is limited to protocols used. Can only be used for network related TPLs.
API calls	Hooking	Features used by similarity comparison tools could be extracted. Can be confronted with static analysis results.
Interactions with files	Hooking, Tracing	TPLs might have their personal temporary file or might work on a subset of shared files, making it a valuable candidate.

4.2.1 Combinining techniques: hybrid analysis

Hybrid analysis is the combination of static analysis and dynamic analysis techniques. As it is observable from Table 4.1, a few candidates (for example, API calls) are also recognizable from the features explained in Chapter 3. This means that if dynamic analysis is implemented successfully, both types of analysis can be used together to improve current results. A theoretical approach would be to use dynamic analysis data as filters: static analysis instrumentation is used for the identification of TPL, while dynamic analysis finds false positives.

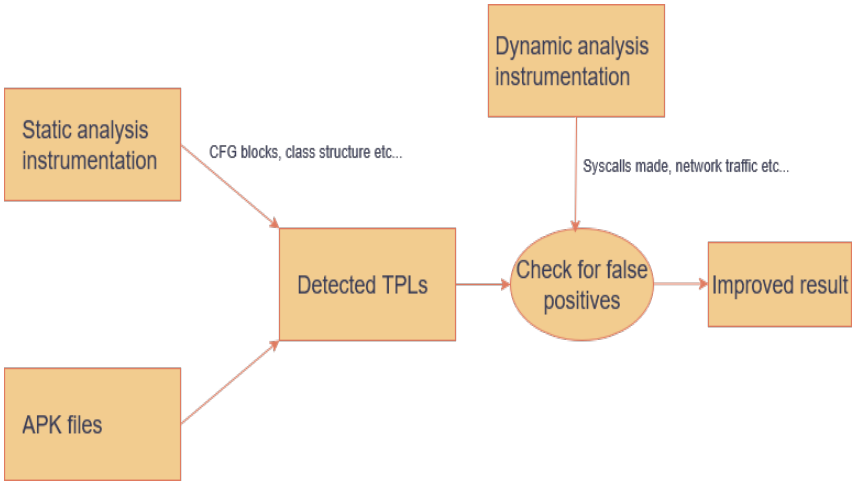


Figure 4.2: Theoretical architecture for additional detection of false positives

4.3 Rooting devices

When Android devices are manufactured, some limitations are put on the OS for security reasons, with the main one being that the user has no capacity to interact with the system using superuser privileges. In fact, that is the point of sandboxing applications: it is to avoid unwanted privileged access to the device data and resources. However, to correctly perform their operations, most DBI frameworks will require being executed in a superuser (su) process (essentially, being run with the **sudo** command [53]). The process of bypassing Android limitations on a device and giving the user root access to the device's OS is referred to as **rooting**. While various techniques can be adopted for rooting, the practice usually consists in installing a modified su binary on the device [54]. While there are different possibilities when it comes to non-rooted devices, this study will focus on a environment with rooted devices.

4.4 Known limitations

Before proceeding with the exploration of used instrumentation for the study, a few known limitations must be explained:

- Because this approach requires directly executing applications, automatization of this process on large scale could be troublesome, since there is no database for retrieving information about features like in static analysis techniques.
- Precise versioning might be difficult in many scenarios, since small version changes usually have too small modifications to be distinguished.

- Having to execute applications means that instrumentation could be limited to specific Android API versions.
- Certain features might be too subscettible to false positives cases as there isn't any defined process for defining TPLs candidates like in similarity comparision methods (more research would be required).
- Most modern applications encrypt network traffic so workarounds must be implemented.
- Different obfuscation techniques will still interfere with dynamic analysis approaches, identifier renaming being one of them.
- Different methods have been developed to protect applications from hooking and similar dynamic analysis techniques.

4.4.1 Protections from dynamic analysis

Just like static analysis has limits imposed by obfuscation techniques, dynamic analysis is limited by some checks to blocks specific operations:

- **Root detection**: the application attempts to detect if the device on which is being executed is rooted or not. If it is rooted, actions can be taken that reduce the application allowed operations or even stop it from running at all [54].
- **TLS/SSL pinning**: SSL (Secure Sockets Layer) and TLS (Transport Security layer) are security protocols used to establish cryptographic communications [55]. The pinning process associates a specific certificate to the application, making it trust only the server associated with the certificate.
- **Anti-debug techniques**: the application tries to check if it is being traced or debugged by another process, and usually stops running if that's the case. A very simple example of this is using the ptrace command: if ptrace tries to trace a program that is already being traced, it will return a negative value [56], so a conditional check can be executed before running the application to stop ptrace based analysis.

While these seem like extremely limiting factors, many workarounds have been developed for them, so some instrumentation manages to bypass checks. However, these are problems to take in consideration when performing dynamic analysis.

5. Instrumentation and experiments

5.1 Emulating Android and rooting devices

As explained in Chapter 2, Android Studio is used to emulate Android devices. To communicate with the devices, the Android Debug Bridge (ADB) tool is used [57]. ADB provides a command-line Unix shell that a user can use to execute commands on the device. Each device has its own ADB **daemon**, which runs as a background process, which executes the commands on the device. To manage communication between the daemon and the client sending the commands (that is, the user on the command-line), ADB also runs a server component. Because of the restrictions explained at Section 4.3, ABD shell does not run commands as a superuser. To solve this problem, emulated devices must be rooted, and to reach such objective, the **Magisk** toolset has been chosen [58]. Magisk is a set of open source software for customizing Android images, with one of the modules being **MagiskSU**, which allows for rooting devices. The project, publicly available, installs an application on the Android Virtual Device (AVD), which in turn allows to make the AVD rooted. If everything is successful, an ADB shell connected to the rooted AVD can execute **su** to run a shell as root.

5.2 Testing Environment

The tests for this study were executed on a machine with 16 GB of RAM, a i5 Intel processor, and the Windows 10 OS installed. The work environment has 2 main components: one being the host machine with Windows 10, the other one being a Virtual Machine (VM) running Kali Linux 2025.3 (but many tests were also tried on a Ubuntu VM). The host runs Android studio, which manages the rooted instances of Android devices, which are a Pixel 9 with Android 16 (Google PlayStore enabled) and a Pixel 4a with Android 11 (Google PlayStore disabled). Both sides have ADB installed and the Kali Linux VM can connect to the specific device being run through ADB using specific ports. Inside the Virtual Machine resides the used instrumentation, with each tool being installed through a container or a Python environment to not cause versioning conflicts with the requirements of each part. Since the devices are rooted, it is also possible from Linux to use

adb shell as root to insert and run binaries from inside the emulated devices. The data collected from any experiment is contained inside the VM. This configuration isn't the only possibility available: A full Linux environment is a good alternative, with the main change being that Android Studio and the instrumentation are on the same machine and that no network configuration is required (like port forwarding or firewall rules). While using only Windows is a possibility, it is not suggested as not all instrumentation might be available on the OS.

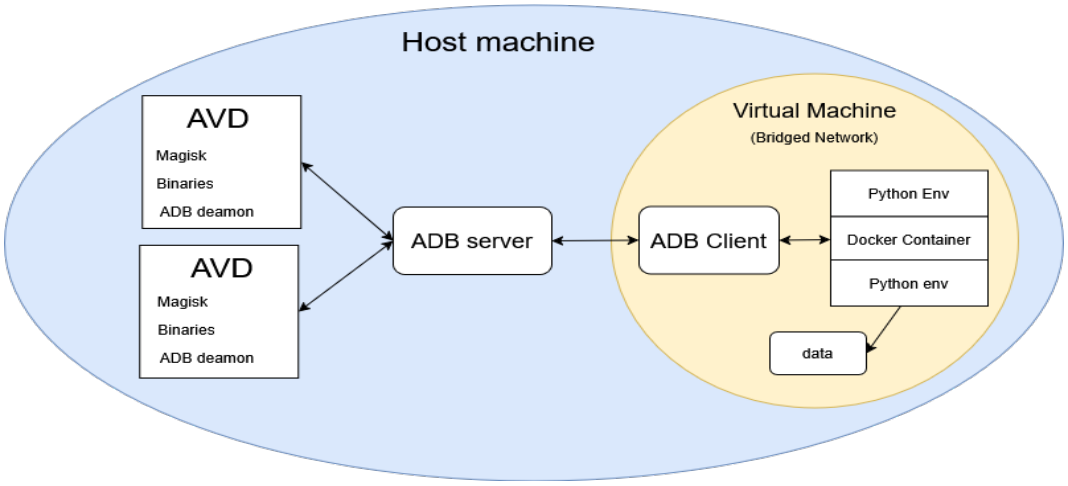


Figure 5.1: Illustration of the workflow

5.3 Frida

Frida is a dynamic instrumentation toolkit which allows for the injection of Javascript code inside of native applications [59]. While the use of Frida is not limited to Android applications, its choice in this study is due to its very large adoption in the Android reverse engineering ecosystem: most existing frameworks are either based on Frida or include Frida in their toolset. Frida has a server component which must be run as a superuser inside the target device, and a client component which reads the JavaScript files used for hooking. While scripts for Frida are written in Javascript or Typescript, Frida APIs are in Python, and both Frida and its extended tools (Frida-tools) can be installed through pip [60]. Frida-Tools is composed by [61]:

- **Frida-Trace**: allows for tracking specified function calls.
- **Frida-ps**: useful for listing processes running on the device.
- **Frida-discover**: discovers functions inside a program.
- **Frida-kill**: kills processes in the device.

5.3.1 The choice of Frida over the Xposed framework

Xposed is a framework which manages to alter the behavior of a system without having access to the applications APKs, with LsPosed [62] being a modern Xposed-like framework example. The way it operates is by inserting an executable inside the */system/bin/app_process* section of the device, which adds an additional jar file to the device classpath before the AVD calls the Zygote process [46] (the classpath being the set of locations where the Virtual Machine looks for classes). In this way, when a function is called, Xposed can insert additional code before or after the call is made. While it can be a reasonable addition to the set of testing tools for this study, Frida has been choosen due to being less invasive to install: the client side can be installed through Pip and the server side can be downloaded and moved through the ADB shell, so no direct installation inside the device through an APK is required (unlike Xposed). Additionally, Frida support and updating are more consistent, while the Xposed framework is split in many independent variations.

5.4 MobSF

MobSF is a security framework which is capable of performing both static analysis and dynamic analysis tasks. It is run through a container, which runs a web interface for managing analysis, and can inject Frida scripts in the applications being executed. While MobSF runs into some limitations, the major one being the possibility of running AVDs only up to version 11 of Android [63] (version 30 for API), it still provides a large amount of information at runtime for making features, and it's been choosen for being constantly kept up-to date. To perform its functionalities, MobSF runs a rooted AVD by modifying the /system partition of the device (which is the reason for the API limitation, since after Android API 30, /system partition is no longer writeable). The framework is run through a container, and it can be managed through a web interface started by the container itself. The MobSF dynamic analyzer can capture logs, HTTP/HTTPS traffic and files used in the application. Custom Frida scripts can also be injected during runtime for either estrapolating additional information or bypassing anti-dynamic analysis techniques described in Chapter 4.

5.5 Libhunter and TraceDroid: cases of TPL detection through Dynamic Analysis

While studies on dynamic analysis for TPL detection are still relatively limited in comparison to static analysis, there are 2 publications worth mentioning. The first one is LibHunter : while it is a static analysis tool, a study was conducted in 2022 which added a dynamic hooking framework that would collect network traffic to

identify libraries [5]. The researchers managed to find many libraries that static analysis tools could not detect due to the TPLs being relatively new. A second instance worth talking about is TraceDroid, which is a framework which collects Android traffic (network requests and execution traces) [4]. A study was conducted with this tool which observed the privacy leakage of TPLs in Android applications, detecting many instances of third party libraries. While both cases have limited coverage in TPLs detection, due to the fact that not all libraries generate network traffic, they represent modern examples of the utility that dynamic analysis can provide in tackling the problem. Unfortunately, testing of these tools results problematic or directly not possible, due to the Libhunter repository being unavailable at the time of writing (giving error 404 on Github), and the Tracedroid repository documentation being limited.

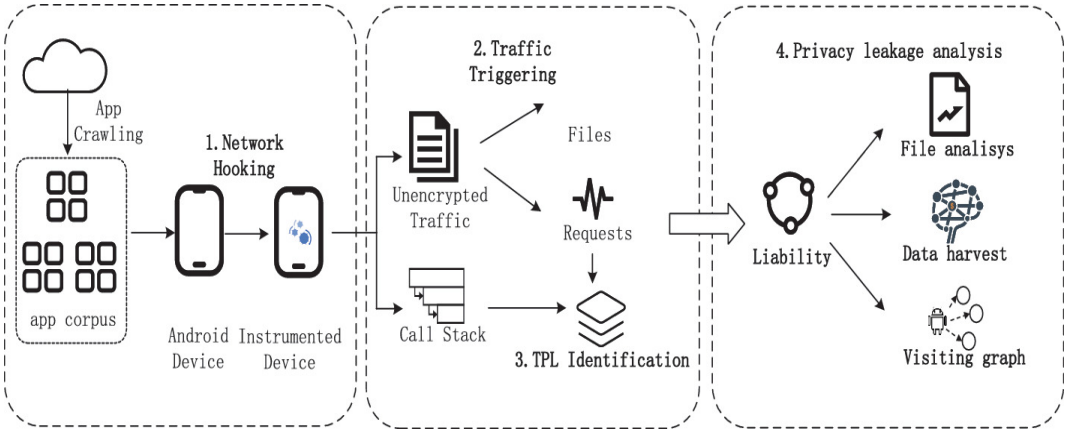


Figure 5.2: Tracedroid work flow [4]

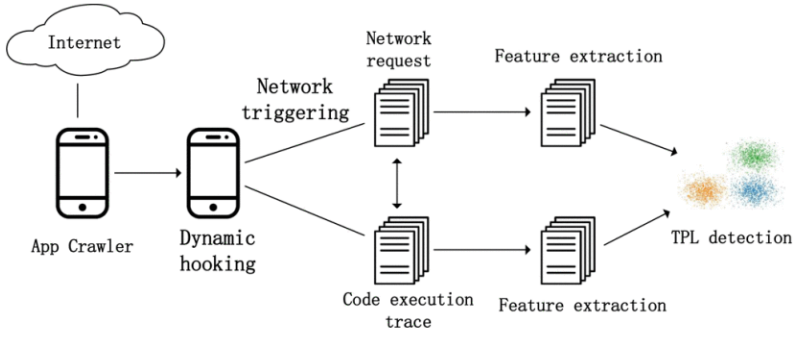


Figure 5.3: LibHunter workflow [5]

5.6 Collecting data in applications

All the testing activity has been performed in the environment described at Section 5.2. The Python script for the collection of the APK files is the work of another Study on TPL detection and is available on Github [64][65]. Installing programs on MobSF was done through the ADB shell using **install-multiple**. The reason for this choice is to avoid certificate issues by merging the split APKs (which is very common for applications that are created as a bundle). Table 5.1 shows the requirements for each tool used, while Table 5.2 shows what data has been collected from the study, with additional speculation on how the data can be used to implement a TPL detection workflow.

Table 5.1: Tools used with their requirements and limitations

Tool	**Requires root**	**Requires APKs**	**Limitations**
Frida	Yes	No	None
MobSF	Yes	Yes	Limited to Android 11 API 30, rooting must be done not with magisk, but with the official scripts.
Frida-tools	Yes	No	None
Tracedroid	Yes	Yes	Version of Frida used for the study is not known.

Table 5.2: Information collected

Extracted information	**Usage**
Loaded classes and methods, including methods signatures	Does not provide the full package structure, but statistical analysis could reveal additional information that can support static analysis.
System calls and API calls	Statistical analysis on what calls get are performed the most during execution could reveal additional information.
HTTP and HTTPS requests	Static analysis cannot detect this, allows for unsupervisionised detection of TPLs. Limited to network libraries.
Accesses to files	Each library operates on a different set of files, potentially allowing creation features.
Application logs	While not being a consistent way for creating recognizable features, it can still provide additional information that static analysis does not have.

As a proof of concept for the possibilities that come with the obtained information, a few simple attempts of TPL detection have been performed using whitelist and blacklist approaches:

- Loaded classes and methods have been collected using a modified version of a publicly available Frida script [66]. Once the data has been obtained, a whitelist approach is performed to find classes related to thrid party libraries.
- HTTP and HTTPs traffic has been obtained using the mobSF dynamic analyzer. Using the user-agent field of HTTP requests, it is possible to identify some network libraries.
- Using MobSF dynamic analyzer, it is possible to dump the application files and use the file names or content to associate a library to it.

6. Results

6.1 Collected information

A small test set of 30 applications of different scopes (social media, public administration, utility etc.) was used to test the instrumentation of Chapter 5. The applications were installed inside the choosen devices and their APKs were extracted. All the informations extrapolated from the analisy process are the results of different Python and Frida scripts, which are available on Github [67]. A sample of third party libraries was took from 2 different websites [68][69].

Table 6.1: TPL sample

Library	**Scope**
OkHttp	Handles HTTP and HTTPS requests
Picasso and Coil	Manages image loading
Retrofit	Manages HTTP APIs through Java/Kotlin
Gson, Moshi	Libraries for managing JSON objects
Eventbus	Manages events through a publish/subscribe bus
Glide	Manages image loading
Lottie	Library for animations
Leakcanary	For detecting memory leaks
Timber	Provides Log utilities
Room	Manages SQL databases
Rxjava	Used for asynchronous programming
Fullstory	Used for various app analytics tasks
Dagger	Used for dependency injection

6.1.1 Loaded classes

The Frida script loading classes at the start of the application created files for a total of more than 32000 classes and 400000 methods istances. For testing pur-

poses, a small whitelist approach was implemented with a Python script that tried to detect TPLs based on class names: when loaded class matches the whitelist, then the candidate library is considered part of the applications. A total 111 candidates were detected across the apps, after manual inspection, 10 were found to be false positives.

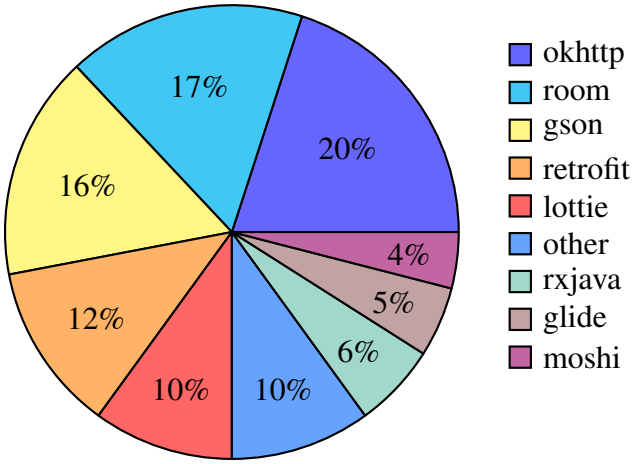


Figure 6.1: Distribution of Android libraries candidates in loaded classes sample set

The study of the dumped files (around 50MB) also showed that all applications have a level of obfuscation using identifier renaming: 4 of the 30 applications did not present any candidate. Sometimes obfuscators perform something partial, obfuscating the classes and methods but keeping the top package name intact, (for example, the class `okhttp3.a`). As explained in Chapter 5 however, looking at which classes get loaded more could allow to make predictions based on statistics, so further study may use such information to make a detection system resilent to identifer renaming techniques. Even in a small test set like the one provided for example, there are classes that get loaded more often than others:

Table 6.2: Most and least frequently loaded class (or enum) for some TPL candidates

Library	Most loaded (instances)	Least loaded (instances)
OkHttp	TlsVersion (19)	EventListener.Factory (1)
Coil	request.CachePolicy(2)	util.Logger.Level (1)
Gson	FieldNamingPolicy (13)	ExclusionStrategy (1)
Room	RoomDatabase.JournalMode (15)	Obfuscated
Moshi	JsonReader.Token (2)	LinkedHashTreeMap.Node (1)
Timber	log.Timber.Tree (3)	log.Timber.Tree (3)
RxJava	internal.util.NotificationLite (5)	core.Completable (1)
Retrofit	CallAdapter.Factory (12)	util.StatusCode (1)
Fullstory	All equal (1)	All equal (1)
Glide	load.ImageHeaderParser.ImageType (4)	load.PreferredColorSpace (1)
Lottie	RenderMode (7)	perf.LottieOrigin (1)

6.1.2 Tracing API/system calls

The idea of analyzing syscalls and low level API of Android is took from Tracedroid. After analyzing the project on the repository, a similar approach was done using Frida-trace. Frida-trace will create basic handlers for intercepting calls, from Java functions to very low level calls (system calls or their ”wrappers”). A set of system calls and low level functions for Linux is choosen, and the stack trace is dumped through the Frida Java interface [70], using **Java.Thread**.

Table 6.3: Choosen syscalls/functions for study [71][72]

System call	Scope
close	files, network
open	files, network
ioctl	files
recvfrom	network
sendto	network
socket	network
pthread_create	multithreading

To keep things simple, a stack trace containing a Class or a function from a target TPL is considered to have responsibility in the happening of the syscall, and

so is counted for the total amount of candidates. An interesting observation of the candidates is how the Dagger library, which was not detected using the loaded classes at the application start, has been found multiple times as a candidate here.

Table 6.4: Syscall and Functions matches

TPL	open	close	socket	ioctl	p_create[1]	sendto	recvfrom
OkHttp	✓	✓	✓	✓	✓	✓	✓
Retrofit	✓	✓	✓	✓	✓	✓	✓
Gson	✗	✓	✓	✓	✗	✗	✗
Eventbus	✗	✗	✗	✗	✓	✗	✗
Glide	✓	✓	✗	✓	✓	✗	✗
Moshi	✗	✗	✗	✗	✗	✗	✗
Lottie	✓	✗	✗	✓	✓	✓	✓
Leakcanary	✗	✗	✗	✗	✗	✗	✗
Timber	✗	✗	✗	✗	✗	✗	✗
Room	✓	✗	✗	✓	✓	✗	✗
Coil	✓	✓	✗	✗	✓	✗	✗
RxJava	✓	✓	✓	✓	✓	✓	✗
Dagger	✓	✓	✓	✓	✓	✗	✗
Fullstory	✗	✓	✓	✓	✓	✗	✗

[1] pthread_create

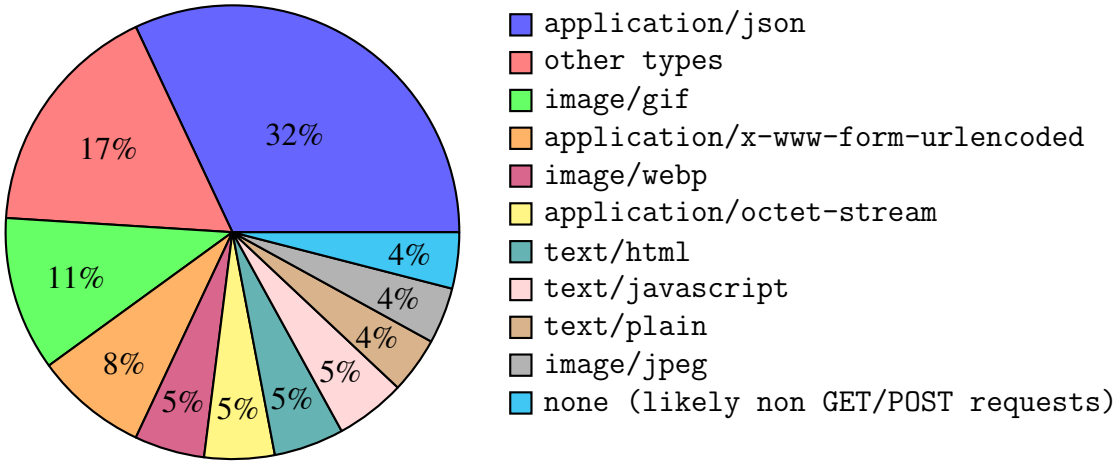
This mapping can serve as a great filter for when obfuscation is present for a target program, and an expansion of the test sample (both for target libraries and target applications) would be worth time.

6.1.3 HTTP and HTTPS traffic

Using MobSF the traffic from applications was intercepted, counting around 3900 requests, with almost a third of them having Json as content-type.

From checking the user agents of the HTTP/HTTPS requests some candidates for network libraries were detected. While this approach is limited to libraries that perform HTTP requests, 2 cases are of interest here: the first one is OkHttp, which is already well-enstablished in this study, the second one being the libraries from the Facebook SDK [73], mostly used for advertisement and integration with Facebook APIs. Not only these targets make their own HTTP agent, but it is also versioned. In the case of the OkHttp library, this is supported by the documentation, which explains that there is a constant pointing out the version that can be used for the user agent [74], while in the case of the Facebook SDK, the source code of the repository shows that the default user agent is `FBAndroidSDK` [75]. While the agent

Figure 6.2: Distribution of content-type fields in HTTP requests



version doesn't necessary always have a 1 to 1 match with the library version, it can still be used to make rough estimates.

Figure 6.3: Versions of FacebookSDK detected from user agent

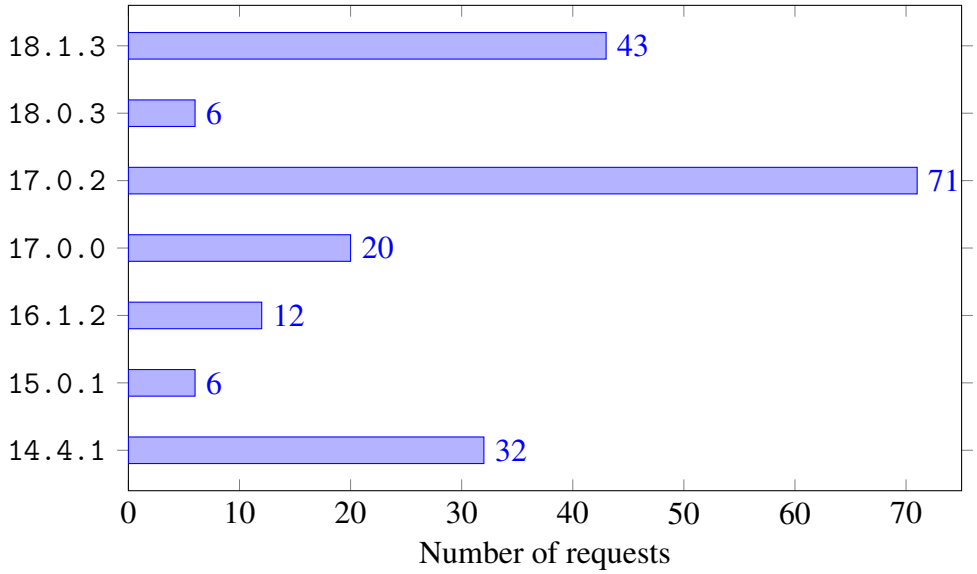
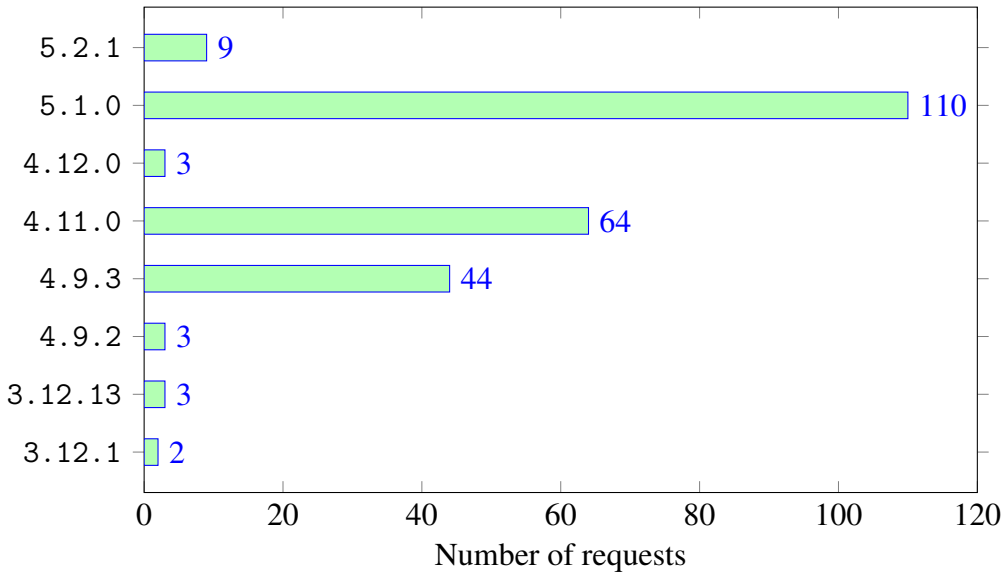


Figure 6.4: Versions of OkHttp detected from user agent



While there is no direct vulnerability reported by CVE for the Facebook SDK, OkHttp has some for older versions. Versions above 4.9.2 being so prelevant is justified by the existance of a vulnerability in versions lower than that causes unwanted Information Exposure [76]. After correlating the requests with the applications, it emerged that 2 targets are using a vulnerable version of OkHttp.

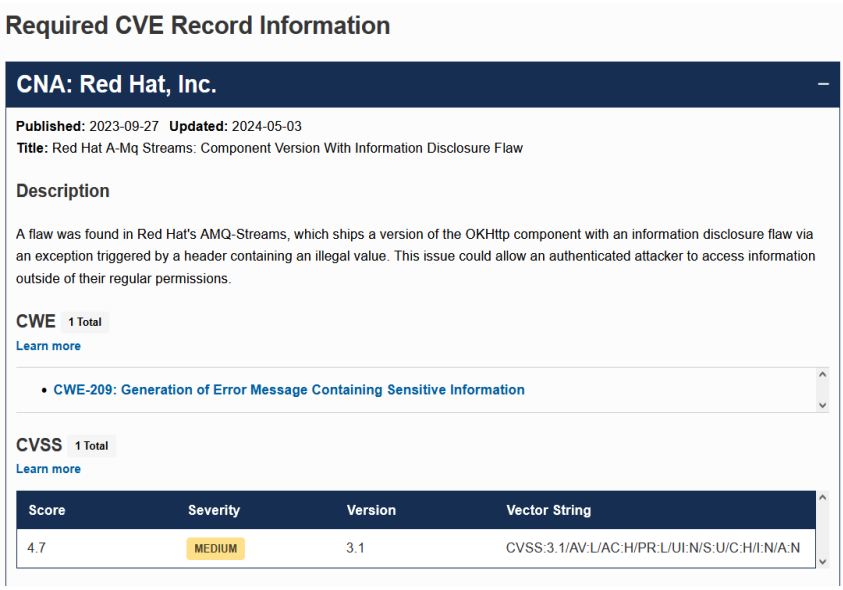


Figure 6.5: CVE-2023-0833 related to OkHttp

6.1.4 Files

MobSF was used to also collect the files of the applications. While theoretically possible to get file names from the Frida-trace scripts on SubSection 6.1.2, MobSF will also save the file contents, giving additional leverage.

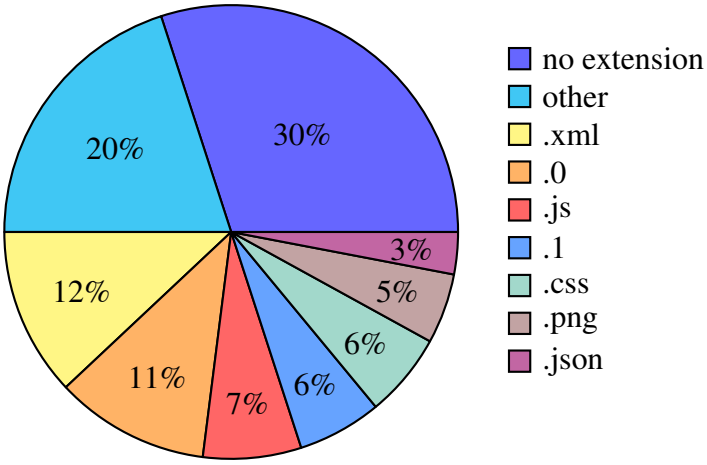


Figure 6.6: File extension distribution

While files are mostly randomized in terms of name, folders aren't, and this allows to detect some TPLs. An important discovery is the presence of the Picasso library in 2 applications, through the presence of its cache.

Table 6.5: Candidates for TPL related files/folders

Candidate	Library	Type	Empty?
`/cache/okHttpClientCache`	OkHttp	Folder	✗
`/cache/okhttp`	OkHttp	Folder	✗
`/cache/OkHttpCache`	OkHttp	Folder	✗
`/cache/coil3_disk_cache`	Coil	Folder	✗
`/cache/fullstory`	Fullstory	Folder	✗
`/cache/picasso-cache`	Picasso	Folder	✓
`/databases/room_database.db`	Room	File	✗

6.2 Limitations encountered

As explained in Chapter 4, there are some countermeasures applications can implement to limit or block dynamic analysis. 3 of the 30 applications for example had root detection mechanism implemented and showed explicit message to the user

that the device was modified, with 2 of them having non-blocking detection mechanisms (that is, it only warns the user but allows to proceed with app usage) and 1 stopping the user from proceeding forward with normal application usage. MobSF worsened the situation, as an additional 4th app detected rooting in a blocking manner. Another problem that emerged with dynamic analysis is overhead: hooking can have a significant impact on the emulator performance and the capacity of executing applications correctly: crashes due to invasive hooks weren't uncommon during data collection. It is also to keep in mind that obfuscation techniques like identifier renaming still posed an obstacle due to limiting the capacity of the whitelisting Python scripts.

One major thing to notice is that correct vesioning of a detected library is way less feasable than static analysis, due to the fact that smaller changes from one version to another are way harder to notice at runtime rather than confronting the bytecode.

6.3 Feasability of making features

A question that this study proposed is if it is possible through dynamic analysis to make similarity-comparision techniques like static analysis does. In order to do so, a set of features that allows to distinguish TPLs from one another is required. Out of the data extracted in Section 6.1, a few possibilities rise:

- Using the Java stack trace, it is possible to map specific class and function signatures to the syscalls performed. This would make both a filter for False Positive detection in already existing systems and a standalone component that can do basic identification of libraries.

- As shown by Tracedroid, it is possible to trace the HTTP/HTTPS requests and the syscalls made by those requests: the reason why this is useful is because one limitation of TPL detection is that most techniques require a dataset for making comparisions, while this approach while being limited to network libraries can make detection unsupervisionised.

- While this study focuses on file accesses, the content of the files themselves could shed more structured information.

In the end, it is clear that some information is more reliable than other: Logs remain the most inconsisten source of data, being the reason why there is not a SubSection dedicated for them despite being captured through MobSF and being mentioned in Chapters 4 and 5.

6.4 Conclusions

The results of the study show that dynamic analysis approaches can provide meaningful additional results in TPL detection studies: with even very simplified search techniques, it was possible to detect CVEs using only runtime information. While the possibilities of this methodology alone cannot cover the problem by itself due to data collected at runtime having limited scope (not all libraries generate network traffic, interaction with json files, etc..), a more hybrid approach with state of the art static analysis techniques has the potential to improve the current threshold on precision, especially when the target is a specific subsection of libraries. Future studies should focus on detecting which features are more suited for the problem at hand and formalize them. Additional inquiry should also be done on how to automate the process: manual interaction with target applications becomes less feasible the bigger the number of targets becomes. A possible solution could be Monkey [77], a tool that performs stress testing on applications, which would help automating the data generation process, but would be limited to elements like network traffic. Touch could be automated through the use of ADB, but working with coordinates would create the problem of making any form of work compatible with all different devices sizes. Another aspect that needs attention is that interaction needs to be able to cover as much as possible of the app use cases, in order to generate as much as unique data as possible for each target. It is also interesting to observe that despite the existence (although scarce) of previous studies related to TPL detection through runtime analysis, Machine Learning approaches seem to be completely out of the picture at the time of writing, which becomes another point of interest for future work. In summary, the detection of third party libraries code in applications is a problem that has met more attention in the latest years, but which is still not solved, requires more research, and could benefit from the less explored alternatives of dynamic and hybrid analysis.

Bibliography

[1] Paul Ratazzi. Understanding and improving security of the android operating system. *ResearchGate*, 2016. https://www.researchgate.net/figure/Figure-APK-file-structure_fig2_316793316.

[2] Laban Ndwaru. Introduction to android art; the next generation of android runtime. *Scientific Figure on ResearchGate*, 2014. https://www.researchgate.net/figure/Android-Boot-process_fig1_313242785.

[3] Fadi Mohsen. Payscan: Detection and security analysis of payment libraries in android apps. *ACM Digital Library*, 2025. https://link.springer.com/article/10.1007/s10207-025-01105-0.

[4] Huajun Cui, Guozhu Mengand Yan Zhang, Weiping Wang, Ting Su Dali Zhu, Xiaodong Zhang, and Yuejun Li. *TraceDroid: A Robust Network Traffic Analysis Framework for Privacy Leakage in Android Apps*. Springer Cham, 2022. Pages 541–556, https://link.springer.com/chapter/10.1007/978-3-031-17551-0_35.

[5] Huajun Cui, Guozhu Meng, Yueqi Li, Yuejun Li, Yan Zhang, and Jiyan Sun. Libhunter: An unsupervised approach for third-party library detection without prior knowledge. 2022. https://ieeexplore.ieee.org/document/9912796.

[6] Data reportal. Digital around the world, 2025. https://datareportal.com/global-digital-overview.

[7] Coin law. Mobile banking statistics, 2025. https://coinlaw.io/mobile-banking-statistics/.

[8] Statista. Digital population worldwide, 2025. https://www.statista.com/statistics/617136/digital-population-worldwide/.

[9] Statcounter. Mobile operating system market share worldwide, 2025. https://gs.statcounter.com/os-market-share/mobile/worldwide.

[10] Android Developer Guide. Android developer verification. https://developer.android.com/developer-verification.

[11] OkHttp official website. https://square.github.io/okhttp/.

[12] Haoyu Wang and Yao Guo. Understanding third-party libraries in mobile app analysis. 2017. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7965410.

[13] Android developer guide, Application fundamentals, app comonents. https://developer.android.com/guide/components/fundamentals.

[14] Android Studio website. https://developer.android.com/studio.

[15] Android developer guide. Analyze your build with the apk analyzer, view file and size information. https://developer.android.com/studio/debug/apk-analyzer.

[16] Appdome: APK file structure. https://www.appdome.com/how-to/devsecops-automation-mobile-cicd/appdome-basics/structure-of-an-android-app-binary-apk/.

[17] Androzoo. https://androzoo.uni.lu/.

[18] Red Hat. Documentation: Java virtual machine. https://docs.redhat.com/it/documentation/red_hat_data_grid/6.4/html/getting_started_guide/java_virtual_machine2.

[19] Beomjun Kim Hyeongseok Oh, Hyung kyu Choi, and Soo mook Moon. Evaluation of android dalvik virtual machine. *Acm Digital Library*, 2012. https://dl.acm.org/doi/abs/10.1145/2388936.2388956.

[20] Ahmad Fikri, Ahmad Fikri, Alfan Presekal, and Riri Fitri Sari. Performance comparison of dalvik and art on,different android-based mobile devices. *IEEExplore*, 2018. https://ieeexplore.ieee.org/document/8864290.

[21] Android developer guide, Applications fundementals. https://developer.android.com/guide/components/fundamentals.

[22] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, , and Wenke Lee. From zygote to morula: Fortifying weakened aslr on android. *IEEExplore*, 2014. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6956579.

[23] Christian Brown and Chris S McDonald. Visualizing berkeley socket calls in students' programs. *ACM Digital Library*, 2007. https://dl.acm.org/doi/abs/10.1145/1268784.1268815.

[24] Android developer guide, Gradle build overview. https://developer.android.com/build/gradle-build-overview.

[25] CVE Overview. https://www.cve.org/About/Overview.

[26] NVD metrics. https://nvd.nist.gov/vuln-metrics/cvss.

[27] Smali introduction. https://kolousek.spselectronic.cz/v4/tandem/cpu/Smali%20Introduction%20Manual.pdf.

[28] Android developer guide, Enable app optimization. https://developer.android.com/topic/performance/app-optimization/enable-app-optimization.

[29] Xiaolu Zhang, Frank Breitinger, Engelbert Luechinger, and Stephen O'Shaughnessy. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *ScienceDirect*, 2021. https://www.sciencedirect.com/science/article/pii/S2666281721002031.

[30] Parvez Faruki, Hossein Fereidooni Vijay Laxmi, Mauro Conti, and Manoj Gaur. Android code protection via obfuscation techniques: Past, present and future directions. *Arxiv*, 2018. https://arxiv.org/abs/1611.10231.

[31] Xian Zhan, Tianming Liu, Yepang Liu, Yang Liu, Li Li, Haoyu Wan, and Xiapu Luo. A systematic assessment on android third-party library detection tools. *IEEE explore*, 2021. https://ieeexplore.ieee.org/abstract/document/9551847/.

[32] Xian Zhan. Detection, analysis and vulnerabilities tracking of android third-party libraries. *Polyu electronic theses*, 2021. https://theses.lib.polyu.edu.hk/bitstream/200/11410/3/5848.pdf.

[33] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Gerard Traynor, and Sascha Fahl. A large scale investigation of obfuscation use in google play. *ACM digital library*, 2018. https://dl.acm.org/doi/10.1145/3274694.3274726.

[34] Xian Zhan, Tianming Liu, Lingling Fan, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. Automated third-party library detection for android applications: are we there yet? *ACM Digital Library*, 2021. https://dl.acm.org/doi/abs/10.1145/3324884.3416582.

[35] Pablo Marchand Ramirez. Empirical study of third-party library detection in ios and android applications. Master's thesis, University of Madrid, 2024. https://oa.upm.es/82866/1/TFM_PABLO_MARCHAND_RAMIREZ.pdf.

[36] APKtool. https://apktool.org/.

[37] AndroGuard. https://github.com/androguard/androguard.

[38] Lige Zhan, Jiang Ming, Jianming Fu, Guojun Peng, Letian Sha, and Lili Lan. The hidden complexities of android tpl detection: An empirical analysis of techniques, challenges, and effectiveness. 2025. https://www.sciencedirect.com/science/article/pii/S016740482500361X.

[39] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. 2016. https://people.svv.lu/derr/publications/pdfs/derr_ccs16.pdf.

[40] Jianjun Huang, Bo Xue, Jiasheng Jiang, Wei You, Bin Liang, Jingzheng Wu, and Yanjun Wu. Scalably detecting third-party android libraries with two-stage bloom filtering. *IEEE Transactions on Software Engineering*, 2022. https://ieeexplore.ieee.org/document/9925074.

[41] Yafei Wu, Cong Sun, Dongrui Zeng, Gang Tan, Siqi Ma, and Peicheng Wang. Libscan: Towards more precise third-party library identification for android applications. 2023. https://www.usenix.org/system/files/usenixsecurity23-wu-yafei.pdf.

[42] Thomas Sutter, Timo Kehrer, Marc Rennhard, Bernhard Tellenbach, and Jacques Klein. Dynamic security analysis on android: A systematic literature review. 2023. https://ieeexplore.ieee.org/abstract/document/10504267.

[43] Logcat command-line tool. https://developer.android.com/tools/logcat.

[44] strace Linux manual page. https://man7.org/linux/man-pages/man1/strace.1.html.

[45] ltrace Linux manual page. https://man7.org/linux/man-pages/man1/ltrace.1.html.

[46] Robo89. *Xposed bridge repository*. https://github.com/rovo89/xposedbridge/wiki/development-tutorial.

[47] Contantin-Alexandru Tudorica and Laura Gheorghe. Context-aware security framework for android. 2016. https://ieeexplore.ieee.org/abstract/document/7913561.

[48] Filippo Alberto Brandolini. Hooking java methods and native functions to enhance android applications security. Master's thesis, Alma Mater Studiorum

– University of Bologna, 2016. https://amslaurea.unibo.it/id/eprint/12257/.

[49] Valerio Costamagna and Cong Zheng. Artdroid: A virtual-method hooking framework on android art runtime. 2015. https://ceur-ws.org/Vol-1575/paper_10.pdf.

[50] Yu an Tan, Shuo Feng, Xiaochun Cheng, Yuanzhang Li, and Jun Zheng. An android inline hooking framework for the securing transmitted data. 2020. https://www.mdpi.com/1424-8220/20/15/4201.

[51] Nikolaos Totosis and Constantinos Patsakis. Android hooking revisited. 2018. https://ieeexplore.ieee.org/abstract/document/8511947.

[52] Wireshark. https://www.wireshark.org/.

[53] sudo Linux manual page. https://man7.org/linux/man-pages/man8/sudo.8.html.

[54] San-Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. Android rooting: Methods, detection, and evasion. 2015. https://dl.acm.org/doi/abs/10.1145/2808117.2808126.

[55] CloudFlare Learning. https://www.cloudflare.com/it-it/learning/ssl/how-does-ssl-work/.

[56] ptrace Linux manual page. https://man7.org/linux/man-pages/man2/ptrace.2.html.

[57] Android Debug Birdge. https://developer.android.com/tools/adb.

[58] Magisk repository. https://github.com/topjohnwu/Magisk.

[59] Frida website. https://frida.re/.

[60] Installing Python packages. https://packaging.python.org/en/latest/tutorials/installing-packages/.

[61] Frida-tools. https://frida.re/docs/examples/android/.

[62] LSposed. https://github.com/LSPosed/LSPosed.

[63] Dynamic analyzer for MobSF. https://mobsf.github.io/docs/#/dynamic_analyzer_docker.

[64] Denise Nanni. Adb apk downloader repository, 2024. https://github.com/dennnanni/adb-apk-downloader.

[65] Denise Nanni. Sicurezza delle applicazioni android: definizione di una metodologia per l'individuazione di vulnerabilità nelle librerie. 2024. https://amslaurea.unibo.it/id/eprint/32387/.

[66] Frida scripts. https://github.com/0xdea/frida-scripts.

[67] Thesis repository. https://github.com/GiacomoBoschi2/Dynamic-analysis-thesis-repository.

[68] Duggu. 13 essential third party android libraries for efficient app development. *Medium*, 2022. https://medium.com/@dugguRK/13-essential-third-party-android-libraries-for-efficient-app-development-b406cc299ed8.

[69] Geeks for geeks android third party libraries. https://www.geeksforgeeks.org/blogs/essential-third-party-libraries-in-android/.

[70] Frida Javascript interface, Java. https://frida.re/docs/javascript-api/#java.

[71] Bionic C library syscalls.txt. https://android.googlesource.com/platform/bionic/+/android-4.2.2_r1/libc/SYSCALLS.TXT.

[72] pthread_create Linux manual page. https://man7.org/linux/man-pages/man3/pthread_create.3.html.

[73] Facebook SDK for Android. https://developers.facebook.com/docs/android/.

[74] OkHttp documentation. https://square.github.io/okhttp/5.x/okhttp/okhttp3/-ok-http/index.html.

[75] Facebook SDK repository. https://github.com/facebook/facebook-android-sdk/blob/0961ae028fc59adb18716d23065fd5214d151dc6/facebook-core/src/main/java/com/facebook/GraphRequest.kt#L209.

[76] CVE-2023-0833. https://www.cve.org/CVERecord?id=CVE-2023-0833.

[77] Monkey tool, Android developer guide. https://developer.android.com/studio/test/other-testing-tools/monkey.