UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE

"Comparative study between static code analysis techniques in the research of possible data and privacy leakages on Android mobile applications."

IVÁN MARCELO CASTILLO CASTAÑON

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Felipe Beroíza
Profesor Correferente: Marcello Visconti

RESUMEN

Resumen— Los dispositivos móviles con sistema operativo Android son los más utilizado alrededor del mundo, cuyo sistema operativo basado en el principio de mínimo privilegio, es decir a través de otorgar solamente los permisos necesarios por la aplicación puede dar como resultado el acceso a información privilegiada por parte de las aplicaciones, lo que busca minimizar las posibles fugas de información en las aplicaciones pero dejando a cargo del usuario de la aplicación la responsabilidad de otorgar o no el permiso. Por lo que se estudiará cómo diversas técnicas de análisis de código estático pueden entregar información respecto de vulnerabilidades, malas prácticas o mal uso de permisos a partir del código fuente al interior de las Apps que resulte en potenciales fugas de información para el usuarios, así como también realizar un análisis de la efectividad de estas técnicas en función de sus hallazgos, rendimiento, complejidad, entre otras.

Palabras Clave— Seguridad, Análisis de código estático, Android, Aplicaciones móviles, fuga de datos.

ABSTRACT

***Abstract*—** Android-based mobile devices are widely used worldwide, and the Android operating system follows the principle of least privilege, granting only necessary permissions to applications. However, this can lead to apps accessing privileged information, placing the responsibility of permission granting on users. This paper explores how various static code analysis techniques can reveal vulnerabilities, poor practices, or permission misuse in app source code, potentially causing information leaks for users. Additionally, the study evaluates the effectiveness of these techniques based on findings, performance, complexity, and other factors.

***Keywords*—** Security, Static code analysis, Android, Mobile applications, Data leakage.

GLOSARIO

API: ***A**pplication **P**rogramming **I**nterface*. Es el conjunto de rutinas, protocolos y herramientas que permiten el desarrollo de aplicaciones.[1]

APK: ***A**ndroid **A**pplication **P**ackage* o Paquete de aplicación Android, es un archivo comprimido en el cual se encuentran los recursos tales como imágenes, permisos, firmas y código fuente de la aplicación.[2]

CWE: ***C**ommon **W**eakness **E**numeration*. Es una lista de debilidades de seguridad comunes en el software, utilizada para identificar y categorizar vulnerabilidades.[3]

DEX: ***D**alvik **ex**ecutable program*, son los archivos utilizados para inicializar y ejecutar aplicaciones desarrolladas para el sistema operativo móvil Android.[4]

JAR: ***J**ava **Ar**chive*. Es un archivo comprimido que contiene clases de Java, recursos y metadatos, utilizado para distribuir bibliotecas y aplicaciones Java.[5]

JSON: ***J**ava**S**cript **O**bject **N**otation*. Es un formato ligero de intercambio de datos que se utiliza para transmitir información estructurada en formato legible por humanos y fácil de interpretar para las máquinas.[6]

OWASP: ***O**pen **W**eb **A**pplication **S**ecurity **P**roject*. Es una comunidad de seguridad dedicada a mejorar la seguridad del software. Proporciona información, herramientas y recursos para identificar y mitigar las vulnerabilidades de las aplicaciones web.[7]

RAR: ***R**oshal **Ar**chive*. Es un formato de archivo comprimido que se utiliza para empaquetar y comprimir archivos y directorios, proporcionando una forma eficiente de organizar y reducir el tamaño de los datos.[8]

USB: ***U**niversal **S**erial **B**us*, sistema de cable que provee alimentación eléctrica y comunicación a computadoras, periféricos y dispositivos electrónicos.[9]

XML: *e**X**tensible **M**arkup **L**anguage*. Es un lenguaje de marcado que define reglas para la

[1] https://developer.mozilla.org/en-US/docs/Web/API
[2] https://developer.android.com/studio/publish/app-signing?hl=es-419
[3] https://cwe.mitre.org/
[4] https://developer.android.com/reference/dalvik/system/DexFile?hl=en
[5] https://docs.oracle.com/en/java/javase/16/docs/api/index.html
[6] https://www.json.org/json-en.html
[7] https://owasp.org/
[8] https://www.loc.gov/preservation/digital/formats/fdd/fdd000450.shtml
[9] https://www.usb.org/

codificación de documentos en un formato legible tanto para humanos como para máquinas, utilizado para estructurar y almacenar datos de manera jerárquica.[10]

ZIP: Es un formato de archivo comprimido utilizado para empaquetar y comprimir uno o más archivos y directorios, facilitando su almacenamiento y distribución.[11]

[10] https://www.w3.org/XML/

[11] https://www.winzip.com/es/learn/file-formats/zip/

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA

1.1 Introducción

Ante la creciente popularidad de los dispositivos móviles durante la última década se ha visto transformada radicalmente la manera en la que interactuamos con el resto, ya sea a través de aplicaciones de mensajería, redes sociales o utilizarlos para el ocio, estos se han vuelto parte fundamental de nuestro dia a dia. Por lo que es necesario preguntarse a uno mismo qué información les estamos otorgando a estas aplicaciones respecto a nuestro diario vivir, qué información obtienen de nosotros sin que lo sepamos, realmente cumplen con los estándares de seguridad que declaran en sus páginas de descarga o si están utilizando los recursos de manera adecuada.

En base a esto surgen diversas técnicas de análisis de aplicaciones, ya sean las estudian su comportamiento que corresponden a análisis de código dinámico o las que evalúan el código fuente de la aplicación el cual es llamado análisis de código estático. Este documento estará enfocado en estas últimas bajo la premisa de que a partir de estas se pueden resolver dudas como qué permisos son los utilizados por las aplicaciones, si existen fugas o mal uso de la información del usuario, si cumplen estándares de calidad en cuanto a código y si la utilización de recursos es la correcta.

A lo largo de esta investigación se iniciará con explicar conceptos que son necesarios para el entendimiento del trabajo que se realizará, como lo puede ser la diferencia a un malware, una mala práctica y una vulnerabilidad, como funciona el sistema operativo Android y cómo son desarrolladas las aplicaciones que serán ejecutadas en este. Posteriormente se plantea la propuesta de solución, la cual corresponde a un estudio comparativo entre diversas técnicas de análisis de código estático, las cuales una vez ya implementadas nos permitan identificar diferencias en la información otorgada por cada una de estas, la aplicabilidad a un conjunto de aplicaciones seleccionadas previamente, como se complementan que cada una otorga entre sí y el rendimiento de estas, en conjunto a plantear una metodología de de implementación y análisis que sea aplicable a técnicas diametralmente distintas.

Para finalmente realizar un análisis de los resultados, obtener conclusiones que nos permitan identificar qué técnicas identifican que tipo de problema, cómo podrían combinarse entre sí para darnos un reporte más completo y que mejoras se harían al trabajo, en conjunto a posibles investigaciones que surgirían desde este trabajo.

1.2 Definición del Problema

Actualmente hay 2.000 millones de usuarios que utilizan el sistema operativo de Android,

los cuales tienen acceso a aproximadamente 2.5 millones de aplicaciones[12]. Debido a la gran cuota de mercado que este Sistema Operativo, tiene dentro del mundo móvil, es que existe una gran cantidad de aplicaciones disponibles para esta plataforma, motivando al desarrollo masivo de Apps, las cuales, día a día, se publican en *Play Store* (Tienda de aplicaciones de Android). Esta tendencia al desarrollo masivo de Apps para Android, ha puesto de manifiesto la presencia de gran cantidad de problemas o malas prácticas dentro del desarrollo de aplicaciones móviles, entre los cuales se pueden listar:

1. Aplicaciones móviles que utilizan permisos que no son informados en sus páginas de descarga o en los *marketplaces* [HuiXu2015] [GokhanKul2018], es decir piden autorización para la utilización de recursos del teléfono como ubicación actual, la cámara, fotos guardadas en el dispositivo, acceso a red de internet u otros, sin haber informado previamente al usuario sobre estos requerimientos.

2. Almacenamiento de información sensible por parte de las aplicaciones, sin consentimiento usuario. [HuiXu2015].

3. Utilización de librerías de terceros, es decir código desarrollado por un tercero que ha sido publicado de tal manera que puede ser utilizado en el desarrollo de aplicaciones con el fin de agilizar este, que podrían llevar a vulnerabilidades en la aplicación. [HuiXu2015] [GokhanKul2018] [AidenPolese2022].

4. Utilización de permisos innecesarios para el objetivo de la aplicación [HuiXu2015].

5. Aplicaciones que tienen comportamientos similares a malwares, como puede ser la extracción de información sin autorización del usuario, la utilización de recursos de manera malintencionada, propagación del mismo sistema a otros sistemas, entre otros. [AsafShabtai2010] [HsuMyatWin2022].

[12] https://www.businessofapps.com/data/android-statistics/

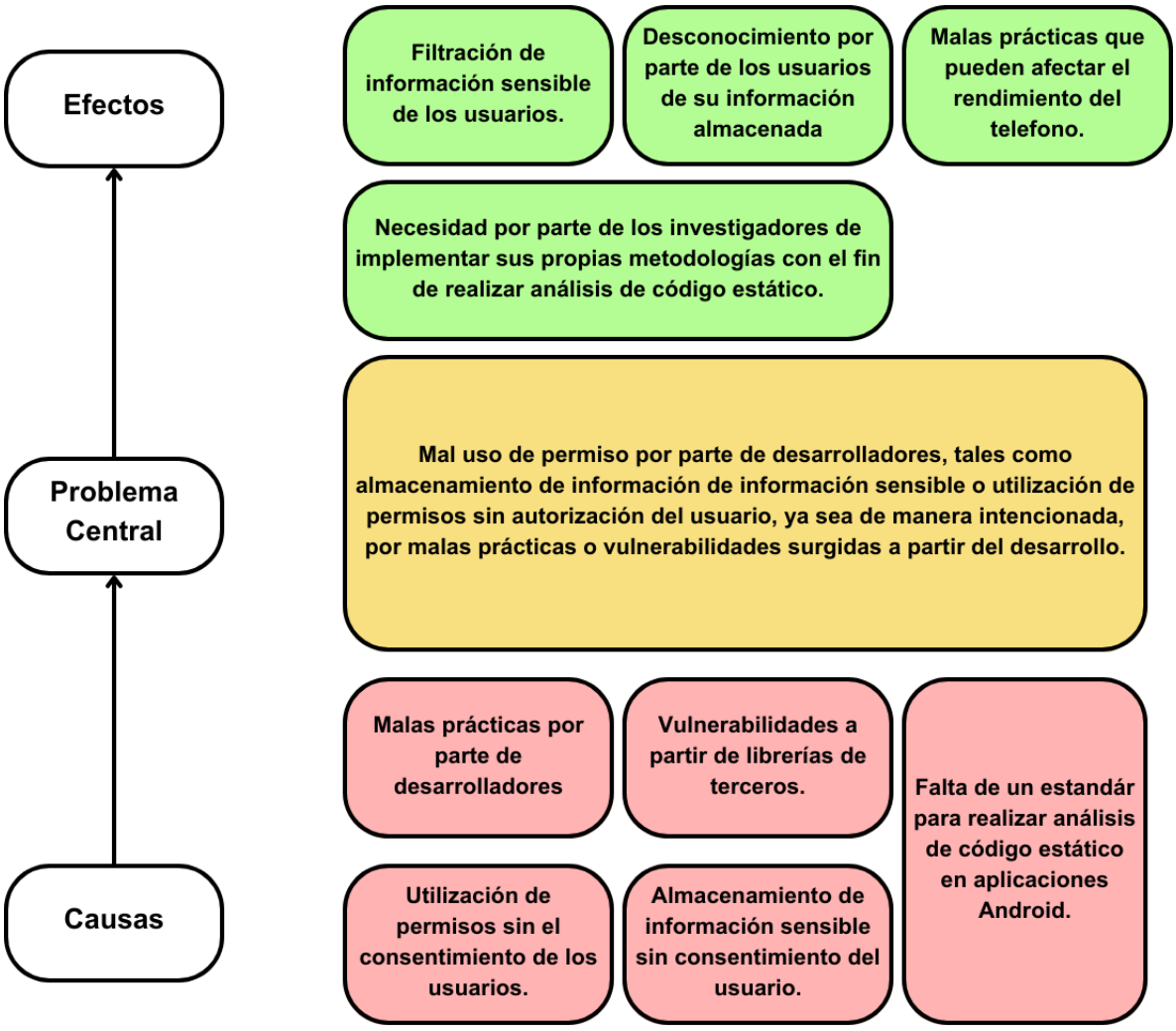


Figura 1: Árbol del problema junto a objetivos generales y específicos.

Fuente: Elaboración propia.

Por otro lado, luego de haber revisado los estudios descritos en [HuiXu2015] [WenyunDai2017] [GokhanKul2018] [QiLi2015] [YuZhang2020] [AsafShabtai2010] [HsuMyatWin2022] [IbrahimSalihu2018] [KristiinaRahkema2022], donde cada grupo de estudio realizó análisis estático en función de distintos objetivos, llama la atención que cada equipo implementó su propia metodología para realizar ingeniería inversa a las aplicaciones de Android, lo que nos lleva a suponer que hace falta una metodología de

trabajo estándar que explique los pasos que son necesarios ante las distintas adversidades que se pueden presentar al momento de analizar el código de las Apps, como también proponer tecnologías que pueden ser utilizadas en cada paso, de forma que se puedan sobrellevar dichos problemas. En resumen a lo anterior, sería útil la existencia de una arquitectura de referencia que sirva como guía para futuros equipos que quieran seguir realizando investigación en esta línea.

1.3 Propuesta de Solución

En base a las se propone la realización de un análisis comparativo entre diversas técnicas de análisis de código estático, el cual tendrá como fin el estudio de la calidad de la información entregada por cada una de las técnicas, su aplicabilidad a un conjunto de aplicaciones, rendimiento en el proceso de análisis y permisos que son utilizados por las aplicaciones, todo esto para lograr entender de mejor manera el contenido que las aplicaciones poseen. De manera subsidiaria, se definirá una metodología de trabajo para el desarrollo, implementación y análisis de las diversas técnicas que pueda ser aplicado de manera transversal a las técnicas.

1.4 Objetivos

1.4.1 Objetivo General

Implementar distintas técnicas de análisis de código estático para analizar el código fuente de diversas aplicaciones móviles que funcionan sobre el sistema operativo Android, con el fin de comparar la calidad de la información que estas otorgan respecto a seguridad y malas prácticas, así como también identificar y evaluar los factores que se ven involucrados en la calidad de la información obtenida.

1.4.2 Objetivos Específicos

- Identificar las vulnerabilidades a analizar a partir de la literatura existente.
- Definir los parámetros de inclusión/exclusión que serán utilizados para seleccionar las técnicas de análisis de código estático a implementar.
- Listar los parámetros de inclusión/exclusión que serán utilizados para elegir las aplicaciones que serán analizadas.
- Esquematizar un *pipeline* de análisis que permita obtener el código fuente de las aplicaciones a partir de su APK, de manera que esta sirva como una arquitectura de referencia replicable por cualquier grupo de interés que desee llevar a cabo

ingeniería inversa en el ámbito móvil.

- Implementar las técnicas seleccionadas en un conjunto de aplicaciones seleccionadas a partir de ciertos criterios de inclusión/exclusión.
- Realizar un análisis de la información que se obtenga con las distintas técnicas y comparar la calidad de estas, las variables que se vieron involucradas en la calidad de las técnicas y la aplicabilidad de estas, así como identificar las brechas de seguridad y malas prácticas más comunes en aplicaciones móviles.

1.5 Estructura del Informe

A continuación abordaremos un aspecto fundamental para el entendimiento de este documento el cual es la estructura que este tendrá. En el primer capítulo contextualizamos el problema que da origen a este informe en conjunto a la propuesta de solución que presentaremos, como también los objetivos que guiarán la investigación, la implementación y el análisis de la solución propuesta. En el segundo capítulo abordaremos el marco teórico y el estado de la literatura que son necesarios para comprender los temas abordados por este documento, tales como conceptos claves, tecnologías que se ven envueltas en la problemática presentada y funcionamiento de estas con el fin de clarificar las decisiones que se toman en el desarrollo de la solución, como también la presentación de posibles aspectos que podrían dificultar el análisis de código estático. En el tercer capítulo se presenta la propuesta de solución más en detalle con información como la metodología de trabajo planteada, las tecnologías a utilizar en el desarrollo, aplicaciones a *testear*, selección de técnicas a implementar y las métricas de selección que serán utilizadas para cada uno de estos aspectos. El cuarto capítulo busca explicar cómo fue implementada la propuesta de solución en cuanto a cada una de las técnicas de interés, como también el levantamiento del entorno de trabajo teniendo como finalidad el facilitar la replicabilidad del sistema en el caso de que algún interesado lo desee. El quinto capítulo aborda los resultados obtenidos a partir de la implementación y el análisis de estos para su mayor comprensión, con el fin de identificar con mayor facilidad la información que cada una de las técnicas implementadas otorgan, como también la capacidad que tienen estas de complementarse. Finalmente en el sexto capítulo se expondrán las conclusiones obtenidas a partir del trabajo realizado, analizando si los objetivos se cumplieron, reflexionando sobre cómo el trabajo fue abordado, formas en las que este podría ser mejorado y finalmente planteando el posible trabajo a futuro que puede surgir desde la investigación realizada.

CAPÍTULO 2: MARCO CONCEPTUAL

A continuación, se describe el marco conceptual que servirá como base para el desarrollo de esta memoria, pues entregará los sustentados teóricos para definir la selección de técnicas de análisis de código estático, selección de aplicaciones a analizar y definiciones de que será considerado una mala práctica, una vulnerabilidad y un *malware*.

2.1 Desarrollo de aplicaciones móviles

Para entender los problemas que surgen en el desarrollo móvil de *Android* es necesario saber que existen diferencias sustanciales entre el desarrollo para equipos de escritorio y equipos móviles. La primera diferencia que debe ser considerada son las limitaciones de *hardware* que inherentemente poseen los dispositivos móviles, tales como el disco duro limitado, distintos protocolos para acceder a redes móviles como lo son el 3G, 4G y Wifi, distintos tipos de entradas según el teléfono como lo pueden ser entradas de audífonos mediante USB o cable auxiliar, entre otras variables [KewelShah2019].

Otro de los principales factores que afectan al desarrollo móvil es los distintos protocolos que poseen los sistemas operativos de cada teléfono al momento de acceder a los recursos, dado que en este tipo de aparatos se le debe solicitar permiso al usuario con el fin de poder acceder a esta información, como ocurre con la lista de contactos, GPS del teléfono, galería, cámara y el resto de las herramientas (sensores) que posea el dispositivo. Lo cual puede generar complicaciones al momento de desarrollar aplicaciones para este tipo de entorno [UmmeMannan2016].

Finalmente, también es necesario que al momento de desarrollar una aplicación para este tipo de entornos se nos proveen distintas tecnologías para lograrlo, siendo estas las aplicaciones web móviles, las aplicaciones nativas y las aplicaciones híbridas [KewelShah2019] [RojasPoblete2016], esto implica que la selección de la tecnología adecuada puede ser de gran impacto como lo podría ser la selección de un framework que sea tanto para desarrollo Android como para iOS, mientras que las aplicaciones web se podrían considerar más universales dado que son independientes del sistema operativo. Este tipo de diferencias implican que la selección de tecnologías para el desarrollo móvil tenga que tomar en consideración factores adicionales como la utilización de recursos, sistemas operativos, *expertise* en el lenguaje de programación a utilizar, entre otros.

2.1.1 Aplicaciones nativas

Son aquellas aplicaciones desarrolladas en el lenguaje utilizado por el sistema operativo del equipo, su mayor beneficio es la cantidad de funciones por defecto que poseen y la utilización de APIs que facilitan el acceso a recursos del teléfono, tales como cámaras, GPS, micrófono u otros.

Por el lado del usuario, son aquellas aplicaciones que tendrán un mejor rendimiento, dado que el acceso a recursos se hace de manera más directa.

2.1.2 Aplicaciones web móviles (Webapps)

Son aquellas aplicaciones cuyo principal entorno de ejecución es el navegador del teléfono, por lo que suelen ser versiones más ligeras que los sitios web principales. Son desarrollados utilizando HTML, CSS y JavaScript.

Su principal limitación es que al ser ejecutadas dentro del navegador no tienen la capacidad de acceder a las APIs del teléfono, lo que causa que solamente puedan acceder a los recursos que el navegador considere necesarios. Esto causa que, en el caso de necesitar la lista de contactos, no será posible acceder a ella si el navegador no ha pedido permisos para esta.

2.1.3 Aplicaciones híbridas

Las aplicaciones híbridas son las que buscan unir los beneficios de ambos tipos de aplicaciones, utilizando los lenguajes de desarrollo web como lo son HTML, CSS y JavaScript, pero que a través de la ejecución de estos en contenedores permitirá el acceso a las APIs del sistema operativo móvil.

El principal beneficio de este tipo de aplicaciones es la capacidad de ser desarrollada en un lenguaje independiente del sistema operativo del teléfono, dado que es el contenedor el encargado de hacer que se ejecute en el dispositivo móvil correspondiente. Sin embargo, el *trade-off* de este tipo de aplicaciones es el impacto al rendimiento que se tiene, dado que la comunicación entre aplicación y API debe ser manejada por el contenedor lo que genera un retraso o demora en la respuesta del sistema operativo a la aplicación, mientras que otro impacto es la dependencia que se tendrá del entorno de desarrollo, porque si este no provee el acceso a algunos recursos a través de sus librerías se volverá imposible tener acceso a estos.

2.2 Android

La plataforma Android surge en el año 2005 por parte de Google y hoy en día posee más de 2.000 millones de usuarios alrededor del mundo[13]. Este sistema operativo, de código abierto, funciona en base al kernel de Linux y cuyas aplicaciones son nativamente desarrolladas utilizando el lenguaje de programación Java, el cual da acceso a las APIs del teléfono.

Una de las principales características por las que Android ha sido un tópico de preocupación es el cómo maneja la seguridad del dispositivo, los archivos de los usuarios y la información delicada. Para esto el sistema operativo tiene políticas de seguridad principalmente enfocadas a los permisos que concede el usuario a las aplicaciones instaladas, tales como la posibilidad de en cualquier momento manejar los permisos con el fin de poder revocarlos a alguna de las aplicaciones y la capacidad del usuario de aceptar o negar el acceso a recursos a aplicaciones instaladas en el dispositivo.

Con el fin de entender de mejor manera la utilización de permisos por parte de las aplicaciones es necesario primero entender el cómo funciona la arquitectura de *Android*, la cual consiste en una capa de aplicaciones, una capa del marco de aplicaciones (*Application Framework*) y la capa de ejecución o *runtime* que funciona en conjunto a la del kernel de Linux. [AnirbanSarkar2019]

[13] https://www.businessofapps.com/data/android-statistics/

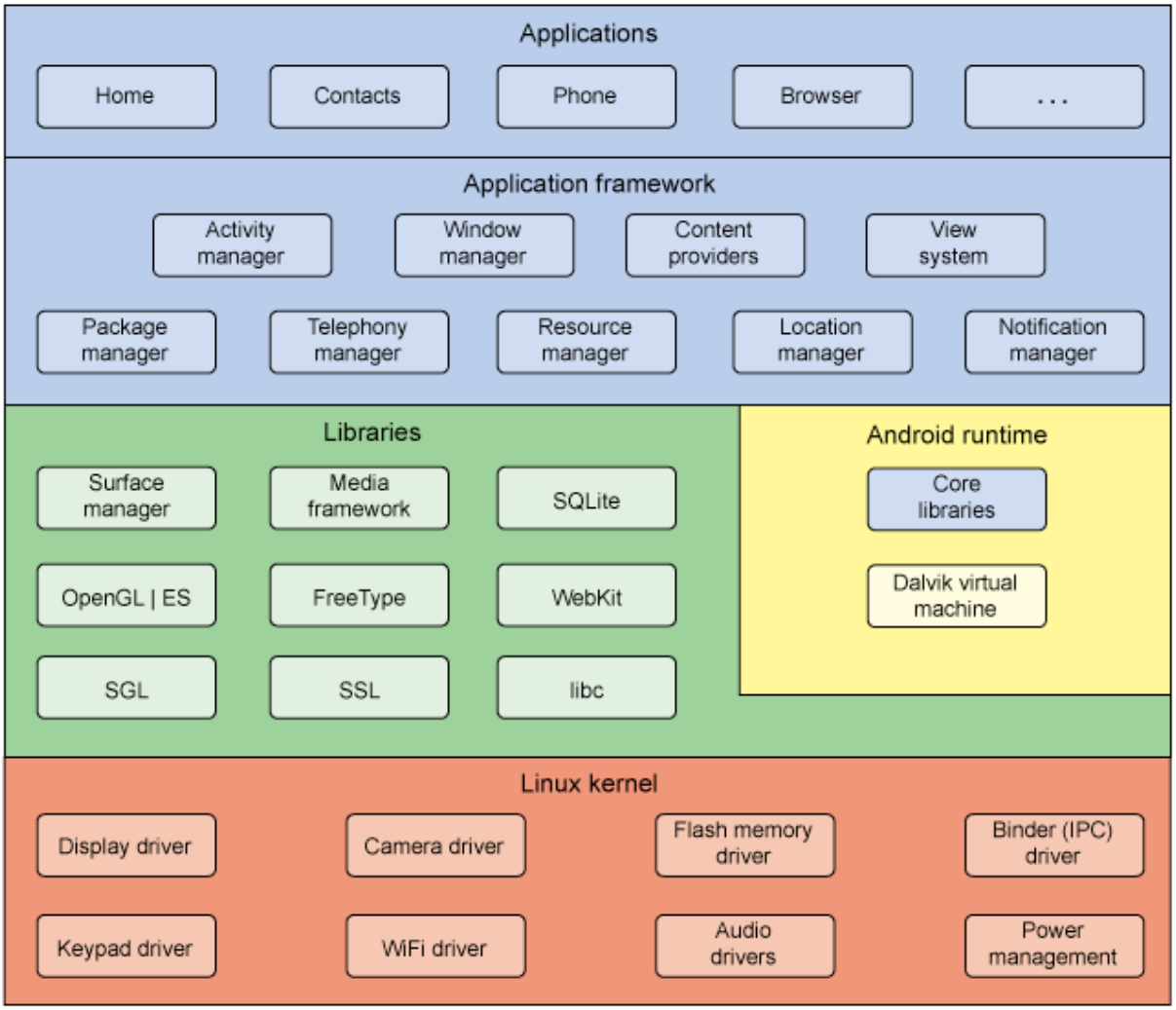


Figura 2: Arquitectura del sistema operativo Android.

Fuente: [RojasPoblete2016]

2.2.1 Capa de Kernel y *Android Runtime*

El kernel de *Android* corresponde a una versión modificada del kernel de Linux para que pueda ser utilizado en dispositivos móviles. Es usualmente asociado a la ejecución de procesos, manejo de la energía y de la memoria. Otra de sus funcionalidades es que es donde se implementan las operaciones que permiten el acceso a recursos como GPS, Cámara, Bluetooth y más.

Por otro lado, el *Runtime* hace referencia a cómo son ejecutadas las aplicaciones de

Android, lo cual va de la mano con el kernel porque una aplicación, al ser ejecutada, crea su propio entorno en una máquina virtual Linux, cuyo principal objetivo es aislar cada una de las aplicaciones del resto con el fin de proveer una capa de seguridad al sistema y, por otro lado, limitar el acceso a recursos dado que es el sistema Linux virtualizado el que tendrá un kernel propio con distintos accesos a recursos, diferenciándolo del resto de aplicaciones y del teléfono en sí, en otras palabras, cada aplicación no tendrá conocimiento del resto de aplicaciones ni acceso a todos los componente que estén en el kernel del teléfono.

2.2.2 Marco de aplicaciones (*Application Framework*)

La principal función de esta capa es ser el intermediario entre el kernel y el código en Java, siendo quien les da acceso a recursos claves a la aplicación. Entre las funcionalidades que esta capa provee, están las bibliotecas de software que proveen código predefinido a las distintas aplicaciones del teléfono, como lo pueden ser acceso a la cámara, gestión de audio, gestión de contactos, etc. Esta capa también provee distintas herramientas para el desarrollo de aplicaciones de Android y es parte fundamental del sistema operativo.

2.2.3 Capa de Aplicaciones

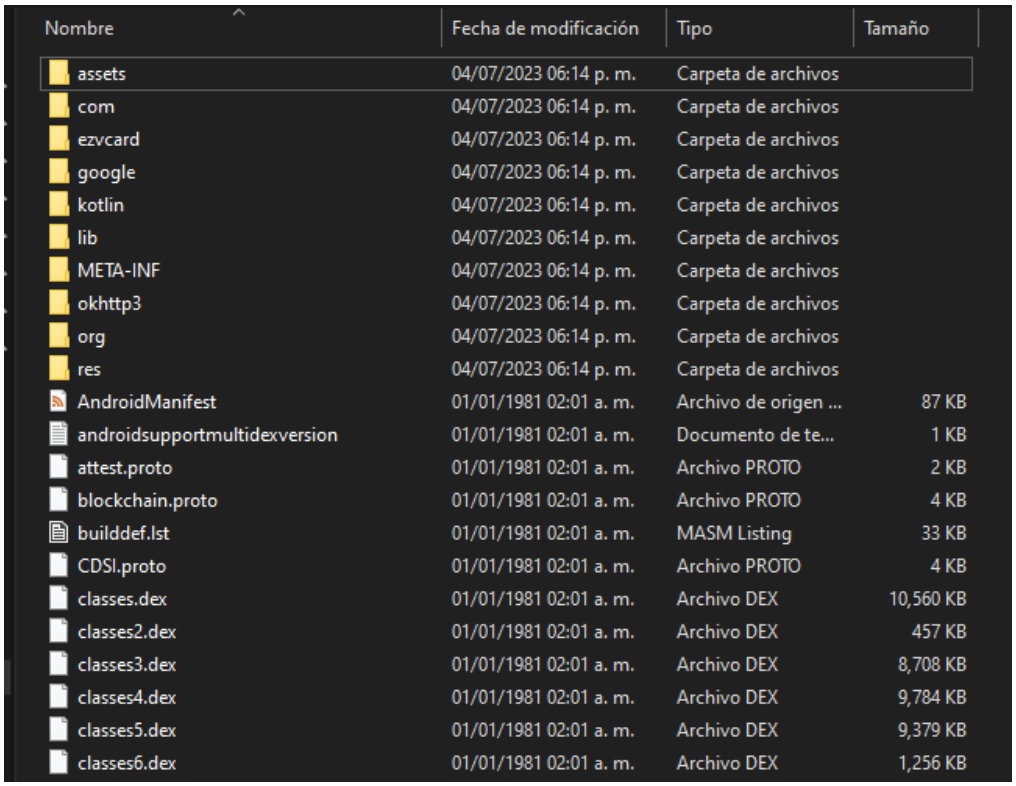
En esta capa es donde las aplicaciones son ejecutadas, aquí se encuentran tanto aplicaciones que provee el distribuidor del dispositivo como lo puede ser el navegador, el gestor de contactos, etc. Es importante destacar que por ejemplo la aplicación de gestor de contactos es aquella con la que el usuario interactúa, pero su funcionalidad se encuentra en el marco de aplicaciones que es la que accede al kernel del teléfono para extraer la información y funcionalidades de más bajo nivel.

2.3 Contenido de una aplicación Android

El método de distribución de aplicaciones Android es mediante archivos comprimidos en formato APK (**A**pplication **P**ac**k**age) que puede ser descomprimido en formato ZIP o RAR en un computador con el fin de ver su contenido, el cual consiste principalmente en paquetes en formato JAR de Java. Una aplicación Android contiene, usualmente, lo siguiente:

- **MANIFEST.MF**: Enumera todos los ficheros incluidos el APK junto con su hash codificado en base 64.
- Directorio **lib:** Contiene las librerías compartidas compiladas.
- Directorio **res:** Contiene los recursos de la aplicación, como lo pueden ser imágenes o audios.

- Directorio **assets:** Contiene los recursos sin procesar.
- **AndroidManifest.xml**: Archivo XML donde se declaran los permisos de la aplicación, componentes y características de uso.
- Directorio **classes.dex**: Contiene el código Java compilado en formato DEX para que la máquina de Android pueda interpretarlo.
- Fichero **resources.arsc:** Contiene recursos pre-compilados.



Nombre	Fecha de modificación	Tipo	Tamaño
assets	04/07/2023 06:14 p. m.	Carpeta de archivos	
com	04/07/2023 06:14 p. m.	Carpeta de archivos	
ezvcard	04/07/2023 06:14 p. m.	Carpeta de archivos	
google	04/07/2023 06:14 p. m.	Carpeta de archivos	
kotlin	04/07/2023 06:14 p. m.	Carpeta de archivos	
lib	04/07/2023 06:14 p. m.	Carpeta de archivos	
META-INF	04/07/2023 06:14 p. m.	Carpeta de archivos	
okhttp3	04/07/2023 06:14 p. m.	Carpeta de archivos	
org	04/07/2023 06:14 p. m.	Carpeta de archivos	
res	04/07/2023 06:14 p. m.	Carpeta de archivos	
AndroidManifest	01/01/1981 02:01 a. m.	Archivo de origen ...	87 KB
androidsupportmultidexversion	01/01/1981 02:01 a. m.	Documento de te...	1 KB
attest.proto	01/01/1981 02:01 a. m.	Archivo PROTO	2 KB
blockchain.proto	01/01/1981 02:01 a. m.	Archivo PROTO	4 KB
builddef.lst	01/01/1981 02:01 a. m.	MASM Listing	33 KB
CDSI.proto	01/01/1981 02:01 a. m.	Archivo PROTO	4 KB
classes.dex	01/01/1981 02:01 a. m.	Archivo DEX	10,560 KB
classes2.dex	01/01/1981 02:01 a. m.	Archivo DEX	457 KB
classes3.dex	01/01/1981 02:01 a. m.	Archivo DEX	8,708 KB
classes4.dex	01/01/1981 02:01 a. m.	Archivo DEX	9,784 KB
classes5.dex	01/01/1981 02:01 a. m.	Archivo DEX	9,379 KB
classes6.dex	01/01/1981 02:01 a. m.	Archivo DEX	1,256 KB

Figura 3: Ejemplo de contenido de un APK para Signal.apk.

Fuente: Elaboración propia

2.4 Acceso a permisos en Android

Tal como se explicó en el punto (2.2.1) cada una de las aplicaciones son ejecutadas de manera aislada dentro de su propia virtualización de Linux para evitar el acceso de esta a procesos de otras aplicaciones y para limitar su acceso a permisos, los cuales pueden ser obtenidos de 2 manera:

- Poseer los permisos por defecto, en el caso de ser aplicaciones que provee el fabricante del dispositivo.

- A través del consentimiento del usuario al momento de utilizar la aplicación y en el momento que estos sean necesarios por primera vez.

Pero este segundo punto posee un problema sustancial en cuanto a la seguridad de los usuarios, el cual es que según estudios como los de Amukelani Ngobeni [AmukelaniNgobeni2019] el 42.9% de los usuarios no leen los permisos al momento de instalar las aplicaciones. Lo cual es preocupante dado que estos son los que dan acceso a información como el GPS, archivos privados, lista de contactos, historial de mensajes SMS y otros, lo cual combinado a que una vez dado el permiso al recurso la aplicación no posee obligación alguna de notificarle al usuario si es que está accediendo a alguno de estos recursos en segundo plano o si está haciendo un respaldo de información sensible sin el respectivo consentimiento del usuario, como nos plantea [HuiXu2015].

2.5 Marketplaces de aplicaciones

Como se planteó previamente en los puntos (2.3) y (2.2.3) son las aplicaciones y sus permisos el mayor peligro para los usuarios, por lo cual es necesario comprender cómo es que estas son obtenidas por las personas.

Google ofrece a sus usuarios la plataforma Play Store, la cual corresponde a un mercado de aplicaciones donde se estima que actualmente hay cerca de 2 millones de estas publicadas[14], las cuales para ser publicadas deben pagar una cuota de 25 dólares, firmar la aplicación por el desarrollador, permisos necesarios para que la aplicación funcione y una descripción de esta. A pesar de esto la tienda de aplicaciones de Google presenta la posibilidad de publicar aplicaciones cuyos permisos no correspondan a su descripción, como sería el caso de un videojuego que podría almacenar información de los contactos, por ejemplo, por lo que el criterio de instalar o no la aplicación queda en manos del usuario, de los cuales una pequeña parte efectivamente leen los permisos que la aplicación les solicita. [ZhengyangQu2014]

Además de la Play Store, los usuarios de Android tienen la posibilidad de descargar directamente los APK de internet con el fin de instalarlo en sus equipos. Para eso los usuarios deben aceptar el riesgo que significa el instalar aplicaciones de terceros, dado que Google no tiene control alguno de los permisos y recursos a los cuales estas aplicaciones acceden [Uma2019].

[14] https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

2.6 Librerías de terceros

Dentro del desarrollo de aplicaciones en Android una de los componentes más utilizados son las librerías desarrolladas por terceros, dado que nos proporcionan funciones o componentes ya desarrollados y probados por múltiples usuarios lo que permite acelerar el desarrollo de aplicaciones e incrementar la calidad de estas. Sin embargo es necesario identificar los riesgos asociados a utilizar código de terceros dentro de nuestra aplicación como lo pueden ser vulnerabilidades dentro de las librerías que son traspasadas a las aplicaciones que la importan, las posibilidad de tener código maliciosos por parte del desarrollador de la librería como lo puede ser la recopilación de información sin consentimiento de quien importa la librería o la falta de actualizaciones de estas que pueden significar que vulnerabilidades no sean solucionadas por los desarrolladores.

Por lo tanto, es de vital importancia que los desarrolladores tomen en cuenta distintos factores al momento de importar librerías como saber quien las desarrolla, cuales son los periodos entre actualizaciones o la posible existencia de vulnerabilidades conocidas [AidenPolese2022].

2.7 Vulnerabilidades, malas prácticas y *malware*

A pesar de que los términos vulnerabilidad, malas prácticas y *malware* van asociados a la ciberseguridad presentan diferencias entre sí.

Las vulnerabilidades corresponden a defectos del sistema o debilidades de este que pueden ser identificadas y explotadas por un tercero con el fin de comprometer el sistema. Un ejemplo de estas son las inyecciones de código SQL por la falta de verificaciones en el sistema.

Las malas prácticas, en tanto, se refieren a acciones por parte de los desarrolladores que pueden o no causar debilidades en el sistema y dar origen a una vulnerabilidad. El ejemplo de estas sería la falta de verificación de usuarios, que como se dijo anteriormente pueden dar origen a vulnerabilidades como las inyecciones de SQL.

Por otro lado, los malware son software diseñado con la intención de causar daño y de dañar sistemas. Existen diversos tipos de malwares como lo son troyanos, keyloggers,

ransomwares y muchos más, los cuales son usualmente transmitidos por medio de correos electrónicos o, en el caso de Android, a través de descargas maliciosas, como puede ocurrir al descargar una aplicación desde un marketplace de terceros que contenga puede ser una versión modificada de una aplicación real en la cuál se inyecta código malicioso dentro.

2.8 Técnicas de análisis de código

Ahora que ya fue explicado como funciona el entorno de ejecución de Android, los permisos, la obtención de aplicaciones y que son considerados potenciales riesgos en las aplicaciones, es que es posible definir como las aplicaciones pueden ser analizadas, siendo las principales técnicas el análisis de código estático y el análisis de código dinámico [Uma2019].

2.8.1 Análisis de código estático

Las técnicas de análisis de código estático son aquellas que buscan la detección de *malware*, malas prácticas o vulnerabilidades a partir del código fuente de la aplicación. Para lograr este tipo de análisis es necesario obtener el APK de la aplicación y a través de procesos de ingeniería inversa, transformar en código el cual pueda ser analizado por distintas técnicas, entre las que se encuentran:

- Análisis de comportamiento: Extracción de patrones a partir de la semántica del código con el fin de obtener un comportamiento, el cual es comparado con alguno de los malwares conocidos.
- Análisis de permisos: A través del análisis del código es posible identificar posibles utilizaciones de permisos que no han sido notificadas a los usuarios o que son utilizados en momento donde el usuario no es notificado.

2.8.2 Análisis de código dinámico

El análisis de código dinámico se basa en permitir a la aplicación funcionar dentro de un entorno de ejecución controlado como lo puede ser una virtualización dentro del dispositivo, el principal beneficio de este tipo de análisis es que se puede reconocer el comportamiento de la aplicación en ejecución y puede ser comparado con un *malware* a través de información como el uso de datos móviles, de memoria o de otros recursos del teléfono. La principal complejidad, en comparación al análisis de código estático, es que requiere un mayor esfuerzo para ser implementada y configurada con el fin de obtener la información deseada.

2.9 Dificultades al momento de analizar código de terceros

Debido a la facilidad con la cual los APK pueden ser descomprimidos, es que los fabricantes de aplicaciones deben tomar medidas de seguridad con el fin de proteger la lógica de los negocios, estructuras internas, algoritmos confidenciales, código encargado de manejar información sensible y otro tipo de información importante que puede ser obtenible a partir del código fuente de las aplicaciones [Geunha2022].

A continuación, se explicarán algunas de las medidas de seguridad que utilizan los fabricantes con el fin de proteger sus productos.

2.9.1 Ofuscación

La ofuscación de código es la técnica asociada a hacer el código inentendible, pero sin afectar su funcionalidad. Para lograr esto se deben aplicar diversas transformaciones sintácticas al código, pero manteniendo la semántica de este [Dharmalingam2022].

Para lograr este resultado existen distintos tipos de técnicas, según Dharmalingam et al. [Dharmalingam2022] existen las de ofuscación de capas, ofuscación de control, ofuscación de clases, ofuscación de métodos y ofuscación de información. Mientras que para otros autores como Geunha et al. [Geunha2022], las categorías son renombrar identificadores, encriptación de *strings*, ofuscación del flujo de control y ofuscación del pensamiento, pero a pesar de las diferencias en nombre sus objetivos son similares, para explicar el funcionamiento de estas técnicas utilizaremos la categorización de Dharmalingam, dado que esta separa la ofuscación de las capas con la del flujo de control, lo cual a futuro puede ser beneficioso para identificar específicamente el tipo de ofuscación en el código fuente de las aplicaciones que serán analizadas.

Tipo de Ofuscación	Descripción	Ventajas y Desventajas
Ofuscación de Capas	Esta técnica agrega identificadores sin sentido, instrucciones sin una utilidad real y separa bloques de código que van unidos en diversos espacios de memoria.	+ Afecta la legibilidad del código. - Impacto leve en el uso de memoria.
Ofuscación de Control	Esta técnica tiene como objetivo afectar el flujo de control de un programa al agregar flujos falsos, complicando los reales, asignando tareas de una clase a otras o haciendo controles implícitos.	+ Dificultades para hacer el grafo de control y entorpece el análisis. - Impacto importante en el rendimiento.
Ofuscación de Clases	Esta técnica afecta directamente a las clases en Java, separándolas y clonando su contenido en otras secciones del código.	+ Entorpece la legibilidad del código. - Impacto leve en el uso de memoria.
Ofuscación de Métodos	En esta técnica en específico existen 2 variantes la ofuscación en línea o interna (inline) y la ofuscación de contorno (outline). La ofuscación interna se encarga de que, si un método llama a otro, entonces puede hacer que el código del método llamado se encuentre dentro del método que lo llama. La ofuscación de contorno extrae declaraciones dentro del método y genera un método nuevo, el cual será llamado por el método original.	+ Entorpece la legibilidad del código. + Entorpece el seguimiento de llamadas de funciones/métodos. - Afecta el rendimiento dado que hay un mayor llamado de funciones/métodos. - Afecta levemente el uso de memoria dado que se crean métodos nuevos.
Ofuscación de Información	Esta técnica toma tipos de datos comunes como lo son strings, arreglos o enteros y los transforma a través de uniones, codificaciones y otro tipo de técnicas posibles para este tipo de datos.	+ Entorpece el análisis del código. + No tiene un efecto en el rendimiento, dado que solo modifica la legibilidad.

Tabla 1: Ventajas y Desventajas según método de ofuscación.

Fuente: Elaboración propia.

A pesar de los beneficios de seguridad que poseen las técnicas de ofuscación y una gama de herramientas que permiten llevar a cabo este proceso de manera más fácil, tales como Proguard, Allatori, Obfuscapk y DexGuard, existe el problema de que el aplicar estas técnicas a las aplicaciones, se genera un trade-off en rendimiento, dado que estas mismas herramientas funcionan bajo el principio de ofuscar todas las clases al máximo posible, lo que genera que muchos desarrolladores tengan dudas de utilizar o no las herramientas existentes para ofuscar su código [HuiXu2017].

2.9.2 Código Ofuscado y Play Store

Considerando la ofuscación de aplicaciones, una de las dudas que surge es cómo los *marketplaces,* y específicamente Play Store, trabajan con estas técnicas de protección. Para disminuir el número de aplicaciones maliciosas, Play Store aplica un proceso de revisión a las apk's con el fin de verificar que cumplan con las políticas de la plataforma, donde uno de los pasos dentro del proceso, es el analizar el código fuente.

Como parte de este proceso entra en conflicto el análisis de código con la ofuscación que se utiliza para proteger las aplicaciones, es que Play Store aplica técnicas de análisis de código estático en la búsqueda de código malicioso o vulnerabilidades, como un análisis de código dinámico, donde el código se ejecuta en la búsqueda de comportamiento similar al de un malware.

En resumen, la Play Store acepta aplicaciones ofuscadas, promueve el uso de la ofuscación y ha desarrollado técnicas para analizar este tipo de aplicaciones.

2.9.3 Código Smali

El código Smali corresponde a un lenguaje de bajo nivel utilizado para el desarrollo en el sistema operativo Android, dado que su función es el escribir y modificar con el fin de ser compilado en código Dalvik[15] o a través de instrucciones para la máquina virtual que ejecuta Android para la aplicación. Es principalmente utilizado para realizar ingeniería inversa de aplicaciones debido a que permite encontrar vulnerabilidad y modificarlas, mientras que un uso que posee, pero que no es el estándar, es para desarrollar aplicaciones que requieren un alto rendimiento[16]. A continuación se presentan algunas de las instrucciones más utilizadas en Smali:

[15] https://source.android.com/devices/tech/dalvik/dalvik-bytecode?hl=es-419
[16] https://github.com/JesusFreke/smali

Instrucción	Descripción
move	Copia un valor de un registro a otro registro.
const	Carga una constante a un registro.
add	Suma dos valores en registros y guarda el resultado en otro registro.
sub	Resta dos valores en registros y guarda el resultado en otro registro.
mul	Multiplica dos valores en registros y guarda el resultado en otro registro.
invoke-direct	Llama al método directamente y espera su resultado.
invoke-virtual	Llama a un método virtual y espera su resultado.
invoke-static	Llama a un método estático y espera su resultado.
return	Devuelve un valor de un método.
.method	Define un nuevo método.
new-array	Crea un nuevo arreglo con un tamaño fijo.
aput	Asigna un valor a un elemento del arreglo.
aget	Obtiene el valor de un elemento del arreglo.
if-eq	Salta a una etiqueta si dos valores son iguales.

Tabla 2: Ejemplos de instrucciones de Smali

Fuente: Elaboración propia.

A partir de estas instrucciones es que podemos ejemplificar cómo se utiliza este lenguaje a través de un ‘Hola Mundo’.

```
.class public HelloWorld
.super Ljava/lang/Object;

.method public static main([Ljava/lang/String;)V
    .registers 2

    sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
    const-string v1, "Hello, World!"
    invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/Strin

    return-void
.end method
```

Figura 4: Hola Mundo en Smali

Fuente: Elaboración propia

Entre las definiciones, se puede ver que primero se crea la clase '*HelloWorld*' con su método '*main*', posteriormente se obtiene el objeto '*System*' que posee el método '*out*' que es el que será utilizado posteriormente a través de una invocación para hacer que el string '*Hello, World!*' almacenado en el espacio de memoria 'v1' sea mostrado por pantalla.

El principal motivo por el cual Smali es una de las herramientas de interés en este estudio es la capacidad de obtener código fuente que puede ser analizado con el fin de entender las aplicaciones en búsqueda de vulnerabilidades o comportamientos similares a malware, pero a pesar de esto hay que tener en cuenta una serie de factores que complican el análisis a través de este lenguaje como lo son el requerimiento de entender las instrucciones de código Dalvik para entender lo que se está viendo. La poca cantidad de documentación que este lenguaje posee y, sumado a que la modificación de código Smali podría significar la violación de propiedad intelectual, implican que el código que será analizado, no sera modificado de ninguna forma con el fin de evitar cualquier problema asociado [Arnatovich2018].

2.9.3 Código Dalvik

El código Dalvik, es el código de bajo nivel ejecutable por las Máquinas Virtuales Dalvik que era utilizada por versiones anteriores de Android. A diferencia de sistemas operativos convencionales, Android desarrolló su propio sistema operativo para la ejecución de aplicaciones en entornos móviles, el cual tiene como principal ventaja que es código ejecutable en dispositivos con recursos limitados, lo que aumenta la portabilidad de las aplicaciones a una mayor cantidad de dispositivos móviles.

Este código es generado a partir de Java o Kotlin, el cual es compilado en bytecode Java que corresponde a los archivos *.class* del APK y luego estos son procesados como archivos DEX por el sistema operativo.

A pesar de esto, a partir de Android 5.0 se decidió modificar el sistema operativo Android para que comenzara a funcionar con Android Runtime como entorno de ejecución, el cual tiene un enfoque de compilación anticipada, lo cual mejora aún más el rendimiento y eficiencia de las aplicaciones. Sin embargo, aunque Dalvik ya no esté siendo utilizado por versiones más modernas de Android, sigue siendo relevante dado que otorga retrocompatibilidad a las aplicaciones.

Instrucción	**Descripción**
mover vA, vB	Mueve el contenido de un registro que no es de objeto a otro.
return-object vAA	Return from an object-returning method.
const/4 vA, #+B	Move the given literal value (sign-extended to 32 bits) into the specified register.
monitor-enter vAA	Acquire the monitor for the indicated object.
array-length vA, vB	Store in the given destination register the length of the indicated array, in entries
new-array vA, vB, type@CCCC	Construct a new array of the indicated type and size. The type must be an array type.
new-instance vAA, type@BBBB	Construct a new instance of the indicated type, storing a reference to it in the

	destination. The type must refer to a non-array class.
throw vAA	Throw the indicated exception.
goto +AA	Unconditionally jump to the indicated instruction.
if-test vA, vB, +CCCC if-eq if-ne if-lt if-ge if-gt if-le	Branch to the given destination if the given two registers' values compare as specified.

Tabla 3: Ejemplo de instrucciones Dalvik.

Fuente: Catálogo de instrucciones Dalvik proporcionado por Android[17].

2.10 Análisis de la Literatura

A partir de la información teórica planteada anteriormente con respecto al análisis de código estático, es que se buscaron documentos cuyo principal foco haya sido este tipo de análisis, pero no se excluyeron aquellos que consideraban técnicas combinadas entre análisis estático y dinámico. Esto con el fin de analizar las distintas técnicas que están siendo utilizadas actualmente, no solo para el análisis de seguridad sino que también en otras áreas como lo son el diseño de interfaces, esto para posteriormente seleccionar técnicas interés a partir de métricas por definir, categorizar las distintas técnicas encontradas y realizar una recolección de herramientas que están siendo utilizadas en las investigaciones presentadas en los últimos años y así estudiar su aplicabilidad en este trabajo.

2.10.1 Técnicas de Análisis de Código Estático

H. Xu et al [HuiXu2015] nos plantea cómo las aplicaciones de la Play Store, efectivamente tienen malas prácticas respecto a la seguridad de las personas tales como almacenar copias de información de los usuarios sin autorización de estos y sin notificarlos, tales como contactos o historial de llamadas. Lo anterior nos da una base para suponer que las aplicaciones que se encuentran en el Marketplace de Android no son del todo seguras, por

[17] https://source.android.com/docs/core/runtime/dalvik-bytecode#instructions

lo que plantea un sistema donde los datos son marcados según el nivel de privacidad que deberían tener y filtrados por el framework desarrollado para evitar que salgan del teléfono sin autorización del usuario. En este mismo año Qi Li et al. [Qili 2015] nos planteó un acercamiento a través de representar como un árbol las características del código y compararlo con aquellos árboles en donde hay presente malwares.

Posteriormente, Wenyun Dai et al [WenyunDai2017] nos planteó un framework similar al plante por Hui Xu, donde los usuarios son capaces de administrar el acceso de las aplicaciones de terceros a los datos del usuario con el fin de limitar a través de por ejemplo marcar imágenes de la galería como de baja privacidad o de alta privacidad.

De forma más reciente han aparecido más documentos respecto a la utilización de técnicas de machine learning e inteligencia artificial para el reconocimiento de patrones sospechosos como lo presenta Yu Zhang et al [YuZhang2020] quien presenta un modelo que a través de la extracción de características semánticas del código se entrena un modelo de aprendizaje que aprende a detectar malwares por sí solo, por lo que al ponerlo a prueba en más aplicaciones el rendimiento debería ir mejorando. Por otro lado, Hsu et al [HsuMyatWin2022] nos presentan un técnica en donde el código es representado como un grafo según los recursos utilizados y como este grafo puede ser separado en los diversos que este puede seguir para ser analizados de manera independiente e identificar si alguno de estos caminos podría presentar alguna vulnerabilidad o comportamiento sospechoso.

Finalmente otra técnica que llama la atención es el plateado por Kritiina Rahkema et al [KristiinaRahkema2021] en donde el análisis no se realiza en sí al código, sino al análisis del control de versiones del código con el fin de identificar cambios importantes que hayan sido capaces de originar una vulnerabilidad y comprender el cómo ha evolucionado el sistema, como también la planteada por Umme Mannan et al [UmmeMannan2016] quien nos presenta una técnica que es capaz de detectar "*Code Smells"* a través del análisis del código fuente de las aplicaciones.

2.10.2 Categorización de técnicas de la literatura

Con el fin de facilitar el análisis de los documentos presentados se decidió categorizar las técnicas estudiadas en la literatura según el tipo de análisis estático que éstas realicen. Las categorías planteadas fueron:

- **Análisis Sintáctico**: Este tipo de técnicas se enfocan en el análisis de la estructura sintáctica del código fuente, es decir encontrar errores de sintaxis, *typos*,

problemas al manejar formatos u otros errores que pueden presentarse al escribir el código. Este tipo de técnicas nos servirán para encontrar malas prácticas o vulnerabilidades en el código.

- **Análisis de Flujo de Datos**: Este tipo de técnicas analizan cómo los datos son enviados entre distintas secciones del código, es decir a través de paso de funciones, obtenciones o envíos a aplicaciones de terceros. Este tipo de técnicas nos servirá para lograr encontrar posibles fugas de información o comportamiento similar a un malware en las aplicaciones.
- **Análisis de Reglas de Codificación**: Este tipo de técnicas tiene como finalidad identificar que el código cumpla estándares de calidad, esto con el fin de identificar posibles malas prácticas en el código como puede ser la duplicación de código, uso inapropiado de variables, complejidad innecesaria u otros.
- **Análisis de Permisos**: Este tipo de técnicas analizará el código fuente de las aplicaciones en conjunto con el archivo *Manifest* que viene dentro del apk, con el objetivo de identificar si los permisos utilizados por la aplicación son los mismos que son declarados y si estos corresponden a permisos que el tipo de aplicación requiere. Este tipo de técnicas pueden ser de utilidad para identificar comportamientos similares a malware o posibles vulnerabilidades en las aplicaciones.

Luego de definir las categorías en las cuales serán agrupadas las técnicas, se comenzó a asignar una categoría a cada una de ellas.

- **Análisis sintáctico**:
    - [KristiinaRahkema2021] provee una técnica para identificar la aparición de code smells a través del análisis de los cambios que sufre el código a través del código y el control de versiones.

    - [YuZhang2020] propone una metodología para procesar las APKs con el fin de generar un grafo que represente las características semánticas de las aplicaciones el cual puede ser analizado mediante redes neuronales de inteligencia artificial con el fin de buscar posibles código maliciosos dentro de la aplicación.

- **Análisis de flujo de dato**s:
    - [WenYunDai2017] nos plantea DASS, un sistema que nos permite detectar filtraciones de datos y violaciones de seguridad a través de otorgar al usuario la opción de filtrar los datos que son entregados a las aplicaciones.
    - [GokhanKul2018] propone un enfoque para detectar fugas de información a través de identificar secciones de código que realicen operaciones de lectura y escritura en bases de datos, y compara el número de operaciones

que se realizan a través del tiempo con el fin de identificar posibles cambios significativos en el comportamiento de las personas.

- [HsuMyatWin2022] plantea que debido a la naturaleza de Android donde las funcionalidades son llamadas a través de eventos es que es posible modelar cada uno de los llamados como nodos de un grafo, lo cual nos permitirá dividir este grafo en todos los caminos posibles que puede tomar la aplicación y por lo tanto realizar análisis de cada uno de estos en búsquedas de vulnerabilidades.

- **Análisis de reglas de codificación**:
  - [QiLi2015] nos plantea el análisis a partir del *Characteristic Tree* generado a partir del código fuente, los pasos para generar este árbol es recolectar clases y métodos, posteriormente se extraen llamados a funciones, permisos y otras características que podrían ser de interés.
  - [AsafShabtai2010] propone la idea de que si su algoritmo de machine learning es capaz de analizar el código fuente de una aplicación y detectar si es un videojuego o una herramienta, entonces debería poder ser extrapolado a analizar malware diferenciando aplicaciones seguras de maliciosas.
  - [UmmeMannan2016] en este artículo los autores desarrollan un catálogo de *code smells* que serán buscados a través del análisis de código estático. Específicamente una técnica llamada DROIDSCOUT cuya primera fase es obtener información relevante de clases y métodos, como la complejidad ciclomática, acoplamiento y cohesión, la segunda fase consiste en analizar sin la información obtenida se encuentra dentro de las métricas de lo que se podría considerar una mala práctica.
- **Análisis de permisos**:
  - [HuiXu2015] presenta una herramienta denominada SpyAware, enfocada principalmente en detectar patrones de comportamiento de las aplicaciones que puedan indicar accesos a permisos o acciones no autorizadas por el usuario como lo son el almacenamiento de información personal, ubicación del dispositivo, contactos u otros.

2.10.3 Ofuscación de Código

Con respecto a la ofuscación de código también se ha estudiado el cómo se podría abordar esta problemática y cuáles son los últimos avances que han existido tanto en como se ofuscan las aplicaciones como en el enfoque para desofuscarlas.

Al realizar el estudio primero analizamos el presentado por Hui Xu et al [HuiXu2017] quien nos presenta una encuesta acerca de los programas de ofuscación más utilizados, más seguros y qué métodos existen para llevar a cabo esta.

Posterior a esto nos enfocamos en la búsqueda respecto a que técnicas estaban siendo más utilizadas actualmente donde encontramos a Dharmalingam et al [Dharmalingam2022] quien nos propone una herramienta para llevar a cabo la ofuscación llamada FineObfuscatos, que utilizando técnicas que modificando el flujo de control y de datos sea capaz de encriptar el código enfocándose en afectar lo menos posible el costo computacional que la ofuscación trae consigo. Esta técnica al ser comparada con otras demostró una mejora significativa contra la ingeniería inversa y un mejor rendimiento de la aplicación al ser comparado con otras herramientas de ofuscación.

Finalmente se encontró el documento presentado por Geunha et al [Geunha2022] que planteaba una herramienta desarrollada Deoptfuscatos que utiliza técnicas de análisis de código estático y dinámico para desofuscar el flujo de control de la aplicación.

2.10.4 Herramientas utilizadas en la literatura

Herramienta	Descripción	Archivo Entrada	Archivo Salida	Referencia
JADX	Herramienta de decompilación de Java.	Archivos .jar, .class y .apk	Archivos en Java	[Arnatovich2018]
Procyon	Herramienta de decompilación de Java.	Archivos .jar y .class.	Archivos en Java	[Geunha2022]
FernFlower	Herramienta de decompilación de Java.	Archivos .jar y .class.	Archivos en Java	[Geunha2022]
Enjarify	Herramienta de decompilación de Python a Java	Archivo .py	Archivos en Java	[HuiXu2017]
APKTool	Herramienta de ingeniería inversa para	Archivos .apk	Archivos en código Smali	[Dharmalingam2022]

	archivos .apk			
Dex2jar	Conversión de archivos .dex de Dalvik a archivos .jar de Java	Archivos .dex	Archivos en Java	[Dharmalingam2022]
JD-GUI	Visor de archivos .class y .jar con funciones de decompilación	Archivos .jar y .class	Código fuente Java	[HuiXu2017], [Arnatovich2018]

Tabla 4: Herramientas utilizadas en la literatura

Fuente: Elaboración propia.

2.10.5 Pipeline de trabajo utilizado en la literatura

A partir de este análisis de la literatura, se puede identificar un pipeline genérico, con pequeñas variantes en casos particulares, pero aborda la mayoría de los documentos presentados.

Figura 5: Pipeline genérico.

Fuente: Elaboración propia.

CAPÍTULO 3: PROPUESTA DE SOLUCIÓN

3.1 Pipeline de trabajo

Con el fin de realizar el análisis estático de diversas aplicaciones es necesario que se defina una forma de trabajo que pueda ser utilizada independiente de la aplicación y sus características, como también para abarcar dos áreas de interés como lo son: la capacidad de replicar el experimento por parte de otros investigadores, lo que podría aportar en la validación de los resultados y el ser capaces de definir en qué fase del proceso serán utilizadas las herramientas, definiendo también el problema que atacan.

A continuación se presenta el pipeline de trabajo que será utilizado con el fin de comparar las diversas técnicas de análisis estático seleccionadas.

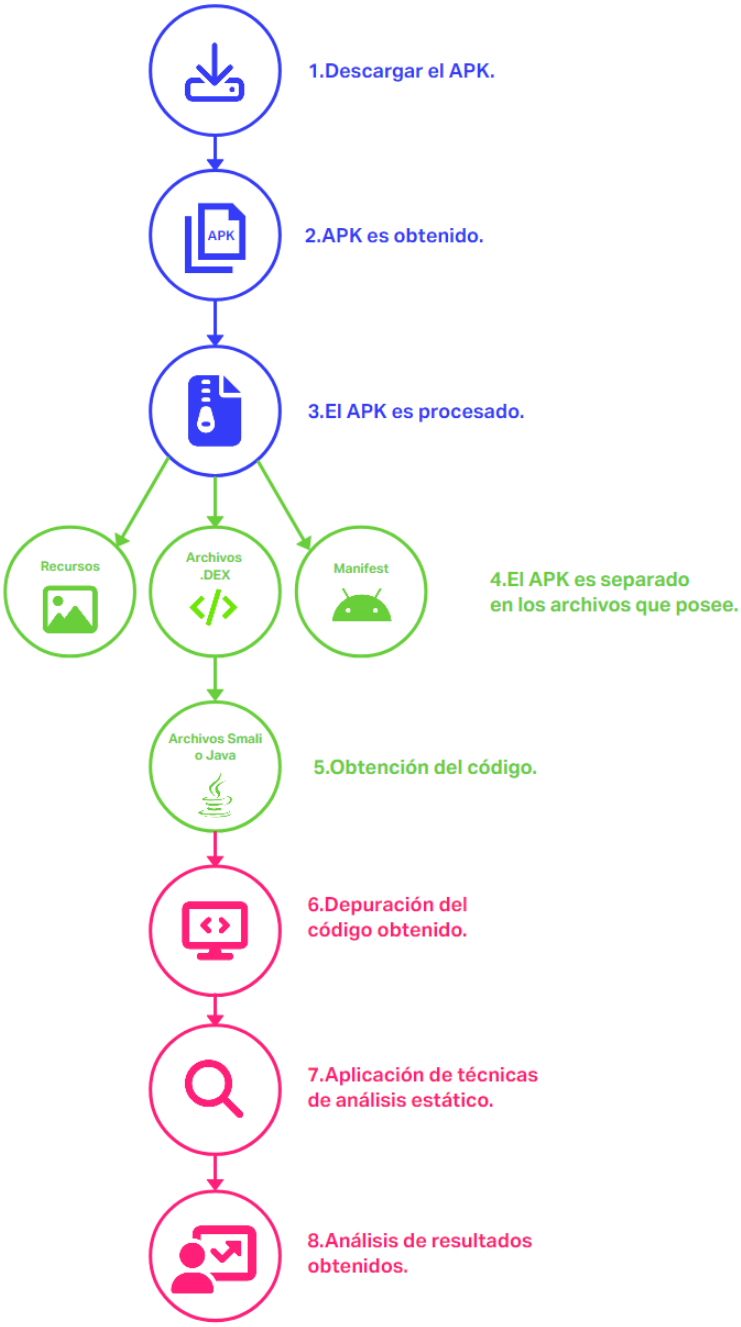


Figura 6: Pipeline de trabajo

Fuente: Elaboración propia

A partir del pipeline planteado, se explicarán más en detalle los pasos a seguir, las problemáticas que se pueden presentar en determinados casos y el cómo estas problemáticas serán atacadas en las distintas fases, cabe recalcar que hay diferencias con el pipeline genérico dado que consideramos la obtención automática de APKs a partir de los *packagenames*, como también la depuración de la aplicación en caso de estar ofuscada u obtener solo los archivos necesarios en el caso de trabajar con código Smali.

3.2 Proceso de análisis de código estático

En el pipeline se pueden identificar distintas fases por colores siendo las de color azul aquellas encargadas de la obtención del APK, las de color verde las encargadas de la obtención del código a analizar y las de color rosado la fase de estudio del código, donde será realizado el estudio comparativo de técnicas de análisis de código estático.

1. Descargar el APK: Consiste en obtener el APK a partir de la Play Store de Google o a partir de una tienda de aplicaciones de terceros, para lo cual se desarrollará un programa en Python utilizando beautifulsoup. Este programa se encargará de descargarlas a partir del nombre del paquete y su versión.
2. Obtención del APK: Este paso es más que nada verificar que el APK fue descargado sin problemas, con el fin de que éste sea procesado posteriormente.
3. Procesamiento del APK: Este momento es en el que transformamos el archivo con extensión APK a alguno que pueda ser descomprimido como RAR, ZIP o alguno similar. Esto nos permitirá obtener los archivos que nombraremos en el siguiente paso.
4. Obtención de archivos que conforman el APK: Luego de descomprimir el APK debemos obtener tres tipos de archivos distintos iniciando por el Manifest.xml siendo el archivo donde son declarados los permisos utilizados por la aplicación, también se obtendrán los recursos de la aplicación como imágenes, audio, etc. Finalmente se obtendrán los archivos con extensión DEX que son aquellos a los cuales se les aplica ingeniería inversa con el fin de obtener el código fuente o similares.
5. Obtención del código: Esta fase es en la cual obtendremos el código fuente dependiendo de la técnica que utilizaremos puede ser código en Java o en Smali, para lo cual utilizaremos herramientas como Dex2Jar junto a JD-Gui en el caso de Java o APKTool en el caso de querer obtener código Smali.
6. Depuración del código obtenido: Una de las fases que es considerada importante es la limpieza del código que se obtuvo a partir de las herramientas utilizadas, dado que aquí es donde se nos puede presentar el problema del código ofuscado. Inicialmente la selección de archivos a analizar será de manera manual, pero se ha visto que existe la posibilidad de filtrar a través de expresiones regulares en el caso de código Smali.

7. Análisis de código estático: Esta fase consiste en aplicar las técnicas seleccionadas al código obtenido, con el fin de obtener métricas respecto a la seguridad de la aplicación, las cuales son vulnerabilidades y malas prácticas encontradas, categoría de la aplicación, tiempo de ejecución y similitud de la aplicación con malware u otras aplicaciones de interés.
8. Análisis de resultados: A partir de las métricas obtenidas aqui se realizará el estudio comparativo desde distintos puntos como puede ser como afecta la categoría de la aplicación, como influye la utilización de Smali o Java, que técnicas dan más información o que tipo de información van y el analizar que tan complementarias pueden ser las distintas técnicas.

3.3 Selección de técnicas a analizar

3.3.1 Proceso de selección de técnicas y técnicas seleccionadas

Con el fin de seleccionar las técnicas previamente descritas y categorizadas en el punto 2.10.2 fue necesario definir parámetros con el fin de explicar nuestras elecciones al momento de incluir o excluir técnicas del estudio.

Los parámetros definidos para la selección de técnicas están:

- **Nivel de complejidad de la técnica**: Capacidad propia con la que podemos replicar la técnica seleccionada y manejo del dominio de esta.
- **Flexibilidad y escalabilidad de la técnica**: Qué tantas aplicaciones podemos abarcar con la técnica en este momento y cuántas podrían ser analizadas a futuro.
- **Capacidad de detección**: A partir de los resultados de la literatura, analizar qué tan eficiente es la técnica en términos de detección de vulnerabilidades.
- **Tipo de falla que es capaz de detectar**: Analizar si la técnica es capaz de detectar malware, vulnerabilidades y malas prácticas al mismo tiempo o si solo es capaz de detectar tipos específicos.
- **Categoría de la técnicas**: Es necesario seleccionar técnicas de más de una de las categorías planteadas anteriormente, con el fin de poder abarcar diversos tipos de análisis estáticos y poder realizar un análisis comparativo entre categorías.
- **Compatibilidad de las técnicas**: Al mismo tiempo al seleccionar distintas categorías de técnicas, es interesante analizar qué tan complementarias son estas al momento de dar información.
- **Modernidad de la técnica**: Qué tan aplicable es la técnica en aplicaciones modernas de Android.

3.3.2 Técnicas seleccionadas

A partir de las métricas planteadas anteriormente, se realizó la selección de las siguientes

técnicas a implementar:

- **Análisis sintáctico del código:** Corresponde a analizar el código como si este fuera texto plano e identificar posibles malas prácticas, vulnerabilidades o malware mediante el uso de expresiones que podrían ser identificadas en cada línea del código. Fue seleccionada dado que puede ser aplicada a cualquier aplicación a la cual podamos acceder a su código fuente, es capaz de detectar las fallas que definamos mediante expresiones regulares y es una técnica sencilla de implementar.
- **Análisis mediante grafo de características:** Se desea implementar la técnica planteada por [QiLi2015] donde se genera un grafo de características del APK mediante distintas capas, pero con la modificación de que en lugar de identificar a qué tipo de malware corresponde mediante un porcentaje de similitud, vamos a calcular la similitud entre aplicaciones que pueden ser obtenidas en la Play Store con APK's donde sepamos que ya existe malware. Fue seleccionada dado que debería ser capaz de identificar posible comportamiento sospechoso similar a un malware, identificar clases con comportamiento sospechoso que podrían ser posteriormente analizadas por análisis sintáctico, respecto a la capacidad de detección debería ser capaz de darnos una idea más global del comportamiento de la aplicación más que vulnerabilidades específicas y debería ser una técnica sencilla de implementar, dado que la mayor complejidad es la generación de los grafos.
- **Análisis de permisos:** Esta técnica fue seleccionada debido a que la información que otorga varía mucho a las técnicas nombradas anteriormente, dado que en lugar de analizar las clases en sí del código se encargará de analizar el AndroidManifest.xml de la aplicación, lo que nos dará información que posteriormente podría ser comparada al código fuente en sí y al mismo tiempo nos dará información respecto a los distintos permisos utilizados por la aplicación y los riesgos a los que se arriesga el usuario al otorgarlos. Esta técnica fue seleccionada por la información que nos otorga, también por el hecho de que todas las aplicaciones deben tener AndroidManifest.xml por lo que es aplicable al 100% de las APK's y que será una técnica relativamente sencilla de implementar, dado que su dificultad se encuentra en elaborar un archivo que asocie permisos con niveles de riesgo a los que se exponen los usuarios.
- **Análisis de código mediante inteligencias artificiales de procesamiento de texto:** A pesar de que esta no corresponde a una técnica que aparezca en la literatura, surge como una idea emergente al ver la potencia de inteligencias artificiales como ChatGPT, por lo que surgió la intención de implementar una técnica que sea capaz de obtener código fuente de las aplicaciones y pasarlo a GPT mediante prompting para estudiar la capacidad que esta herramienta puede otorgar al análisis de código estático. Fue seleccionada principalmente por la utilización de una herramienta

que está en la vanguardia tecnológica hoy en dia y que se ve como una tecnología que podría volver más eficiente el análisis de código estático, dado que debería ser capaz de volverlo independiente de los conocimientos que el investigador tenga del lenguaje de programación a analizar y le ahorrará el tiempo de entrenar inteligencias artificiales por su cuenta que sean capaz de detectar vulnerabilidades o malas prácticas.

3.4 Selección de tecnologías

3.4.1 Herramientas utilizadas para la obtención de código

Dentro del contexto del análisis de código estático, uno de los pasos más importantes que debemos decidir es con qué tecnologías obtendremos el código fuente a partir del apk, es por esto que decidimos experimentar con distintas herramientas que son comúnmente utilizadas en la literatura y analizar la facilidad de uso de estas, la facilidad de instalación y la calidad del código que nos es otorgado.

La primera herramienta que se analizó es APKTool, la cual requiere ser instalada desde su página web[18] e instalar Java en el equipo. Una vez instalada es relativamente fácil de utilizar a través de la línea de comandos, a través de comandos como "*apktool d packageName.apk*".

[18] https://ibotpeaches.github.io/Apktool/

```
C:\Users\Lenovo\Documents\USM\Memoria\Experimentos\Experimento apktool>apktool d ../apks/tiktok.apk
I: Using Apktool 2.6.1 on tiktok.apk
I: Loading resource table...
I: Decoding Shared Library (com.zhiliaoapp.musically.df_live_cast), pkgId: 113
I: Decoding Shared Library (com.zhiliaoapp.musically.df_live_cast), pkgId: 113
I: Decoding Shared Library (com.zhiliaoapp.musically.df_trending), pkgId: 114
I: Decoding Shared Library (com.zhiliaoapp.musically.df_live_cast), pkgId: 113
I: Decoding Shared Library (com.zhiliaoapp.musically.df_trending), pkgId: 114
I: Decoding Shared Library (com.zhiliaoapp.musically.df_ship), pkgId: 115
I: Decoding Shared Library (com.zhiliaoapp.musically.df_live_cast), pkgId: 113
I: Decoding Shared Library (com.zhiliaoapp.musically.df_trending), pkgId: 114
I: Decoding Shared Library (com.zhiliaoapp.musically.df_ship), pkgId: 115
I: Decoding Shared Library (com.zhiliaoapp.musically.df_oppo_push), pkgId: 117
I: Decoding Shared Library (com.zhiliaoapp.musically.df_live_cast), pkgId: 113
I: Decoding Shared Library (com.zhiliaoapp.musically.df_trending), pkgId: 114
I: Decoding Shared Library (com.zhiliaoapp.musically.df_ship), pkgId: 115
I: Decoding Shared Library (com.zhiliaoapp.musically.df_oppo_push), pkgId: 117
I: Decoding Shared Library (com.zhiliaoapp.musically.df_music_dsp), pkgId: 120
I: Decoding Shared Library (com.zhiliaoapp.musically.df_live_cast), pkgId: 113
I: Decoding Shared Library (com.zhiliaoapp.musically.df_trending), pkgId: 114
I: Decoding Shared Library (com.zhiliaoapp.musically.df_ship), pkgId: 115
I: Decoding Shared Library (com.zhiliaoapp.musically.df_oppo_push), pkgId: 117
I: Decoding Shared Library (com.zhiliaoapp.musically.df_music_dsp), pkgId: 120
I: Decoding Shared Library (com.zhiliaoapp.musically.df_mgl_h5), pkgId: 126
I: Decoding Shared Library (com.zhiliaoapp.musically.df_live_cast), pkgId: 113
I: Decoding Shared Library (com.zhiliaoapp.musically.df_trending), pkgId: 114
I: Decoding Shared Library (com.zhiliaoapp.musically.df_ship), pkgId: 115
I: Decoding Shared Library (com.zhiliaoapp.musically.df_oppo_push), pkgId: 117
I: Decoding Shared Library (com.zhiliaoapp.musically.df_music_dsp), pkgId: 120
I: Decoding Shared Library (com.zhiliaoapp.musically.df_mgl_h5), pkgId: 126
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: C:\Users\Lenovo\AppData\Local\apktool\framework\1.apk
W: Could not decode attr value, using undecoded value instead: ns=android, name=resource, value=0x71020000
I: Regular manifest package...
I: Decoding file-resources...
W: Cant find 9patch chunk in file: "o/gl.9.png". Renaming it to *.png.
W: Cant find 9patch chunk in file: "o/gl.9.png". Renaming it to *.png.
W: Cant find 9patch chunk in file: "o/gl.9.png". Renaming it to *.png.
W: Cant find 9patch chunk in file: "o/gl.9.png". Renaming it to *.png.
W: Cant find 9patch chunk in file: "o/gl.9.png". Renaming it to *.png.
W: Cant find 9patch chunk in file: "o/gl.9.png". Renaming it to *.png.
W: Cant find 9patch chunk in file: "o/gl.9.png". Renaming it to *.png.
W: Cant find 9patch chunk in file: "o/gl.9.png". Renaming it to *.png.
W: Cant find 9patch chunk in file: "o/gl.9.png". Renaming it to *.png.
W: Cant find 9patch chunk in file: "o/gl.9.png". Renaming it to *.png.
W: Cant find 9patch chunk in file: "o/gl.9.png". Renaming it to *.png.
W: Cant find 9patch chunk in file: "o/gl.9.png". Renaming it to *.png.
```

Figura 7: Utilización de apktool

Fuente: Elaboración propia

Como se puede ver en la figura, la ejecución es relativamente simple y el código obtenido está en el lenguaje de programación de Smali, por lo que la mayor desventaja que esta herramienta posee es la falta de documentación que el lenguaje utilizado posee, pero al mismo tiempo, que sea código de bajo nivel, es una ventaja que permite un análisis estático que podría ser más beneficioso para nosotros en la búsqueda de vulnerabilidad, malas prácticas o malware.

La segunda herramienta que analizamos en función de la literatura fue Dex2Jar en

conjunto con JD-Gui. Dex2Jar, en específico, nos permite obtener código Java a partir del apk de la aplicación, por lo que la utilización de este lenguaje podría ser beneficioso para un análisis estático, debido a la alta documentación que éste posee, el problema es que el código obtenido con esta técnica, no es el código fuente de la aplicación, sino que un aproximado, por lo que la compilación del código obtenido no será funcional y realmente no estaríamos analizando el código en sí de la aplicación.

Para instalar Dex2Jar, los pasos utilizados fueron los siguientes:

1. Descarga Dex2Jar desde su sitio web oficial[19] y descomprimir el ZIP descargado.
2. Descarga el archivo APK que deseas analizar.
3. Cambiar la extensión del archivo APK a ZIP o RAR para que pueda ser descomprimido en el Manifest, archivos DEX y archivos como imágenes u otros.
4. Ubicar en la misma carpeta Dex2Jar con los archivos DEX.
5. Ejecuta los comandos de Dex2Jar usando el siguiente comando en la línea de comandos:"d2j-dex2jar.bat nombre_del_archivo.dex", lo cual generará un archivo JAR.
6. Para visualizar el contenido del archivo JAR, es necesario instalar JD-GUI.
7. Descarga JD-GUI desde su repositorio[20] y obtén el archivo JAR correspondiente.
8. Abre el archivo JAR de JD-GUI con doble clic o ejecutar "java -jar jd-gui-x.x.x.jar" en la línea de comandos, reemplazando "jd-gui-x.x.x.jar" con el nombre del archivo JD-GUI que descargaste.
9. Ahora podrás visualizar el contenido del archivo JAR generado previamente por Dex2Jar.

Finalmente una de las herramientas que genera más interés es AndroGuard, dado que es posible utilizarla mediante su librería de python a la cual se le puede entregar el APK de una aplicación como parámetro, para que la librería nos devuelva diversa información de esta como lo es: el código Smali, clases de la aplicación, instrucciones por clase, nombre del paquete y otros.

Herramienta	**Entrada**	**Utilización**	**Salida**	**Visualización del código**
APKTool	APK	Consola	Código Smali	Editor de texto

[19] https://sourceforge.net/projects/dex2jar/
[20] https://java-decompiler.github.io/

AndroGuard	APK	Librería Python	Código Smali	-
Dex2Jar	DEX	Consola	Código Java	JD-GUI
JD-Gui	-	Archivo JAR	-	-

Tabla 5: Análisis de posibles herramientas a utilizar

Fuente: Elaboración propia

3.4.2 Lenguajes de programación

A partir de las herramientas planteadas anteriormente, se toma la decisión de que los lenguajes de programación que serán utilizados para analizar el código serán Java y Smali. Lo anterior, debido principalmente a que son los lenguajes entregados por estas herramientas y la diferencias que estos presentan al ser obtenidos por ingeniería reversa del APK, siendo las principales que el código en Java obtenido a través de Dex2Jar es código aproximado al real por lo que Smali podría otorgarnos información faltante en este lenguaje o información que, en un nivel más alto, se encuentra ofuscada. Por otro lado, Java es un lenguaje más sencillo de entender debido a que se encuentra en un nivel más alto que Smali, de manera que, tanto el análisis sintáctico de esto así como el análisis a través del flujo de datos, podría resultar más sencillo en este lenguaje. De todas maneras, cabe recalcar que el análisis de una aplicación no será a través de un lenguaje u otro, sino que serán utilizados de manera complementaria, en la medida de lo posible.

En relación con lo anterior, las técnicas a implementar serán desarrolladas inicialmente utilizando Python 3.10 debido a la estabilidad que tiene esta versión y al dominio que el ejecutor de este proyecto de tesis, posee sobre este lenguaje en función a experiencia previa programando en él. Otro de los puntos fuertes que posee Python, es que posee un gran número de librerías útiles tales como: Pandas, numpy y matplotlib, que podrían sernos de utilidad al momento de querer realizar el análisis de datos para comparar el rendimiento de las distintas técnicas.

3.4.3 Otras tecnologías a utilizar

Dado que las técnicas en sí tienen comportamientos distintas hay otras tecnologías y técnicas que serán utilizadas y se verá que tan factible es su aplicabilidad a las aplicaciones, entre estas se encuentran:

- **RegEx:** La utilización de expresiones regulares nos podría ser beneficiosa en el filtrado de la información otorgada por herramientas como lo son APKTool o

Dex2Jar, dado que estas otorgan código que no es necesariamente clases y métodos desarrollados por la aplicación en sí, sino que a código base de todas las aplicaciones o de librerías importadas por la aplicación.

- **FastAPI:** Corresponde a un framework de Python que busca facilitar el desarrollo de APIs REST, lo que nos podría ayudar en el traspaso de información entre distintas técnicas y a sistemas externos en el caso de querer disponibilizar los resultados de los análisis.

Con el objetivo de seleccionar la base de datos, lo primero que se hizo fue comparar los beneficios que nos otorga una base de datos SQL en comparación a una NoSQL.

Aspecto	Base de Datos SQL	Base de Datos NoSQL
Esquema de datos	Estructuras fijas con tablas y columnas	Esquema flexible, adaptado a formato JSON.
Complejidad de las consultas	Permite unir múltiples tablas y realizar consultas complejas.	No requiere unir tablas ni la utilización de consultas complejas.
Rendimiento	En el caso de consultas complejas pueden tenerse resultados lentos.	Tienen un rendimiento superior a las DB SQL, dado que están adaptadas a trabajar con archivos JSON.
Utilización de formato JSON	Es necesario *parsear* la información de las tablas a formato JSON.	Directamente en formato JSON.
Adaptabilidad	Modificar los esquemas requiere la modificación de las tablas y columnas.	Permite una mayor flexibilidad en los cambios de la estructura.

Tabla 6: Comparación entre BD SQL y BD NoSQL

Fuente: Elaboración propia

A partir de este cuadro comparativo, es que se decidió seleccionar la utilización de una base de datos **NoSQL** por distintos motivos, entre los que están:

- Adaptabilidad que provee al requerir cambios en el formato de entrada o salida.
- Facilidad para trabajar con archivos en formato JSON, que es el formato en el que se desea entregar los archivos de salida y recibir las consultas de entrada con información como el *packageName.*
- El tener un rendimiento mayor, permitirá realizar análisis donde las tecnologías tengan un menor impacto en los resultados obtenidos.

3.4.4 Resumen de tecnologías seleccionadas

A continuación se presenta un resumen de las herramientas, lenguajes y tecnologías seleccionadas como también el problema que abordan o el beneficio que nos podrían entregar.

Nombre	Versión	Tipo	Utilidad
Python	3.10	Lenguaje de programación	Desarrollo de las técnicas, API y análisis de datos.
Smali	2.0	Lenguaje de programación	Código a analizar.
Androguard	3.4.0	Librería de Python	Librería que permite obtener el código Smali de una aplicación mediante el uso implícito de APKTool.
APKTool	2.4.1	Herramienta	Obtención de código Smali.
MongoDB	4.1.1	Base de Datos noSQL	Almacenamiento de reportes de aplicaciones, como también de grafos serializados en formato JSON.
FastAPI	0.78	Framework de Python	Desarrollo de API en Python.
Pandas	1.5.3	Librería de Python	Análisis de datos en el estudio comparativo.
MatPlotLib	3.7.1	Librería de Python	Graficar datos en el

			estudio comparativo.
Docker	24.0.5	Manejo de Contenedores	Utilizado principalmente para la creación del contenedor con la API que será disponibilizada.
Docker Compose	2.20.2	Orquestador de Contenedores	Utilizado para orquestar los contenedores de manera local en el ambiente de desarrollo.
Visual Studio Code	1.82.3	Interfaz de desarrollo	IDE utilizada para desarrollar las técnicas.

Tabla 7: Técnicas, tecnologías y herramientas a utilizar.

Fuente: Elaboración propia.

3.5 Consideraciones sobre: Mala práctica, Vulnerabilidad y Malware.

Como se nombró en la sección 2.6, los conceptos de malas prácticas, vulnerabilidad y malware son distintas entre sí, por lo que a continuación se nombraran diversos ejemplos de éstas y finalmente se seleccionara un subgrupo de estas para analizar dentro del código fuente de las aplicaciones.

3.5.1 Malas prácticas y vulnerabilidades

Con el objetivo de realizar este listado, se consideraron dos fuentes de datos que son las utilizadas en la literatura. En primer lugar se utilizará el "*Common Weakness Enumeration*" del Mitre[21] y la segunda fuente que se utilizará es la fundación OWASP, cuyo propósito es mejorar la seguridad del software a través de proyectos por lo que tienen publicadas a lo largo de su página web[22] diversas fuentes desde las que se pueden obtener las malas prácticas y vulnerabilidades que consideran al analizar.

En cuanto al Mitre, estos poseen un Top 25 de malas prácticas más comunes del año 2022 que pueden dar origen a vulnerabilidades en el código. Estas son:

[21] https://cwe.mitre.org/data/definitions/699.html
[22] https://owasp.org/

1. **CWE-787 Escritura fuera de límites:** Ocurre cuando se escriben datos más allá de lo que permite el búfer. Un ejemplo de esto es escribir sin verificar que el espacio de memoria sea válido por lo que ocurre una corrupción de los datos en el caso de que no se tenga el espacio suficiente.
2. **CWE-79 Neutralización inadecuada de la entrada durante la generación de páginas web**: Ocurre cuando el sistema no neutraliza los datos ingresados por el usuario de manera correcta por lo que puede dar origen a inyecciones de código.
3. **CWE-89 Neutralización inadecuada de elementos especiales utilizados en un comando SQL o inyecciones SQL:** El sistema no limpia los datos ingresados por un usuario verificado de que no sean código SQL, lo que permite que existan posibles inyecciones de código a través de formularios del sistema.
4. **CWE-20 Validación inadecuada de la entrada:** Corresponde cuando el sistema no verifica de manera correcta la entrada y da origen a otras vulnerabilidades comunes como las neutralizaciones inadecuadas o escrituras fueras de límite.
5. **CWE-125 Lectura fuera de límites:** Ocurre cuando una aplicación o una sección de código es capaz de acceder a áreas de memoria fuera de sus límites de lectura.
6. **CWE-78 Neutralización inadecuada de elementos especiales utilizados en un comando del sistema operativo:** Cuando un llamado a alguna función del sistema operativo requiere información ingresada por el usuario lo que puede dar como origen a que el usuario pueda ingresar comandos del sistema operativo dentro de campos de formularios por ejemplo.
7. **CWE-416 Uso después de liberar:** Referenciar a espacios de memoria que son liberados lo que puede causar que se ejecute código, se obtengan valores inesperados o que el sistema deje de funcionar.
8. **CWE-22 Limitación inadecuada de una ruta a un directorio restringido:** Ocurre cuando se quiere limitar el acceso a ciertos recursos de la aplicación, pero las rutas ocupadas por el código no son validadas por lo que son accesibles por secciones de código que no deberían ser capaces de acceder a ellas.
9. **CWE-352 Falsificación de petición en sitios cruzados (CSRF):** El sitio web no verifica o no es capaz de verificar que las solicitudes realizadas por un usuario fueron intencionales por este. Un ejemplo de esto son enlaces maliciosos que al ser clickeados por un usuario pueden ejecutar funcionalidad que el usuario no deseaba dentro de la página web a la que se redirecciona.
10. **CWE-434 Carga no restringida de archivos con tipos peligrosos:** El sistema permite la subida de archivos maliciosos que son procesados automáticamente.
11. **CWE-476 Referencias a punteros nulos:** El sistema hace referencias a espacios de memoria donde se espera que haya valores válidos, pero se encuentra con *NULLS*. Lo que puede causar que el sistema deje de funcionar correctamente o se cierre.
12. **CWE-502 Deserialización de datos no confiables:** El sistema no verifica los datos cuando estos son procesados a formato legible por máquina o estructuras de

objetos, por lo que es posible inyectar instrucciones dentro de grupos de datos que serán ejecutados de manera automática al ser procesados.

13. **CWE-190 Desbordamiento de enteros:** Ocurre cuando las operaciones aritméticas dan resultados más allá de los que pueden ser almacenados por el sistema, lo cual retorna valores erróneos.
14. **CWE-287 Autenticación inadecuada:** Cuando no se poseen mecanismos que validen la autenticación del usuario más allá de nombre de usuario y contraseña.
15. **CWE-798 Uso de credenciales *hardcodeadas:*** Cuando dentro del código se encuentran credenciales privadas como accesos a APIs, sistemas de las nubes u otras por lo que podrían ser accedidas a través de ingeniería reversa.
16. **CWE-862 Falta de autorizaciones:** Como su nombre lo dice es cuando el sistema no pide autorizaciones al usuario en caso de intentar acceder a datos sensibles o realizar operaciones que pueden ser de alto riesgo para la privacidad de los datos.
17. **CWE-77 Neutralización inadecuada de elementos especiales utilizados en un comando:** Cuando un comando es ejecutado a partir de información ingresada por el usuario y no se realiza una verificación de esta, por lo que es posible ingresar instrucciones a través de formularios de información por ejemplo.
18. **CWE-306 Faltas de autenticaciones en funciones críticas:** Similar al CWE-862 y corresponde a la falta de algún tipo de autorización extra por parte del usuario al realizar acciones críticas que podrían poner en riesgo la plataforma o su información confidencial.
19. **CWE-119 Restricciones inadecuadas dentro de los límites de un buffer de memoria:** Ocurre cuando no se imponen limitaciones a las operaciones que pueden ser realizadas en un buffer de memoria por lo que pueden generarse errores como escrituras fueras del buffer o sobrescribir datos adyacentes.
20. **CWE-276 Permisos por defecto incorrectos:** Tal como el nombre lo dice es cuando a un usuario se le otorgan más o menos permisos de los que debería tener, siendo más preocupante el primer caso cuando se le otorgan más.
21. **CWE-918 Server-Side Request Forgery (SSRF):** No verificar que las solicitudes son enviadas de vuelta a la URL correspondiente, lo que puede dar como resultado que el servidor envíe información sensible a enlaces externos a la plataforma.
22. **CWE-362 Ejecución concurrente utilizando recursos compartidos pero con una sincronización inadecuada:** Ocurre cuando múltiples hilos o microservicios acceden al mismo recursos para manipularlo o leerlo, lo cual puede dar como resultado datos inconsistentes.
23. **CWE-400 Utilización descontrolada de recursos:** El sistema no limita los recursos que pueden ser utilizados por una sesión lo que puede causar que el sistema falle por falta de recursos como memoria o disco.
24. **CWE-611 Restricción inadecuada de referencia a XML con referencias externas:** Este tipo de fallas permitiría que al acceder a un XML este puede referenciar a

alguna entidad externa y darle acceso a la lectura de archivos, revelación de datos sensibles o denegaciones de servicio.

25. **CWE-94 Control inadecuado de la generación de código:** El sistema no verifica el código que es ejecutado y permite la inserción de código externo, lo cual puede tener consecuencias como ejecución de comandos del servicios, extracción de datos sensibles o denegaciones de servicio.

Por otro lado una de las secciones que posee el Mitre es la de vulnerabilidades más comunes en el área del desarrollo de software, entre las que podemos encontrar:

- **CWE-561 Código muerto:** Corresponde a código que no es utilizado.
- **CWE-319 Transmisión de información sensible como texto:** Información sensible es pasada en el código como texto plano sin ningún tiempo de codificación, lo que puede ser generar robo de información sensible por parte de terceros.
- **CWE-327 Uso de métodos de criptografía riesgosos:** Utilización de criptografía que realmente no funciona o es muy riesgoso en el caso de ser accedido por terceros.
- **CWE-478 Falta de casos por defecto en condiciones múltiples:** Cuando en condiciones múltiples como operadores switch o condiciones múltiples como *elseif* no se generan casos para valores que no cumplan ninguna condición.
- **CWE-563 Asignaciones de variables no usadas:** Cuando una variable recibe un valor, pero esta no es utilizada.
- **CWE-1041 Uso de código redundante:** Cuando el producto posee duplicación de código a través de funciones, métodos, procedimientos, etc.
- **CWE-640 Mecanismos débiles para la recuperación de contraseñas olvidadas:** El sistema posee mecanismos para recuperar la contraseña cuando este es olvidada, pero los mecanismos son débiles en cuanto a verificaciones de identidad.
- **CWE-602 Forzar la seguridad al lado del usuario:** Mala práctica donde la seguridad del producto es en gran parte responsabilidad del usuario y no una función del sistema.

Como se nombro anteriormente, otra de las fuentes de vulnerabilidades y malas prácticas que será considerada para el estudio es el OWASP, específicamente el Top 10 de riesgos de seguridad en aplicaciones web elaborado el año 2021[23] el cual busca recalcar cuales vulnerabilidades identificadas por el Mitre, son las más comunes en las aplicaciones web:

- **A01 Control de Acceso roto**
- **A02 Fallos en criptografía**
- **A03 Inyecciones**
- **A04 Diseño Inseguro**
- **A05 Configuraciones erróneas de Seguridad**

[23] https://owasp.org/www-project-top-ten/

- **A06 Componentes vulnerables y obsoletos**
- **A07 Fallas en autenticación e identificación**
- **A08 Fallas en la integridad del software y de los datos**
- **A09 Fallas en el registro y monitoreo de Seguridad**
- **A10 Server Side Request Forgery (SSRF)**

La utilidad de este Top se basa en la capacidad que éste tiene para identificar qué tan comunes son ciertas categorías del CWE y, por lo tanto, es una métrica útil al momento de seleccionar las malas prácticas a investigar en el análisis de código estático y, por ende, saber que posibles vulnerabilidades podría tener el sistema.

Finalmente, las vulnerabilidades y malas prácticas seleccionadas al realizar el análisis de código estático, serán algunas del top 25 de la CWE dependiendo de si son analizables mediante el código fuente de la aplicación, en conjunto a algunas de la sección de desarrollo de software del CWE, las cuales serán analizadas mediante expresiones regulares o instrucciones que dan origen a estas como se explica en la sección 4.2.

3.6 Selección de aplicaciones a analizar

Con el fin de seleccionar qué aplicaciones serán analizadas a través de las técnicas a implementar es que se definieron criterios de selección. Estos son:

- **Popularidad o Número de Descargas:** La cantidad de descargas de una aplicación podría ser un factor a considerar, dado que podría impactar el alcance de usuarios a los que les podría ser de utilidad el análisis de una aplicación en específico.
- **Categoría de la aplicación:** El identificar cómo las técnicas se aplican a distintas categorías de aplicaciones como videojuegos, herramientas, redes sociales, mensajería u otras podría ser de interés para analizar la aplicabilidad de estas.
- **Calificación de la aplicación en Play Store:** La calificación de una aplicación en la Play Store podría ser un indicativo importante dado que esta va usualmente de la mano con el rendimiento de la aplicación, lo que nos podría dar indicios de malas prácticas en cuestiones de rendimiento o en posibles vulnerabilidades que podrían existir.
- **Número de actualizaciones durante los últimos 2 años:** El seleccionar aplicaciones que están siendo regularmente actualizadas podría ser un indicativo de que el equipo de desarrollo está abordando de manera activa los posibles problemas de seguridad u optimizaciones.
- **Peso de la aplicación:** El peso de la aplicación posee 2 implicancias, el primero es que aplicaciones como videojuegos ven su peso principalmente afectado por recursos de la aplicación como modelos, audios u otros tipo de archivo por lo que el peso no siempre es indicativo de más código, pero por otro lado lado una aplicación que tenga un peso inusual para la categoría a la que pertenece podría

ser una señal de que el realizar el análisis estático de como resultado la identificación de malas prácticas, vulnerabilidades o directamente comportamiento de malware.

- **Código abierto:** Para la experimentación se seleccionaron inicialmente aplicaciones de código abierto para evitar problemáticas como la ofuscación o encriptación de código, los cuales deberían ser abordados al realizar el análisis de resultados al aplicar el código a un mayor número de aplicaciones.

A partir de estos criterios se seleccionaron las siguientes aplicaciones a analizar, en base a las categorías anteriores, son:

- **Redes Sociales:** Dada la popularidad de este tipo de aplicaciones es que pueden ser encontradas en la mayoría de los dispositivos móviles, por lo que es de utilidad el analizar este tipo de aplicaciones debido al público que podría verse beneficiado de analizarlas.
    - **Signal[24]:** Es una aplicación de mensajería segura y de código abierto que también se puede considerar como una red social.
    - **Mastodon[25]:** Es una red social descentralizada y de código abierto que se basa en el protocolo ActivityPub.
- **Mensajería:** Bajo el mismo criterio de redes sociales, la mayoría de los dispositivos móviles posee aplicaciones de mensajería por lo que también es interesante estudiar la aplicabilidad de las técnicas en este tipo de aplicaciones.
    - **Element[26]**: Es una aplicación de mensajería basada en el protocolo Matrix y de código abierto. Matrix es un protocolo descentralizado y federado para la comunicación en tiempo real.{
    - **Conversations[27]**: Es una aplicación de mensajería instantánea de código abierto que utiliza el protocolo XMPP (Extensible Messaging and Presence Protocol).
- **Juegos móviles:** El principal criterio para analizar juegos de teléfono es que su comportamiento varía considerablemente en comparación al resto de aplicaciones y es usualmente desarrollado en lenguajes como C# u otros.
    - **Flappy Bird[28]**: Es un juego móvil sencillo y adictivo desarrollado en Unity con lenguaje C#.
    - **2048[29]**: Otro juego móvil muy conocido y simple, desarrollado en HTML5 y JavaScript.

[24] https://github.com/signalapp/Signal-Android
[25] https://github.com/mastodon/mastodon-android
[26] https://github.com/vector-im/element-android
[27] https://github.com/iNPUTmice/Conversations
[28] https://github.com/FelgoSDK/FlappyBird
[29] https://github.com/gabrielecirulli/2048

- **Comida rápida:** Esta categoría es principalmente debido a la información sensible que manejan este tipo de aplicaciones tales como tarjetas de crédito/débito, direcciones, números de teléfonos, nombres u otras que podrían ser de interés para atacantes, por lo que analizar la seguridad de estas aplicaciones es de vital importancia.

3.7 Metodología utilizada al experimentar

A partir de las tecnologías y aplicaciones seleccionadas, a continuación se explicará cómo será aplicado el pipeline de trabajo en la fase de experimentación con el fin de desarrollar un sistema que sea aplicable a la mayor cantidad de aplicaciones y que sea capaz de realizar el análisis en la búsqueda de las vulnerabilidades seleccionadas.

Para esto se plantea un flujo de trabajo que será utilizado para la implementación de las distintas técnicas, tal como sigue:

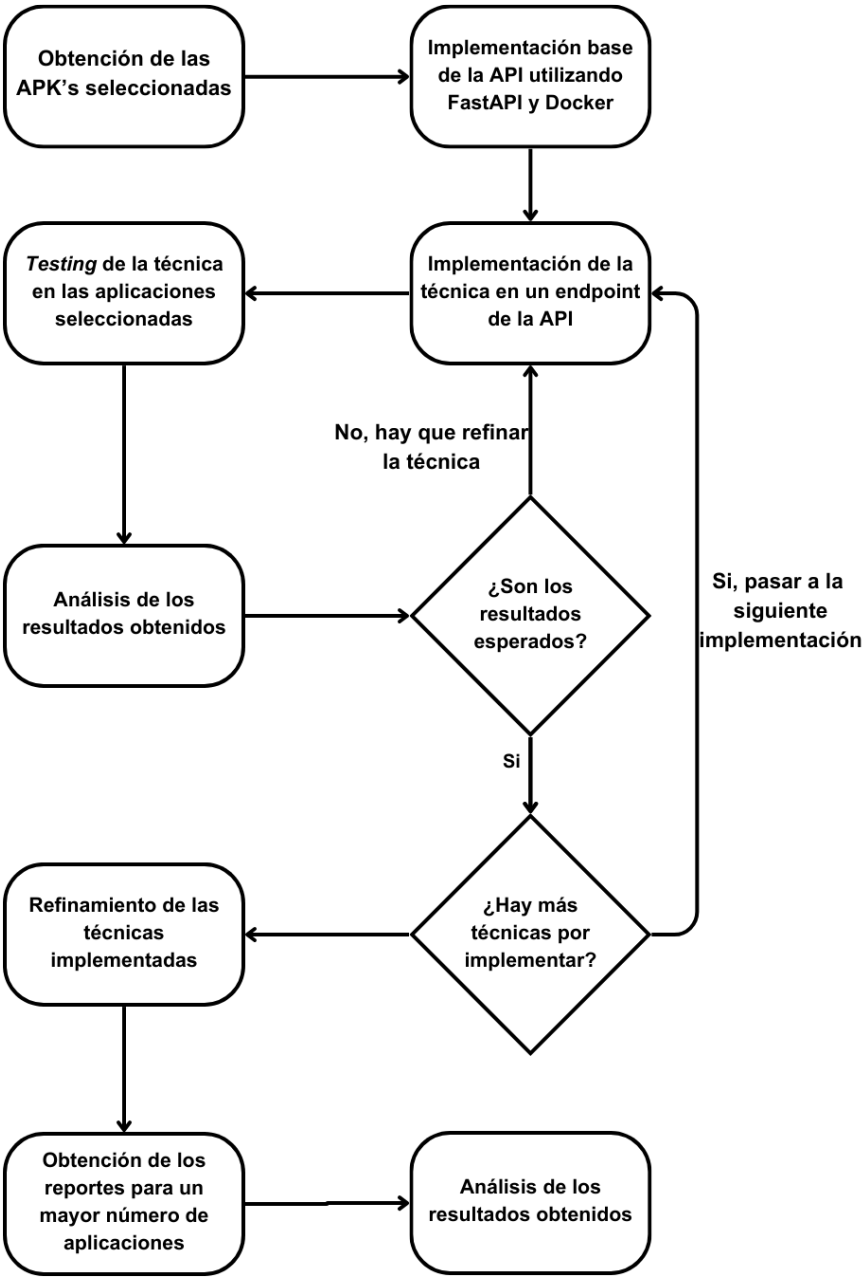


Figura 8: Flujo de trabajo en la experimentación

Fuente: Elaboración propia

3.8 Salida de resultados

Otro de los puntos que son definidos en esta sección es la estructura con la cual contarán los informes, los cuales serán almacenados inicialmente en formato JSON con el fin de poder ser analizados con mayor facilidad por las herramientas de análisis a utilizar y con el fin de poder ser disponibilizados a través de una API a otros sistemas en el caso de ser necesario. Para esto se realizó un análisis de los resultados que se pueden obtener en herramientas que se encuentran en el mercado:

3.8.1 Checkmarx

Checkmarx es una empresa especializada en seguridad de aplicaciones que ofrece soluciones y herramientas de análisis estático de código y pruebas de seguridad dinámica para identificar y mitigar vulnerabilidades en el software. Sus productos ayudan a las organizaciones a evaluar y mejorar la seguridad de sus aplicaciones y a protegerse contra posibles amenazas cibernéticas al detectar y corregir problemas de seguridad en el código fuente y durante la ejecución de las aplicaciones.


```
{
    "files_scanned": 2,
    "files_parsed": 2,
    "files_failed_to_scan": 0,
    "queries_total": 253,
    "queries_failed_to_execute": 0,
    "queries_failed_to_compute_similarity_id": 0,
    "queries": [
        {
            "query_name": "Container Allow Privilege Escalation Is True",
            "query_id": "c878abb4-cca5-4724-92b9-289be68bd47c",
            "severity": "MEDIUM",
            "platform": "Terraform",
            "files": [
                {
                    "file_name": "assets/queries/terraform/kubernetes/container_allow_privilege_escalation_is_true/test/positive.tf",
                    "similarity_id": "063ed2389809f5f01ff420b63634700a9545c5e5130a6506568f925cdb0f8e13",
                    "line": 11,
                    "issue_type": "IncorrectValue",
                    "search_key": "kubernetes_pod[test3].spec.container.allow_privilege_escalation",
                    "search_value": "",
                    "expected_value": "Attribute 'allow_privilege_escalation' is undefined or false",
                    "actual_value": "Attribute 'allow_privilege_escalation' is true",
                    "value": null
                }
            ]
        }
    ],
    "scan_id": "console",
    "severity_counters": {
        "HIGH": 0,
        "INFO": 0,
        "LOW": 0,
        "MEDIUM": 1
    },
    "total_counter": 1
}
```

Figura 9: Ejemplo de salida de Cherkmarx en formato JSON

Fuente: Documentación de Cherkmarx[30]

El formato de Cherkmarx nos entrega información valiosa como el número de archivos analizados, si es que se presentaron errores durante la ejecución del análisis, número de consultas realizadas y cuáles fueron y, finalmente, el contador de riesgos encontrados según la severidad de estos. Es a partir de este formato que consideramos relevante obtener la severidad de las vulnerabilidades que encontremos en nuestro análisis.

3.8.2 SonarQube

SonarQube es una plataforma de código abierto diseñada para la gestión continua de la calidad del código en proyectos de desarrollo de software. Proporciona una amplia gama de herramientas y análisis estáticos que permiten a los equipos de desarrollo identificar y corregir problemas de calidad en el código fuente de manera eficiente. SonarQube ayuda a evaluar métricas de calidad, detectar vulnerabilidades de seguridad, verificar estándares de codificación y mantener un alto nivel de calidad del código a lo largo del ciclo de vida del desarrollo de software. Esta herramienta es especialmente útil en entornos de desarrollo ágiles y colaborativos, donde la calidad del código es esencial para la entrega exitosa de software.

[30] https://github.com/Checkmarx/kics/blob/master/docs/results.md

```
{ "issues": [
    {
      "engineId": "test",
      "ruleId": "rule1",
      "severity":"BLOCKER",
      "type":"CODE_SMELL",
      "primaryLocation": {
        "message": "fully-fleshed issue",
        "filePath": "sources/A.java",
        "textRange": {
          "startLine": 30,
          "endLine": 30,
          "startColumn": 9,
          "endColumn": 14
        }
      },
      "effortMinutes": 90,
      "secondaryLocations": [
        {
          "message": "cross-file 2ndary location",
          "filePath": "sources/B.java",
          "textRange": {
            "startLine": 10,
            "endLine": 10,
            "startColumn": 6,
            "endColumn": 38
          }
        }
      ]
    },
    {
      "engineId": "test",
      "ruleId": "rule2",
      "severity": "INFO",
      "type": "BUG",
      "primaryLocation": {
        "message": "minimal issue raised at file level",
        "filePath": "sources/Measure.java"
      }
    }
]}
```

Figura 10: Ejemplo de salida en JSON de SonarQube que puede ser obtenido utilizando --report-formats "sonarqube"

Fuente: Documentación de SonarQube[31]

En el caso del formato de salida de SonarQube es en el que más se inspira el formato de

[31]https://docs.sonarqube.org/latest/analyzing-source-code/importing-external-issues/generic-issue-import-format/

salida de nuestra API dado que contiene información relevante como la severidad de la vulnerabilidad, archivo en el que es encontrada y tiempo de ejecución necesario para hallarlo, es de aquí que se decide guardar el archivo específico donde se encuentra la vulnerabilidad como también el almacenar el tiempo de ejecución necesario para realizar el análisis lo que podría ser de interés al realizar el estudio comparativo entre diversas técnicas.

3.8.3 Code Climate

Code Climate es una plataforma de análisis de código y automatización de la revisión de código que ayuda a los equipos de desarrollo a mejorar la calidad y la salud de su código fuente. Proporciona métricas y retroalimentación automatizada sobre la calidad del código, identifica problemas de estilo y potenciales vulnerabilidades de seguridad, y ofrece recomendaciones para realizar mejoras.


```
[
    {
        "type": "issue",
        "check_name": "Disk Encryption Disabled",
        "description": "VM disks for critical VMs must be encrypted with Customer Supplied Encryption Keys (CSEK) or with Customer-m
        "categories": ["Security"],
        "location": {
            "path": "positive1.yaml",
            "lines": {
                "begin": 8
            }
        },
        "severity": "major",
        "fingerprint": "93b3ba78cf3f7aba06e480f40b10c754cb923118abb13e37106e56106406415f"
    },
    {
        "type": "issue",
        "check_name": "IP Forwarding Enabled",
        "description": "Instances must not have IP forwarding enabled, which means the attribute 'canIpForward' must not be true",
        "categories": ["Security"],
        "location": {
            "path": "positive1.yaml",
            "lines": {
                "begin": 16
            }
        },
        "severity": "major",
        "fingerprint": "ac9b7b7b621eeeaa28d50f5cb6a047dd53df706624b509bcc7db775e53de6db5"
    },
    {
        "type": "issue",
        "check_name": "Project-wide SSH Keys Are Enabled In VM Instances",
        "description": "VM Instance should block project-wide SSH keys",
        "categories": ["Security"],
        "location": {
            "path": "positive1.yaml",
            "lines": {
                "begin": 4
            }
        },
        "severity": "major",
        "fingerprint": "e84dabdbe179ec9bbe4fb5c0abe8fe8a0f8a3b679a8eef6cfc98d7e5403600ab"
    },
```

Figura 11: Code Climate ejemplo de salida en JSON

Fuente: Repositorio y documentación de Code Climate[32]

En el ejemplo de salida de Code Climate se puede ver una estructura bastante similar a la que nosotros proponemos pero solo para vulnerabilidades, donde los campos que comparte con nuestro *output* propuesto son el nombre de la vulnerabilidad, la categoría, el archivo donde se encuentra especificando la línea y la severidad de este. Siendo este reporte un ejemplo de que la estructura de salida planteada para nuestro análisis está basada en las herramientas presentes en el mercado por lo que la información obtenida mediante las técnicas otorgará valor a los usuarios del sistema.

3.8.4 Formato de salida propuesto

A partir de los formatos de salida que se encuentran en el mercado es que el formato propuesto para la forma es el siguiente:

[32] Fuente: https://github.com/CycloneDX/specification/blob/master/schema/ext/vulnerability-1.0.xsd

```
{
  "app_name": "MyApp",
  "app_version": "1.0.0",
  "analysis_date": "YYYY-mm-dd",
  "vulnerabilities": [
    {
      "name": "Name",
      "description": "Description",
      "severity": "High/Medium/Low",
      "file": "path",
      "line": 0
    }
  ],
  "bad_practices": [
    {
      "name": "Name",
      "description": "Description",
      "file": "path",
      "line": 0
    }
  ],
  "malware": [
    {
      "name": "Name",
      "description": "Description",
      "file": "path"
    }
  ]
}
```

Figura 12: Propuesta de formato de salida para API sin ejemplos

Fuente: Elaboración propia

El cual busca otorgar información importante como el nombre del paquete de la aplicación, la versión de esta, vulnerabilidades encontradas, malas prácticas encontradas y posible malware encontrado. El cual puede ser ejemplificado de la siguiente manera:

```json
{
  "app_name": "MyApp",
  "app_version": "1.0.0",
  "analysis_date": "2023-05-17",
  "vulnerabilities": [
    {
      "name": "SQL Injection",
      "description": "The application is vulnerable to SQL injection attacks.",
      "severity": "High",
      "file": "src/com/example/MainActivity.java",
      "line": 42
    },
    {
      "name": "Cross-Site Scripting (XSS)",
      "description": "The application has a cross-site scripting vulnerability.",
      "severity": "Medium",
      "file": "src/com/example/Utils.java",
      "line": 76
    }
  ],
  "bad_practices": [
    {
      "name": "Hardcoded Credentials",
      "description": "Sensitive credentials are hardcoded in the source code.",
      "file": "src/com/example/Config.java",
      "line": 18
    },
    {
      "name": "Insecure Encryption",
      "description": "The application uses weak encryption algorithms.",
      "file": "src/com/example/CryptoUtils.java",
      "line": 58
    }
  ],
  "malware": [
    {
      "name": "Trojan.Agent",
      "description": "A malware Trojan.Agent was detected in the file.",
      "file": "lib/malware.dll"
    }
  ]
}
```


Figura 13: Propuesta de formato de salida para API

Fuente: Elaboración propia

3.9 Metodología para el análisis de resultados

Para realizar el análisis de resultados se realizará un estudio por técnica, en cuanto a los reportes que estos generan y la información que es obtenible a partir de esto, también se realizará un análisis de rendimiento de las técnicas para ver información tal como la aplicabilidad de la técnica a un conjunto de aplicaciones y los tiempos de ejecución requeridos para obtener un reporte, finalmente se realizará la comparación entre información que cada técnica otorga, las diferencias de información que es otorgada por cada técnica y al mismo tiempo el cómo esta información se complementa entre sí.

CAPÍTULO 4: IMPLEMENTACIÓN

Este capítulo tiene como objetivo explicar el proceso realizado para implementar las técnicas seleccionadas en el capítulo anterior utilizando las herramientas planteadas, como también dar a conocer problemáticas o decisiones tomadas en la implementación con el fin de que este estudio pueda ser replicado o discutido en el futuro.

4.1 Levantamiento de la API

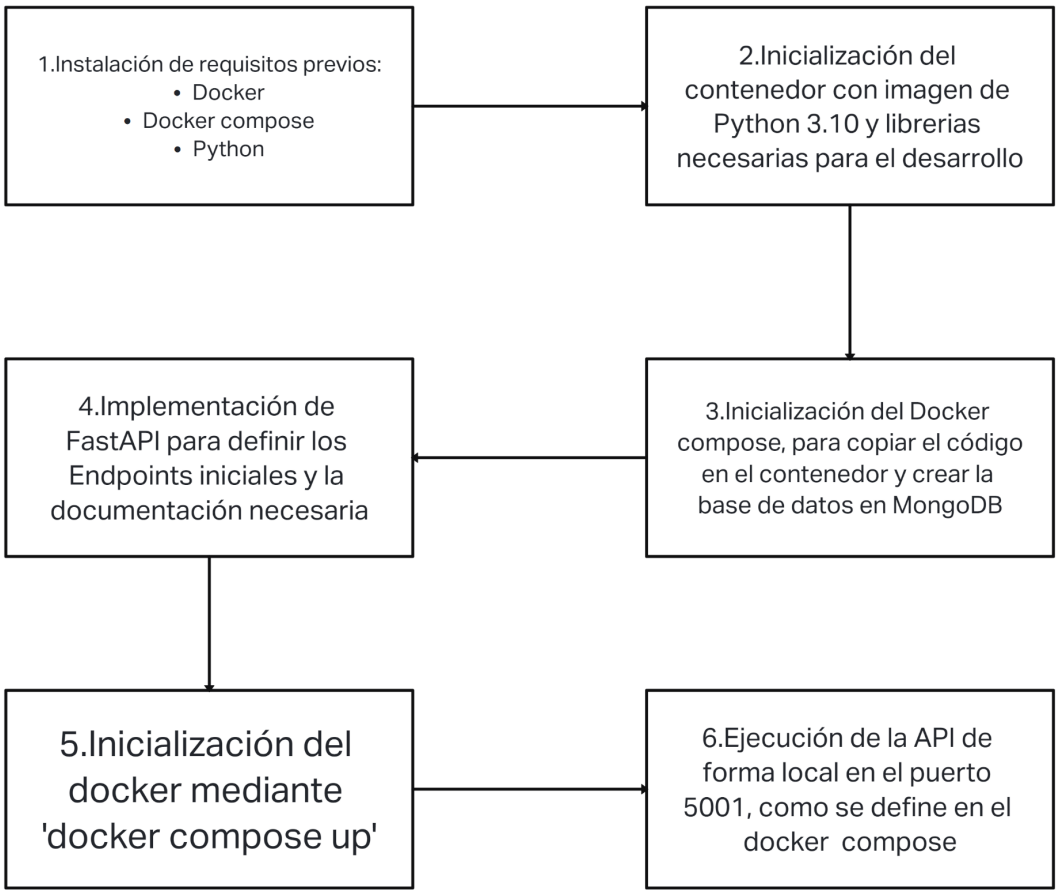


Figura 14: Diagrama para implementar la API

Fuente: Elaboración propia

El primer paso realizado para la implementación, fue el desarrollar una API base que

implementa las operaciones de crear, obtener, actualizar y borrar un reporte, a pesar de que además de la función de obtener reportes, el resto no debería ser utilizado por un externo de manera directa.

Para el levantamiento de la API se creó un contenedor en Docker que utiliza la imagen python 3.10 el cual instala las librerías correspondientes. Para el levantamiento de este contenedor se utilizará Docker Compose, que agrega el código de la API al contenedor y también se encarga de crear la base de datos en MongoDB donde serán almacenados los reportes de las diversas aplicaciones a analizar.

```
FROM python:3.10

WORKDIR /code

COPY ./app /code/app
COPY ./requirements.txt /code/requirements.txt

RUN apt-get update \
    && apt-get install -y default-libmysqlclient-dev \
    && pip install --no-cache-dir --upgrade -r /code/requirements.txt

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80", "--reload"]
```

Figura 15: Docker para el levantamiento del entorno

Fuente: Elaboración propia

```yaml
services:
  service_app_reports:
    build: .
    ports:
      - "5001:80"
    volumes:
      - .:/code

  reports_mongodb:
    image: mongo:5.0
    volumes:
      - reports_mongodb_container:/data/db
    logging:
      driver: none

volumes:
  reports_mongodb_container:

networks:
  default:
    name: service_app
    external: true
```

Figura 16: Docker Compose para el levantamiento del contenedor y su base de datos.

Fuente: Elaboración propia

Una vez levantado el entorno, el siguiente paso es implementar la API en python y, utilizando como principal framework de desarrollo FastAPI, este entorno es levantado por el comando del Docker y puede ser accedido a través del puerto local 5001, la cual puede ser utilizada directamente a través de los endpoints definidos utilizando herramientas como PostMan o el navegador web, pero el recomendado es a través de la documentación de ésta dado que se especifican los modelos utilizados, la versión, los endpoints con sus respectivos formatos de entrada y salida, como también una descripción de la API.

Figura 17: Documentación de la API implementada.

Fuente: Elaboración propia

Una vez ya terminada la base de la API, es que se comenzó con el desarrollo de cada una de las técnicas que podrá ser accedida a través de cada endpoint respectivo y a través del formato de **Consulta** que debe traer el nombre del paquete y su versión para ser analizados en el caso de no estar en la base de datos u obtenidos desde esta.

4.2 Técnica de análisis sintáctico

El primer paso realizado para implementar la técnica fue definir el *endpoint* en el API que realizará el llamado a esta técnica, que a partir del nombre del paquete y su versión obtendrá el APK. Posteriormente utilizando la librería de python **androguard** es que transformamos el APK a código de bajo nivel Smali que consiste principalmente de instrucciones realizadas por las distintas clases y métodos de la aplicación, los cuales pueden ser analizados a través de la identificación de posibles comportamientos peligrosos.

Con el fin de analizar las distintas vulnerabilidades y malas prácticas, se establece la relación entre la vulnerabilidad y la expresión que podría dar origen a esta, como también el nivel de riesgo que presenta. Con el fin de llevar a cabo la implementación, se definió una expresión para vulnerabilidades en el top 25 de vulnerabilidades de la CWE que puedan ser analizadas mediante el código de la aplicación dado que algunas están más asociadas al comportamiento de la aplicación, las cuales son presentadas a continuación con su nivel de riesgo asociado según la CWE y la OWASP:

Nombre	Código CWE	Nivel de Riesgo	Instrucción
Uso de almacenamiento no seguro	CWE-276	Alto	"if 'getExternalStorageDirectory' in instruction.get_output():"
Uso de HTTP en lugar de HTTPS	CWE-319	Medio	"if 'http://' in instruction.get_output():"
Uso de un TrustManager inseguro	CWE-319	Alto	"if 'TrustManager' in instruction.get_output():"
Uso inseguro de HttpClient	CWE-319	Alto	"if 'HttpClient' in instruction.get_output():"
Uso inseguro de HttpURLConnection	CWE-319	Alto	"if 'HttpURLConnection' in instruction.get_output():"
Uso inseguro de función criptográfica	CWE-327	Alto	"if 'Cipher' in instruction.get_output():"
Deserialización insegura	CWE-502	Alto	"if 'readObject' in instruction.get_output():"
Uso de Intents implícitos peligrosos	CWE-589	Alto	"if 'startActivity' in instruction.get_output() and 'ACTION_VIEW' in instruction.get_output():"
Uso de WebView sin configuración segura	CWE-602	Medio	"if 'WebView' in instruction.get_output() and 'setJavaScriptEnabled' in instruction.get_output():"
Uso inseguro de WebView addJavascriptInterface	CWE-602	Alto	"if 'WebView' in instruction.get_output() and 'addJavascriptInterface' in instruction.get_output():"

Uso inseguro de WebView setWebChromeClient	CWE-602	Medio	"if 'WebView' in instruction.get_output() and 'setWebChromeClient' in instruction.get_output():"
Uso inseguro de WebView setWebViewClient	CWE-602	Medio	"if 'WebView' in instruction.get_output() and 'setWebViewClient' in instruction.get_output():"
Uso inseguro de WebView loadUrl	CWE-602	Medio	"if 'WebView' in instruction.get_output() and 'loadUrl' in instruction.get_output():"
Uso inseguro de WebView loadData	CWE-602	Medio	"if 'WebView' in instruction.get_output() and 'loadData' in instruction.get_output():"
Uso inseguro de WebView postUrl	CWE-602	Medio	"if 'WebView' in instruction.get_output() and 'postUrl' in instruction.get_output():"
Uso inseguro de WebView loadDataWithBaseURL	CWE-602	Medio	"if 'WebView' in instruction.get_output() and 'loadDataWithBaseURL' in instruction.get_output():"
Uso inseguro de WebView evaluateJavascript	CWE-602	Medio	"if 'WebView' in instruction.get_output() and 'evaluateJavascript' in instruction.get_output():"
Uso inseguro de WebView removeJavascriptInterface	CWE-602	Medio	"if 'WebView' in instruction.get_output() and 'removeJavascriptInterface' in instruction.get_output():"
Uso inseguro de WebView	CWE-602	Medio	"if 'WebView' in

setHttpAuthUsernamePassword			instruction.get_output() and 'setHttpAuthUsernamePassword' in instruction.get_output():"
Uso inseguro de WebView savePassword	CWE-602	Medio	"if 'WebView' in instruction.get_output() and 'savePassword' in instruction.get_output():"
Uso inseguro de WebView clearCache	CWE-602	Medio	"if 'WebView' in instruction.get_output() and 'clearCache' in instruction.get_output():"
Uso inseguro de WebView clearHistory	CWE-602	Medio	"if 'WebView' in instruction.get_output() and 'clearHistory' in instruction.get_output():"
Uso inseguro de SQLiteQueryBuilder	CWE-89	Medio	"if 'SQLiteQueryBuilder' in instruction.get_output():"
Uso inseguro de la función delete de un ContentProvider	CWE-922	Alto	"if 'ContentProvider' in instruction.get_output() and 'delete' in instruction.get_output():"

Tabla 8: Expresiones utilizadas para reconocer vulnerabilidades

Fuente: Elaboración propia

Con esto somos capaces de iterar a través de un APK instrucción por instrucción con el fin de ver si alguna posee un comportamiento que podría dar origen a una vulnerabilidad y generar los reportes. Esta estratégia tiene, como principales puntos a favor, los siguientes:

- Reconocer la clase específica en la que se origina la vulnerabilidad
- Reconocer el método específico en el que se origina la vulnerabilidad.
- Identificar qué línea y que instrucción pueden dar origen a la vulnerabilidad.

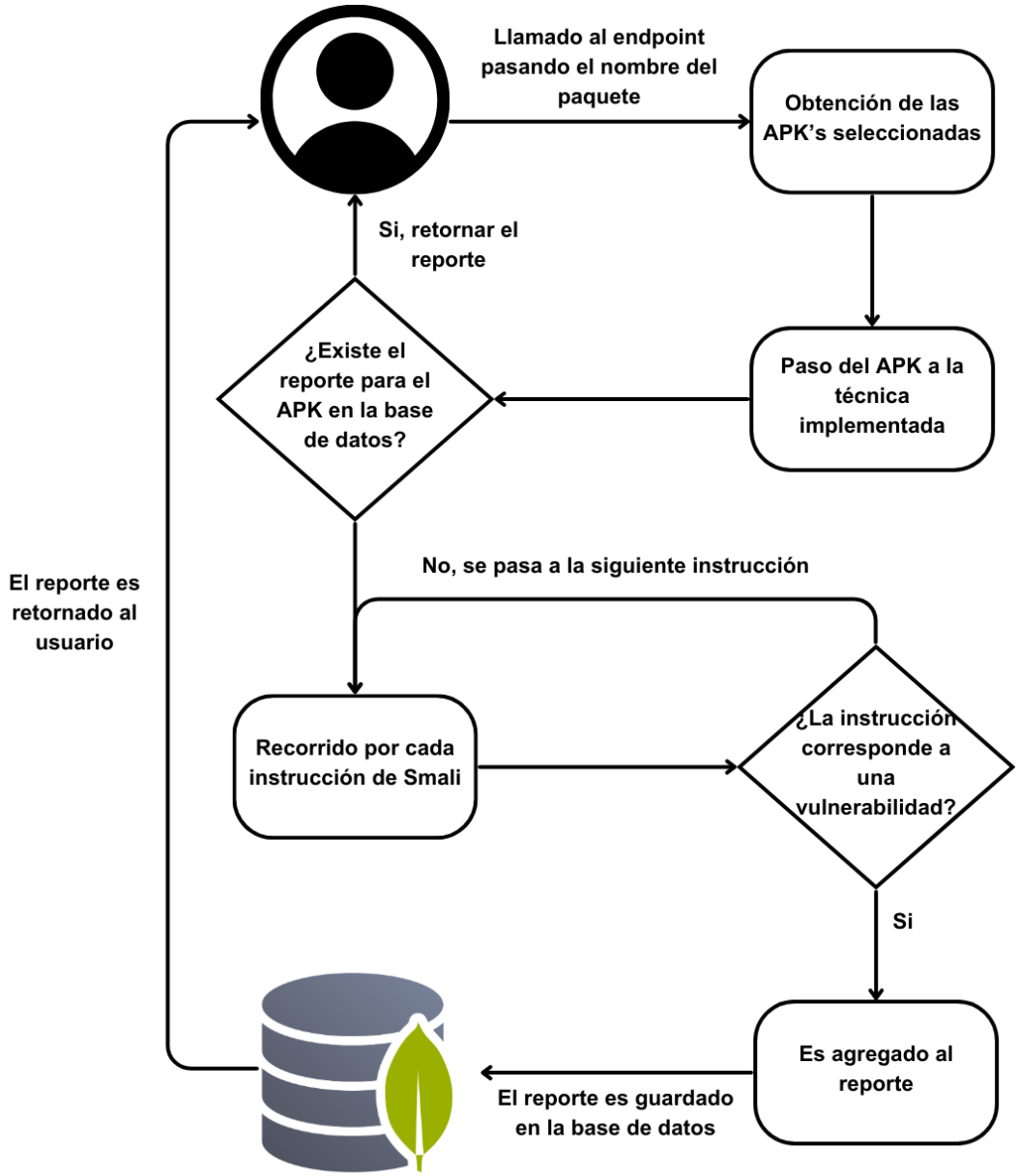


Figura 18: Diagrama de trabajo de análisis sintáctico

Fuente: Elaboración propia

4.3 Análisis a través del grafo de flujo de datos.

Para la implementación de esta técnicas nos basamos en la estrategia desarrollada por [QiLi2015], donde se utilizó el árbol de características de un APK.

En el desarrollo utilizamos python para el procesamiento de la información del APK dado que este nos permite trabajar con el Dalvik de la aplicación, lo cual es una diferencia con el trabajo referenciado dado que estos utilizaban C++ para la transformación del código Dalvik a un grafo. Otra de las diferencias que presentamos, es que el árbol de características utilizado en el estudio tiene 4 capas siendo las de paquete, la de clases, la de métodos y la llamados a la API del SO, pero a diferencia de estas capas nosotros tenemos la diferencia en la llamada a la API del sistema operativo donde nosotros almacenamos las instrucciones, donde cada capa será utilizada para obtener distinta información:

- **Capa de Paquete:** Obtiene el nombre del paquete de la aplicación y puede ser utilizado para analizar si 2 grafos son del mismo paquete o de la misma aplicación, pero variando versiones.
- **Capa de Clases:** Incluye el nombre de la clase y la clase padre, sirve como primer indicador para ver niveles de similaridad con malware o posibles diferencias notables entre aplicaciones del mismo paquete.
- **Capa de Funciones/Métodos:** Incluye el nombre de la función y parámetros, nos da una mayor información respecto a las similitudes o diferencias de las aplicaciones indicándonos si dentro de las clases podrían existir posibles métodos maliciosos.
- **Capa de Instrucciones:** Posee las instrucciones que posee el método, lo que nos podría indicar si la utilización de recursos, malas prácticas o posibles vulnerabilidades son similares entre aplicaciones.

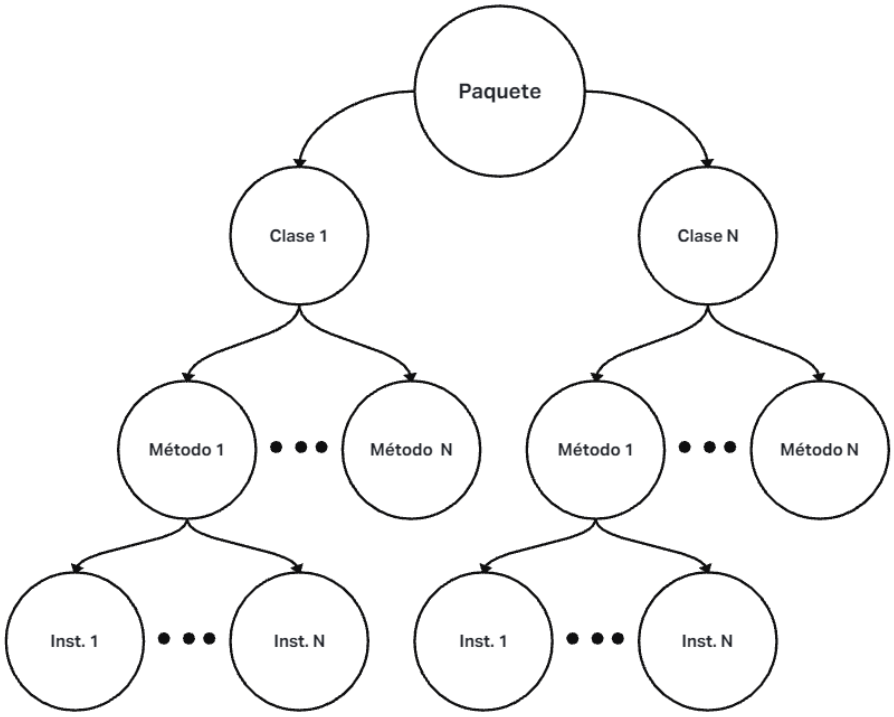


Figura 19: Árbol de características implementado

Fuente: Elaboración propia

Con el grafo de características definido procedimos a implementar el endpoint de análisis de grafos, el cual se encarga de obtener porcentajes de similitud entre cada capa con el fin de analizar qué tan similar es un APK cualquier a APK's almacenados en la base de datos con el fin de ver si existe un porcentaje de similitud con malware o si hay una diferencia significativa con APK's de la misma categoría.

Para esto el primer paso es obtener el APK de malwares, que en este caso obtuvimos de repositorios que se encargan de encontrar APK's interesantes de analizar[33] que obtienen de MalwareBazaar[34], como también APK's de la categoría de mensajería de las cuales generamos sus grafos de características y almacenamos en la base de datos de mongo como grafos serializados en JSON.

[33] https://maldroid.github.io/android-malware-samples/
[34] https://bazaar.abuse.ch/

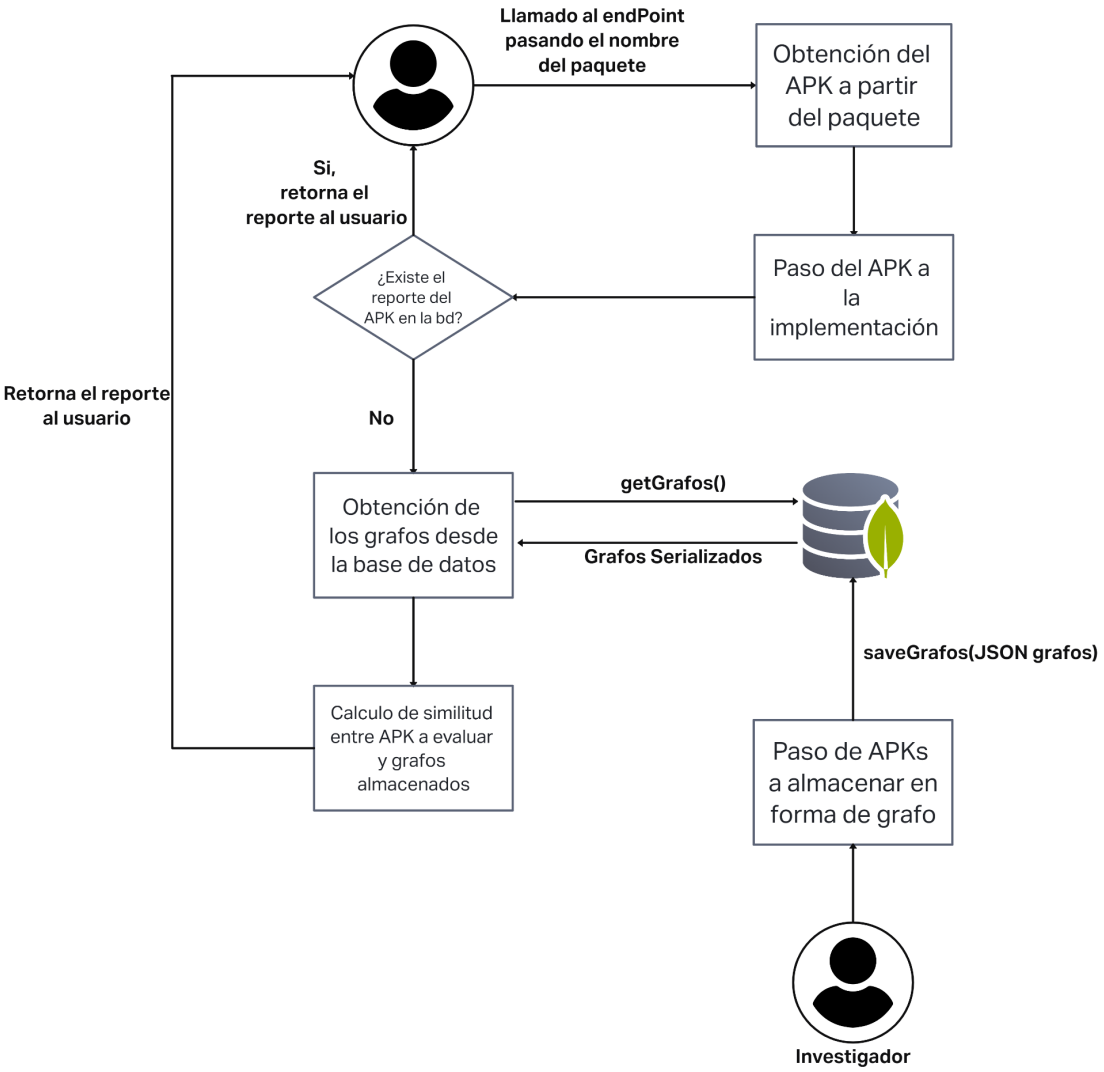


Figura 20: Flujo de trabajo para análisis mediante grafo de características

Fuente: Elaboración propia

Posteriormente una vez almacenadas las APK's con código malicioso o de interés es que entregamos la aplicación que deseamos comparar con las almacenadas, lo que nos retornará un reporte a través de la API con el porcentaje de similitud entre capas en escala del 0 al 1. Como el siguiente ejemplo:

Response body


```
  {
    "appname": "FacebookMessenger.apk",
    "apk_packages_similarity": 0,
    "classes_similarity": 0.002954112781461301,
    "functions_similarity": 0.03134404633467719,
    "instructions_similarity": 0.9154929577464789
  },
  {
    "appname": "Facebook.apk",
    "apk_packages_similarity": 0,
    "classes_similarity": 0.005658953722334004,
    "functions_similarity": 0.03239944521497919,
    "instructions_similarity": 0.9436619718309859
  },
  {
    "appname": "Whatsapp.apk",
    "apk_packages_similarity": 0,
    "classes_similarity": 0.010689432527440992,
    "functions_similarity": 0.05612552806276403,
    "instructions_similarity": 0.9142857142857143
  },
  {
    "appname": "SMSstealer.apk",
    "apk_packages_similarity": 0,
    "classes_similarity": 0.04480651731160896,
    "functions_similarity": 0.1721759945286675,
    "instructions_similarity": 0.9238095238095239
  },
```


Figura 21: Ejemplo de reporte de similitud de grafos

Fuente: Elaboración propia

Para el cálculo de similitud entre dos grafos se manejaron distintas alternativas que fueran capaces de comparar dos conjuntos de nodo, entre las cuales estaban:

- **Índice de Jaccard**: Corresponde a la proporción de la intersección de 2 grafos considerando sus nodos o aristas en base a la unión de ambos grafos. Es una medida popular para el análisis de grafos debido a que otorga información sencilla de interpretar y al mismo tiempo representativa de la similitud entre grafos.
- **Índice de Hamming:** Mide la fracción de características distintas entre dos grafos, es útil cuando los grafos poseen características binarias donde es relativamente sencillo decir si son iguales o diferentes.
- **Índice de similitud entre grafos:** Mide la similitud entre dos grafos mediante la distribución de subgrafos o patrones en común.
- **Coeficiente de correlación de Pearson:** Mide la correlación lineal entre dos conjuntos de características numéricas.

Dentro de las opciones barajadas se consideró que las que era más aplicables al grafo de

características eran el de Jaccard o el de similitud entre grafos, pero se decidió por el primero debido a diversos motivos:

- El primer motivo es que era el utilizado por [QiLi2015] siendo una forma efectiva de calcular la similitud entre dos aplicaciones que sean malware, con el fin de identificar a qué tipo de malware corresponde. Por lo que inspirado en esto consideramos que sería una buena forma de calcular la similitud para cualquier APK y un posible malware.
- El segundo motivo es que el índice de Jaccard toma en consideración el grafo en su totalidad, mientras que el índice de similitud entre grafo sólo considera subconjuntos de éste, por lo que podría dar similitudes que no necesariamente sean por la codificación en sí sino que por la utilización de un lenguaje de bajo nivel como lo es Smali.
- La implementación e interpretación del índice de Jaccard es más sencilla que la del índice de similitud entre grafos, por lo que considerando que el análisis se encuentra dentro del contexto de una API que será consumida por usuarios es que se decidió que es mejor para estos el obtener un dato más sencillo de interpretar.

Por los motivos nombrados anteriormente es que se decidió el índice de Jaccard como la forma de cálculo a implementar para obtener la similitud entre dos aplicaciones a partir de su grafo de características, siempre y cuando se cumplan las siguientes reglas:

$$C_1 = \{ c_i \mid nodo\ clase\ perteneciente\ al\ APK_1 \}$$

$$C_2 = \{ c_j \mid nodo\ clase\ perteneciente\ al\ APK_2 \}$$

$$F_1 = \{ f_i \mid nodo\ función\ perteneciente\ al\ APK_1 \}$$

$$F_2 = \{ f_j \mid nodo\ función\ perteneciente\ al\ APK_2 \}$$

$$I_1 = \{ i_i \mid nodo\ instrucción\ perteneciente\ al\ APK_1 \}$$

$$I_2 = \{ i_j \mid nodo\ instrucción\ perteneciente\ al\ APK_2 \}$$

$$N_1 = \{ n_i \mid nodo\ clase/función/instrucción\ perteneciente\ al\ APK_1 \}$$

$$N_2 = \{ n_j \mid nodo\ clase/función/instrucción\ perteneciente\ al\ APK_2 \}$$

$$Similitud\ (N_1, N_2) = \frac{N_i \cap N_j}{N_i \cup N_j}$$

Con esto podemos saber si el comportamiento de la aplicación tiene alguna semejanza con malware ya identificado, lo que podría ser utilizado para advertir al usuario respecto a estas semejanzas o el ser un indicador para llevar a cabo un análisis más exhaustivo al código con el fin de identificar por qué motivo el código Dalvik tiene tantas semejanzas con un malware.

4.4 Análisis de permisos

Con el fin de analizar los permisos de una aplicación y los riesgos que significa el otorgar acceso a estos para el usuario final es que el primer paso fue definir un catálogo de permisos de Android el cual elaborado por el profesor Felipe Beroíza junto a Benjamín Pizarro que se encuentra disponible en Google Drive[35]. Este catálogo contiene información del permiso tal como el nombre del permiso que tendrá en el *Manifest* de la aplicación, la descripción del permiso tanto en inglés como en español, las palabras claves del permiso, el grupo del permiso asociado al tipo de API que consume desde el sistema operativo que podría ser de Localización, Micrófono, Contactos, entre otros, y finalmente el nivel de protección que el permiso posee que puede ser *Dangerous, Normal, Signatura, SignatureOrSystem.*

Con esta asociación entre permiso y riesgo que significa para el usuario se procedió a implementar el endpoint para la técnica de análisis de permisos, el cual consiste principalmente es obtener el APK de la aplicación mediante el nombre de su paquete el cual es procesado mediante la librería de Androguard que permite obtener un JSON con todos los permisos definidos por la aplicación en su Manifest, los que son procesados uno a uno buscando las columnas de Nombre, Descripción en inglés, Grupo y Nivel de Protección, siendo la columna de nombre la utilizada para obtener los datos. A partir de estos se genera un reporte con las columnas anteriormente nombradas y el nombre utilizado por la plantilla.

[35] Tabla relacion permisos Android

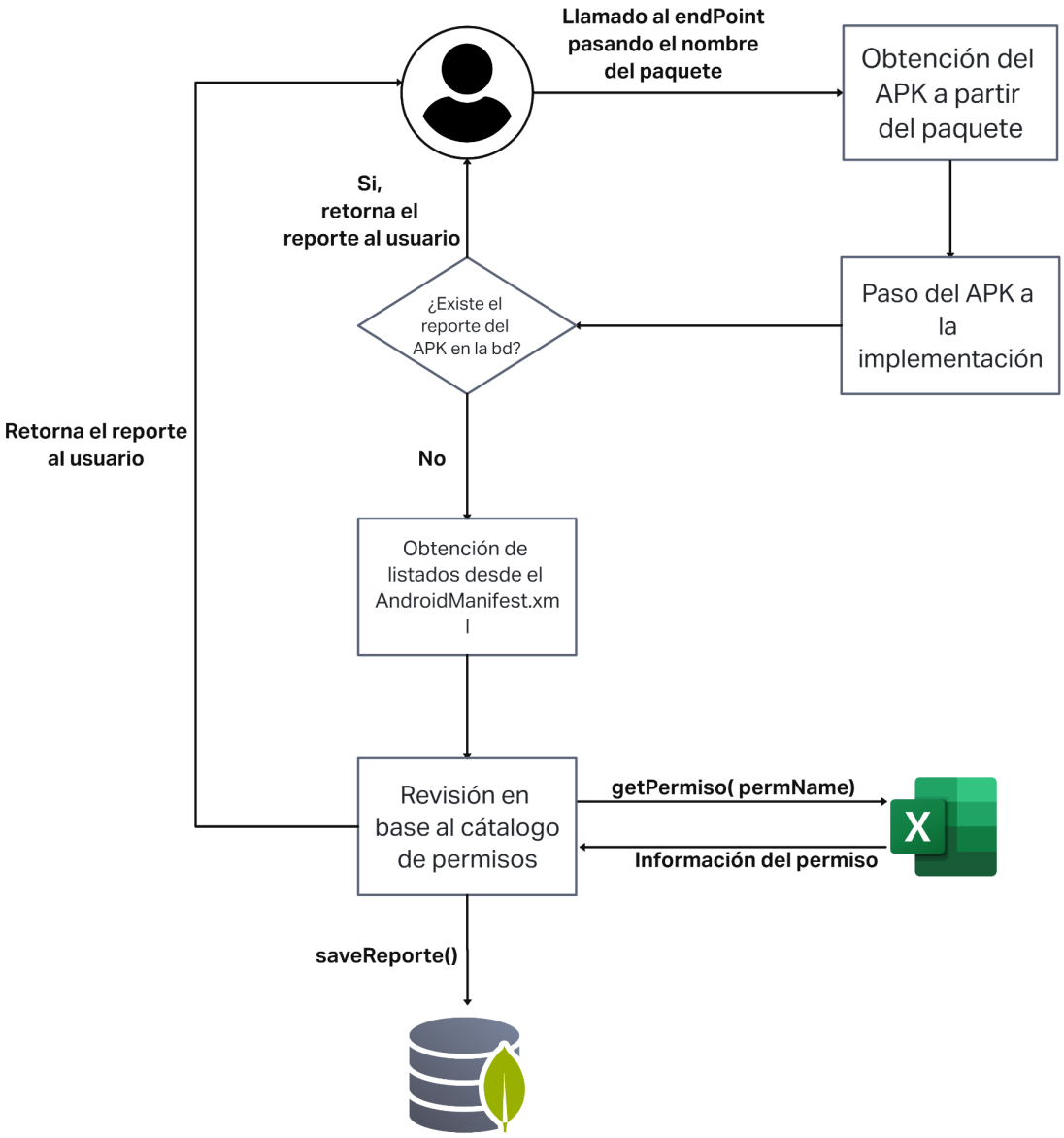


Figura 22: Diagrama de trabajo para análisis de permisos

Fuente: Elaboración propia

Es importante considerar que dentro del proceso de implementación de esta técnica, se encontró que en el Manifest se definen permisos para sistemas operativos o equipos específicos, los cuales no pueden ser procesados dado que, al no ser definidos por

Android, no se conocen los riesgos que estos podrían significar para el usuario y otro error que ocurre es que no hay un listado de estos como sí ocurre con el catálogo de permisos existentes publicado por Android[36].

4.5 Análisis mediante Inteligencias Artificiales de procesamiento de texto

El primer acercamiento al análisis de código mediante inteligencias artificiales que utilizan prompting fue a través de Chat GPT de OpenAI, debido a la popularidad que ha tenido esta herramienta durante los últimos meses, para esto se generó una técnica que nos devolviera el código fuente del APK en lenguaje Smali. Pero esto generó diversas complicaciones en el análisis debido a las tecnologías utilizadas, los cuales fueron:

1. FastAPI tiene un número máximo de caracteres capaz de devolver, por lo que no podía retornar el código fuente de toda la aplicación a través del endpoint.
2. Dado que la aplicación está funcionando en un Docker, también existían problemas de memoria y utilización de memoria que tampoco permitían el retorno de todo el código fuente debido a que excedían los recursos del contenedor provocando que FastAPI dejará de funcionar.
3. No era posible guardar todo el código fuente de una aplicación en MongoDB por limitaciones de memoria y el hecho de que no seguiría una lógica en la base de datos, sino que sería un string excesivamente largo almacenado en la base de datos.

Por otro lado también se presentaron problemas y limitaciones en cuanto a GPT 3.5, siendo estas el límite de caracteres que puede tener una consulta por lo que era imposible pasarle el código fuente de una aplicación que tuviera más de 3.125 palabras[37].

Debido a estas limitaciones, se hicieron modificaciones al método, para que en lugar de retornar el código fuente de toda la aplicación solo se retornará el de clases específicas, esto soluciono los problemas de memoria que limitaban las respuestas por lo que si era posible obtener el código a través de la API y también era posible pasarla a Chat GPT mediante prompting, el problema ocurre que al pasarle código de clases sin un filtro detrás de por qué se está pasando esa clase, la respuesta de la IA es que necesita más contexto para realizar el análisis como se puede ver en la siguiente imagen.

[36] https://developer.android.com/reference/android/Manifest.permission
[37] https://openai.com/pricing

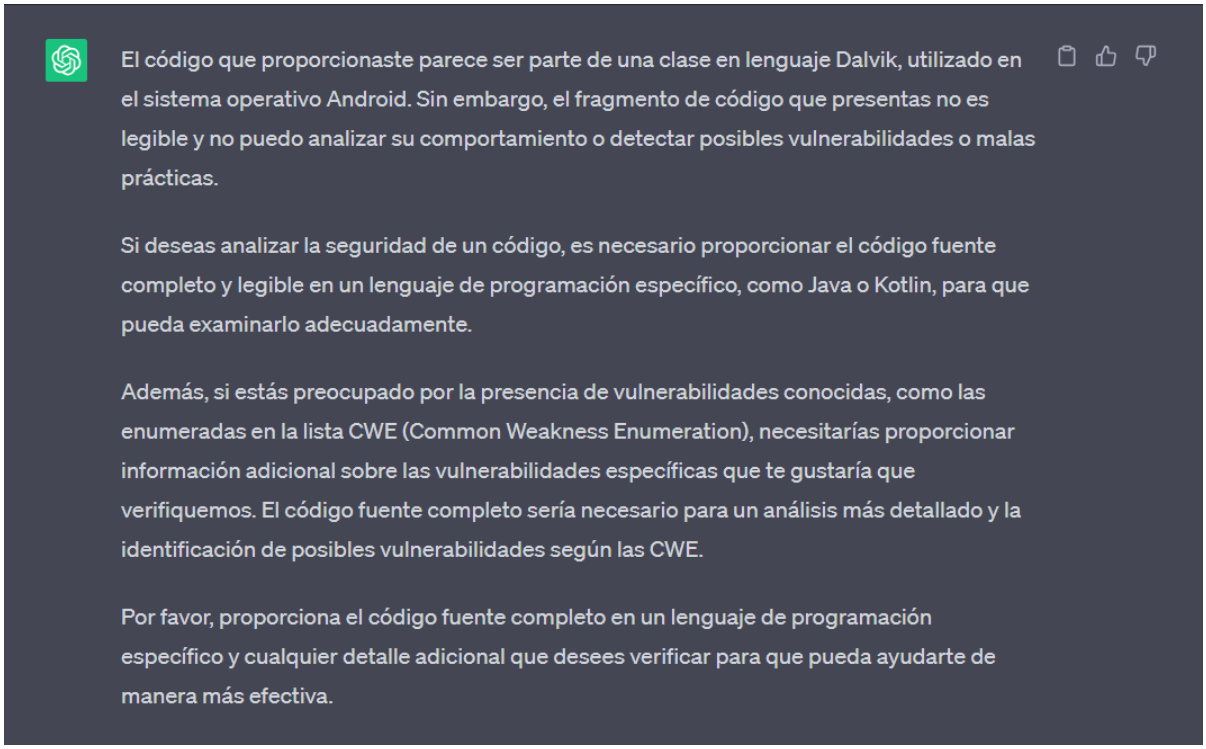


Figura 23: Respuesta de Chat GPT sin contexto

Fuente: Elaboración propia

Es por esto que finalmente se planteó un método combinado que aplicaría grafos para la identificación de clases en común entre el APK y malware, las cuales serían pasadas a ChatGPT para su análisis. De esta forma se espera que al pasar un conjunto de clases con comportamiento sospechoso la respuesta de GPT sea un análisis más completo.

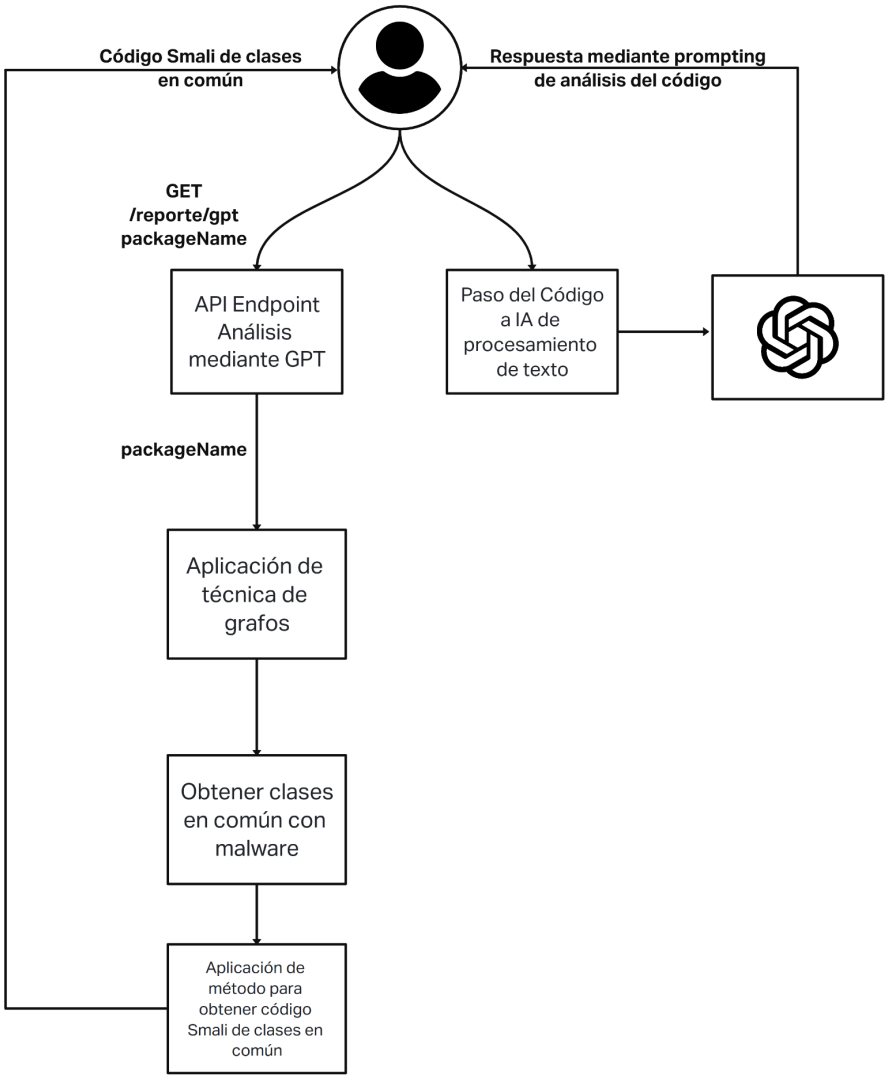


Figura 24: Diagrama de trabajo de análisis de código estático utilizando Inteligencias Artificiales de procesamiento de texto.

Fuente: Elaboración propia

El problema surgió al implementar el método planteado anteriormente, dado que sufrio de las mismas limitaciones que el método planteado originalmente, debido a que al comparar APK's de distintas aplicaciones el número de clases en común varía bastante

pero sigue siendo imposible pasarle todo el código a ChatGPT, por lo que se planteó un tope para la cantidad de clases en común que serían almacenadas pero este tope generaba el mismo error que ocurría al pasar clases de una en una que es que GPT requiere más contexto para realizar el análisis de código.

CAPÍTULO 5: VALIDACIÓN DE LA SOLUCIÓN

Una vez realizado el proceso de implementación mencionado anteriormente se procedió a validar la solución en 300 aplicaciones seleccionadas a partir de una base de datos que contiene un listado con todas las aplicaciones de la Play Store[38], desde donde seleccionamos aquellas con más descargas a nivel mundial.

5.1 Análisis de resultados por técnica

5.1.1 Resultados de análisis sintáctico

El primer dato que se buscaba obtener mediante el análisis de código sintáctico fue la identificación de la vulnerabilidad más común entre todas las aplicaciones, la cual como se puede ver en la Figura 25 fue *HttpURLConnection* el cual está asociado a la CWE-319 Transmisión de información sensible a través de texto, lo que quiere decir que muchas de las conexiones a través de internet de las aplicaciones como pueden ser conexiones bases de datos, envíos de información, conexiones a la nube, conexiones a redes internas, etc. Lo cual podría dar lugar a filtraciones de información sensible por interferencia de actores externos al sistema.

[38] https://www.kaggle.com/datasets/gauthamp10/google-playstore-apps

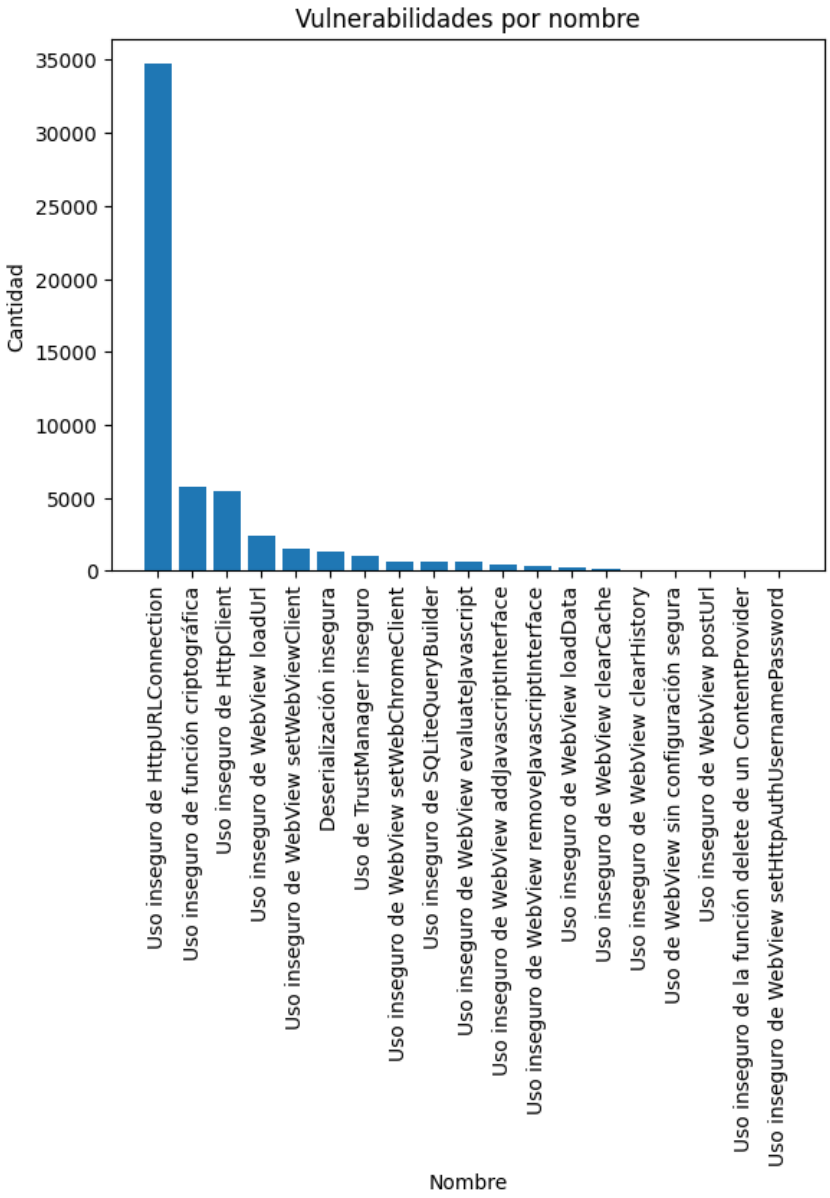


Figura 25: Listado de vulnerabilidades más comunes mediante análisis sintáctico.

Fuente: Elaboración propia

Posteriormente se hizo el análisis por categoría de las aplicaciones, donde el primer análisis fue identificar cuál era la vulnerabilidad más común por categoría.

Categoría	Vulnerabilidad	Cantidad de encuentros
Action	Uso inseguro de función criptográfica	88
Adventure	Uso inseguro de HttpURLConnection	159
Arcade	Uso inseguro de HttpURLConnection	129
Beauty	Uso inseguro de HttpURLConnection	69
Books & Reference	Uso inseguro de HttpURLConnection	291
Business	Uso inseguro de HttpURLConnection	20
Casual	Uso inseguro de HttpURLConnection	133
Communication	Uso inseguro de función criptográfica	2
Education	Uso inseguro de HttpURLConnection	291
Educational	Uso inseguro de HttpURLConnection	66
Entertainment	Uso inseguro de HttpURLConnection	295
Finance	Uso inseguro de HttpURLConnection	182
Health & Fitness	Uso inseguro de HttpURLConnection	126
Maps & Navigation	Uso inseguro de HttpURLConnection	144
Music & Audio	sUso inseguro de HttpURLConnection	171
Photography	Uso inseguro de HttpURLConnection	113
Productivity	Uso inseguro de HttpURLConnection	98
Puzzle	Uso inseguro de HttpURLConnection	133
Racing	Uso inseguro de HttpURLConnection	101
Role Playing	Uso inseguro de HttpURLConnection	171
Shopping	Uso inseguro de HttpURLConnection	97

Simulation	Uso inseguro de HttpURLConnection	217
Social	Uso inseguro de HttpURLConnection	120
Sports	Uso inseguro de HttpURLConnection	123
Strategy	Uso inseguro de HttpURLConnection	88
Tools	Uso inseguro de HttpURLConnection	60
Travel & Local	Uso inseguro de HttpURLConnection	183
Unknown	Uso inseguro de HttpClient	193
Video Players	Uso inseguro de HttpURLConnection	189

Tabla 9: Vulnerabilidades más comunes por categoría

Fuente: Elaboración propia

De la tabla anterior, se desprende que entre todas las categorías, se sigue teniendo que las conexiones a través de *HttpURL* es la vulnerabilidad más común, pero por otro lado llama la atención que en la categoría de Comunicación, la vulnerabilidad más común, sea el “Uso inseguro de función criptográfica” dado que por el manejo de mensajes privados que este tipo de aplicaciones maneja se esperaría que la encriptación de los datos fuera una funcionalidad esencial para estas aplicaciones.

Finalmente se analizó cuál era la categoría que presentaba más vulnerabilidades en promedio, para lo cual se elaboró el siguiente gráfico.

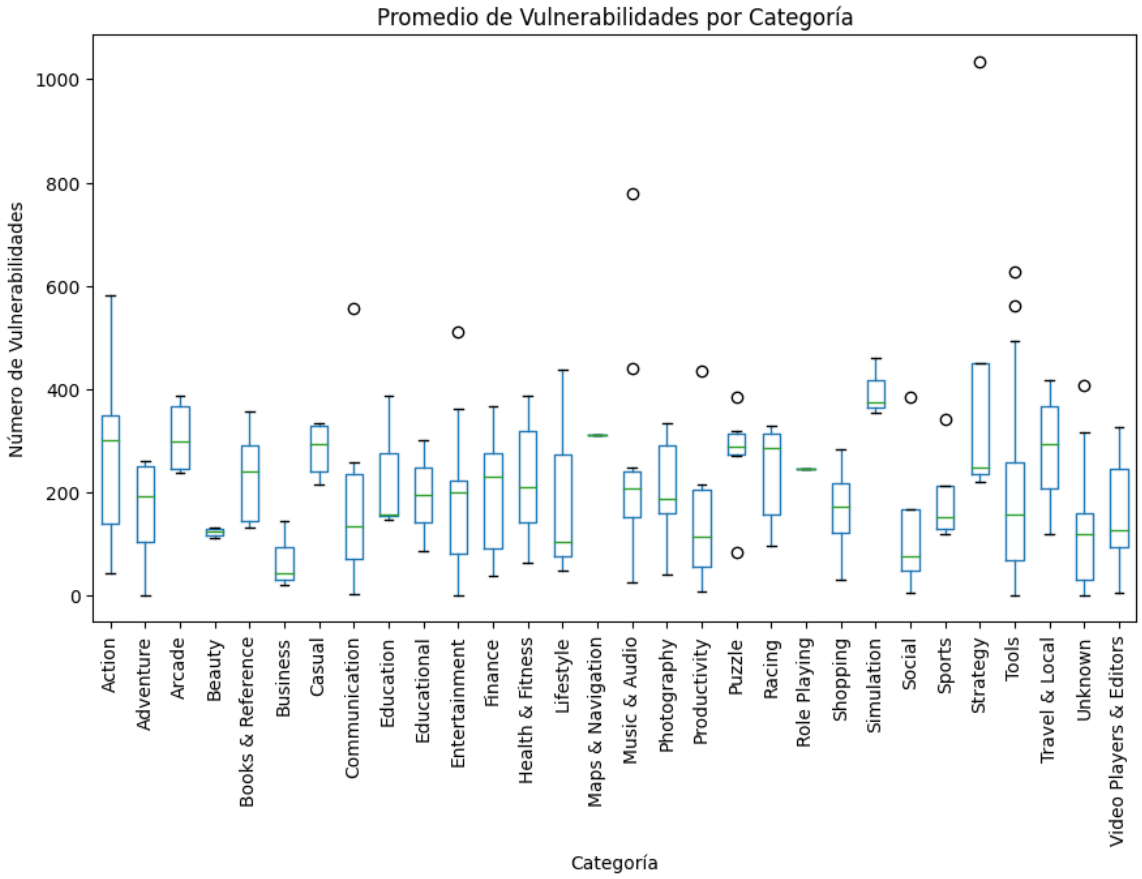


Figura 26: Vulnerabilidades promedio por categoría

Fuente: Elaboración propia

Del gráfico anterior se desprende que la categoría que presenta un mayor número de vulnerabilidades promedio es en las aplicaciones de Simulación, que corresponde a aplicaciones que principalmente son videojuegos de simulación. Por otro lado las categorías de Acción y de Herramientas presentan la mayor desviación, lo cual puede ser causado principalmente por la cantidad de aplicaciones que fueron evaluadas correspondientes a estas categorías. Finalmente se puede ver que varias categorías no presentan boxplots y esto es debido a que la muestra de aplicaciones de aquellas categorías era demasiado baja, por lo que no es posible analizar el comportamiento de categorías como *LifeStyle* o *Role Playing*.

Los principales riesgos que fueron encontrados a través de este análisis es el alto número de conexiones no seguras a internet, lo cual puede ser altamente perjudicial para el usuario de las aplicaciones evaluadas dado que puede causar filtraciones de información

sensible.

5.1.2 Resultados de análisis mediante grafos

Con el fin de analizar los resultados obtenidos mediante la técnica de grafos se aplicó el código generador de grafos a las 300 aplicaciones, de los cuales solo 230 grafos pudieron ser generados para el conjunto de aplicaciones. Posteriormente se itero por cada una de las 300 aplicaciones y fue comparada con el resto de 229 grafos, generando 230 reportes que otorgaban información y 70 que vienen vacíos debido a la imposibilidad de generar los grafos.

A partir de estos reportes generados se buscó analizar el porcentaje de similitud promedio por categoría con el fin de identificar qué información era posible de obtener a partir de los reportes, donde es posible ver en la siguiente figura que los porcentajes de similitud entre clases es inferior al 10%, el porcentaje de similitud entre métodos/funciones es menor a 20% en promedio y el porcentaje por instrucciones se encuentra entre el 80% y 100%. Estos resultados no nos dan mucha información al respecto, pero cabe destacar que la categoría de “Herramientas” tiene promedios inferiores al resto, lo que podría indicar que a medida que hay una mayor cantidad de aplicaciones en la muestra, el promedio tiende a disminuir.

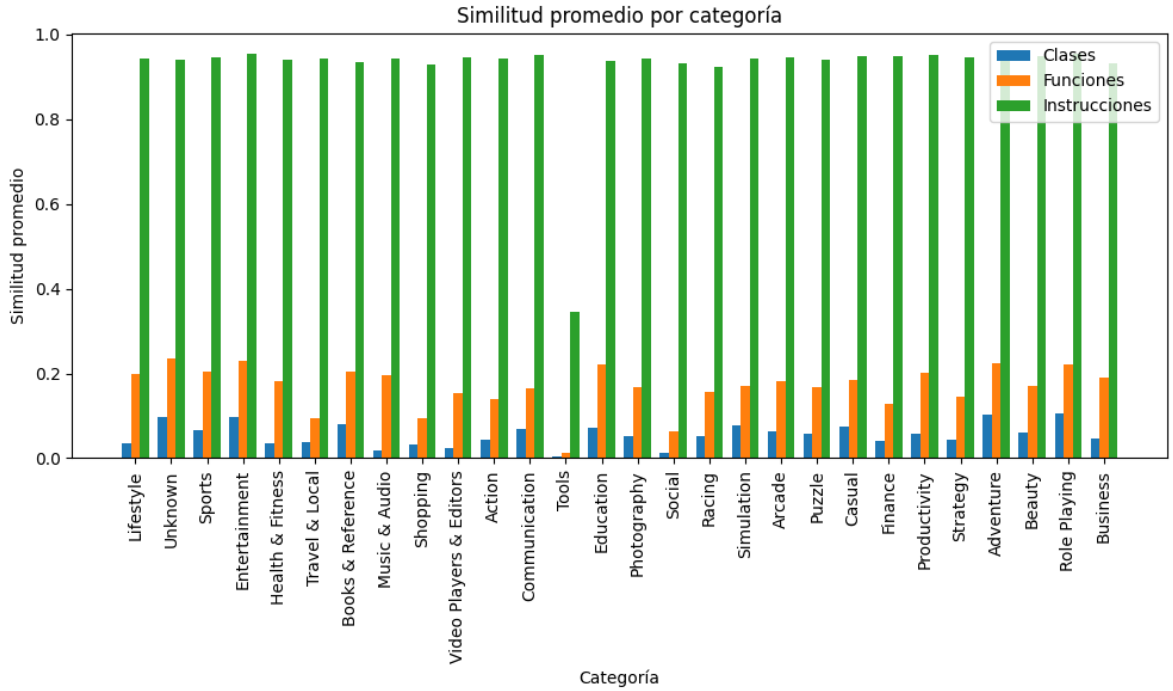


Figura 27: Similitud de aplicaciones por capa según su categoría.

Fuente: Elaboración propia.

A medida que se analizaron los reportes generados, uno de los resultados que se pudo obtener, es el análisis de aplicaciones del mismo desarrollador, dado que estos presentan un aumento significativo en sus porcentajes de similitud superando en algunos casos hasta el 80% en Clases y Métodos/Funciones. El principal valor que este resultado nos otorga, es la posibilidad de aplicar la técnica para identificar si aplicaciones desconocidas, han sido desarrolladas por las mismas personas, lo que podría ser utilizado en casos donde se quiera saber si una aplicación desconocida tiene algún tipo de relación a nivel de desarrollo con algún *malware* ya identificado previamente.

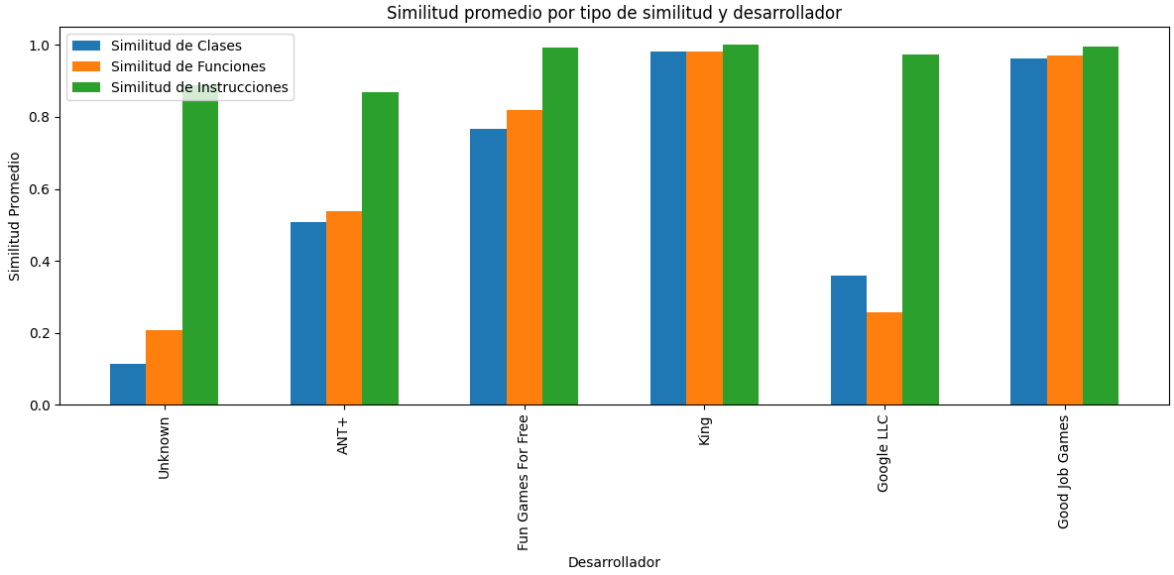


Figura 28: Porcentaje de similitud por capas según desarrollador.

Fuente: Elaboración propia.

5.1.3 Resultados de análisis de permisos

Mediante el análisis de permisos que las aplicaciones declaran en sus *Manifest,* se busca identificar qué aplicaciones son las más comunes en general, aplicaciones más comunes según su nivel de riesgo y si es posible identificar aplicaciones que pidan permisos que no son comunes para su categoría o que piden más permisos que los necesarios.

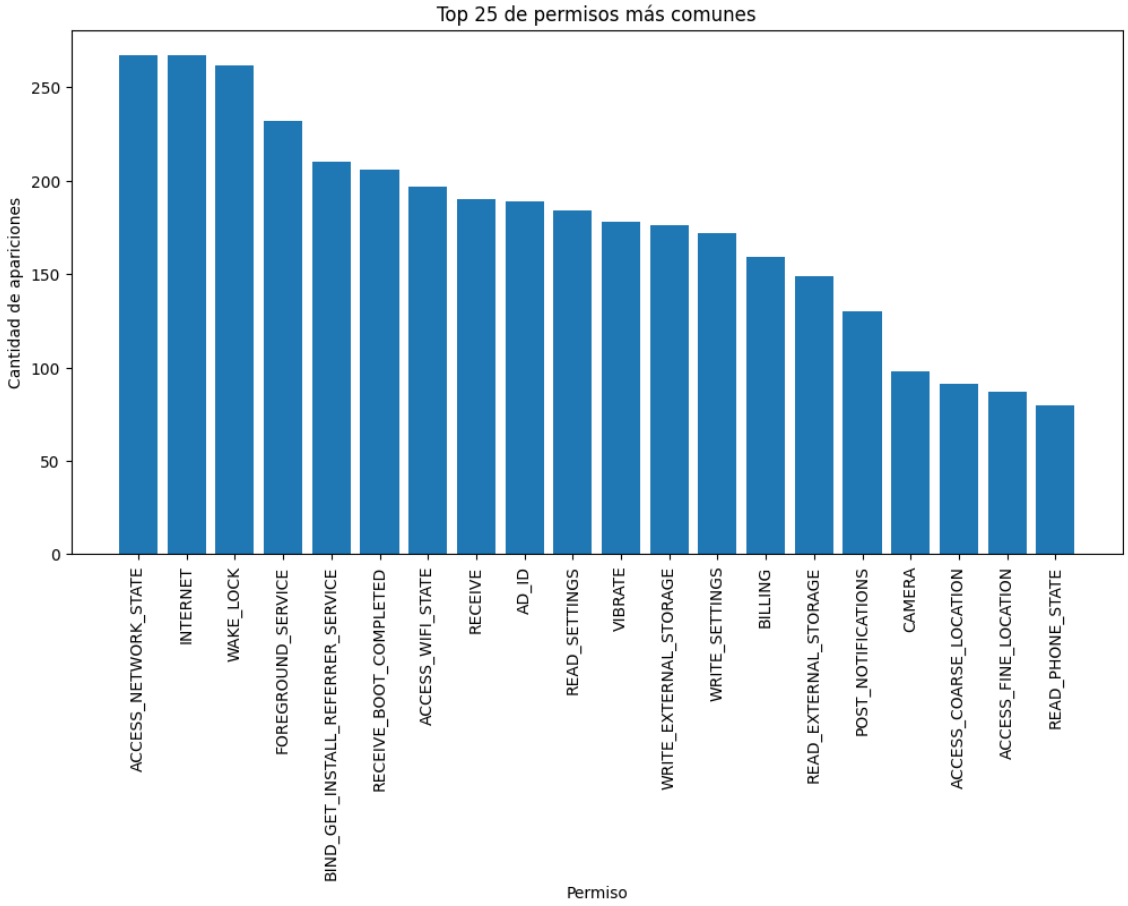


Figura 29: Permisos más utilizados en general

Fuente: Elaboración propia

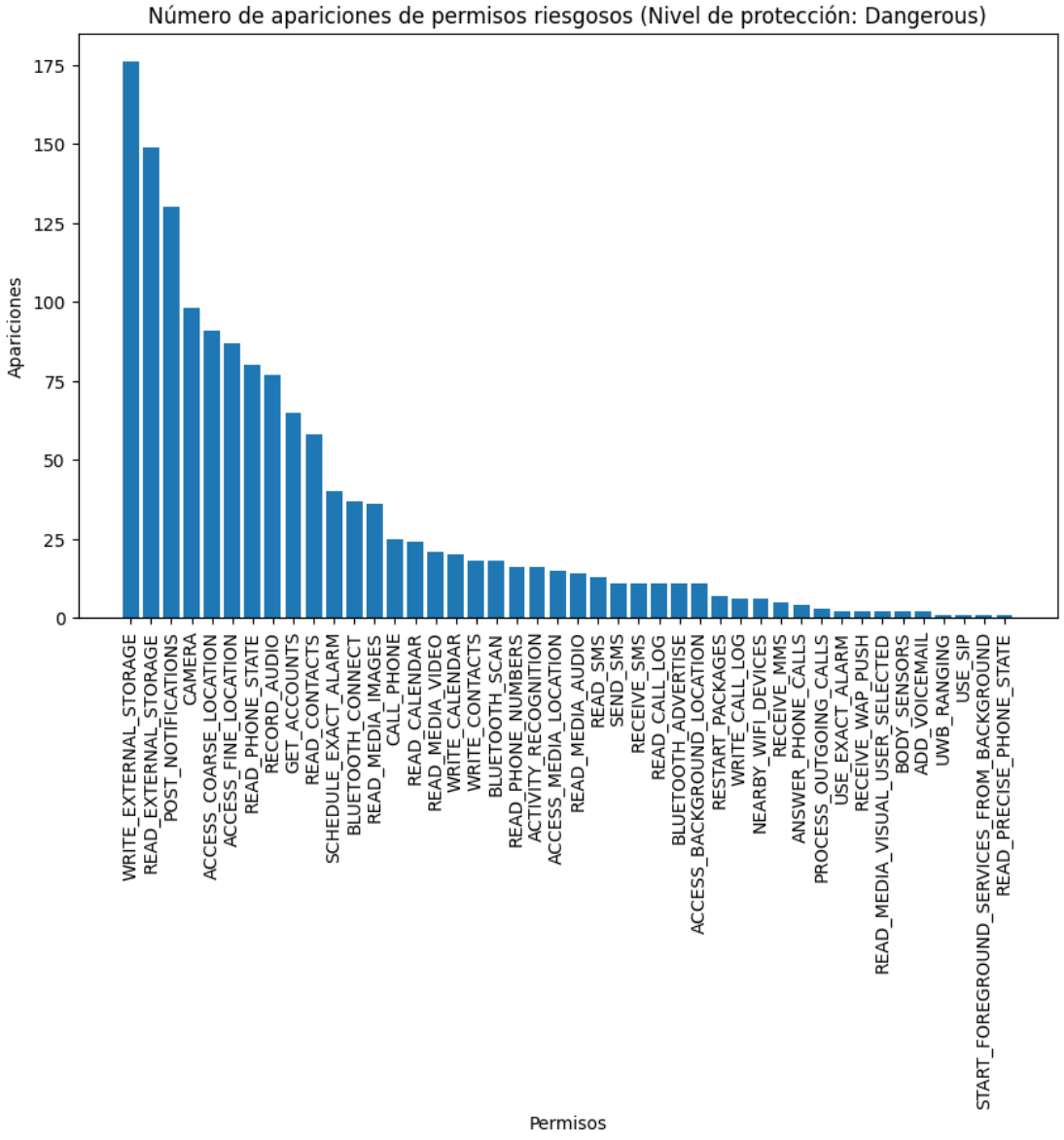


Figura 30: Permisos más utilizados de alto riesgo.

Fuente: Elaboración propia

Del gráfico anterior, se desprende que los permisos riesgosos más comunes son los de escritura y lectura en almacenamiento externo los que pueden dar lugar a peligros como pérdida de datos, robo de información o fuga de información confidencial, como también el de envío de notificaciones que puede dar lugar al *spam* de anuncios no deseado y el acceso a la cámara del teléfono que presenta peligros como la grabación no autorizada y

monitoreo de actividades privadas.

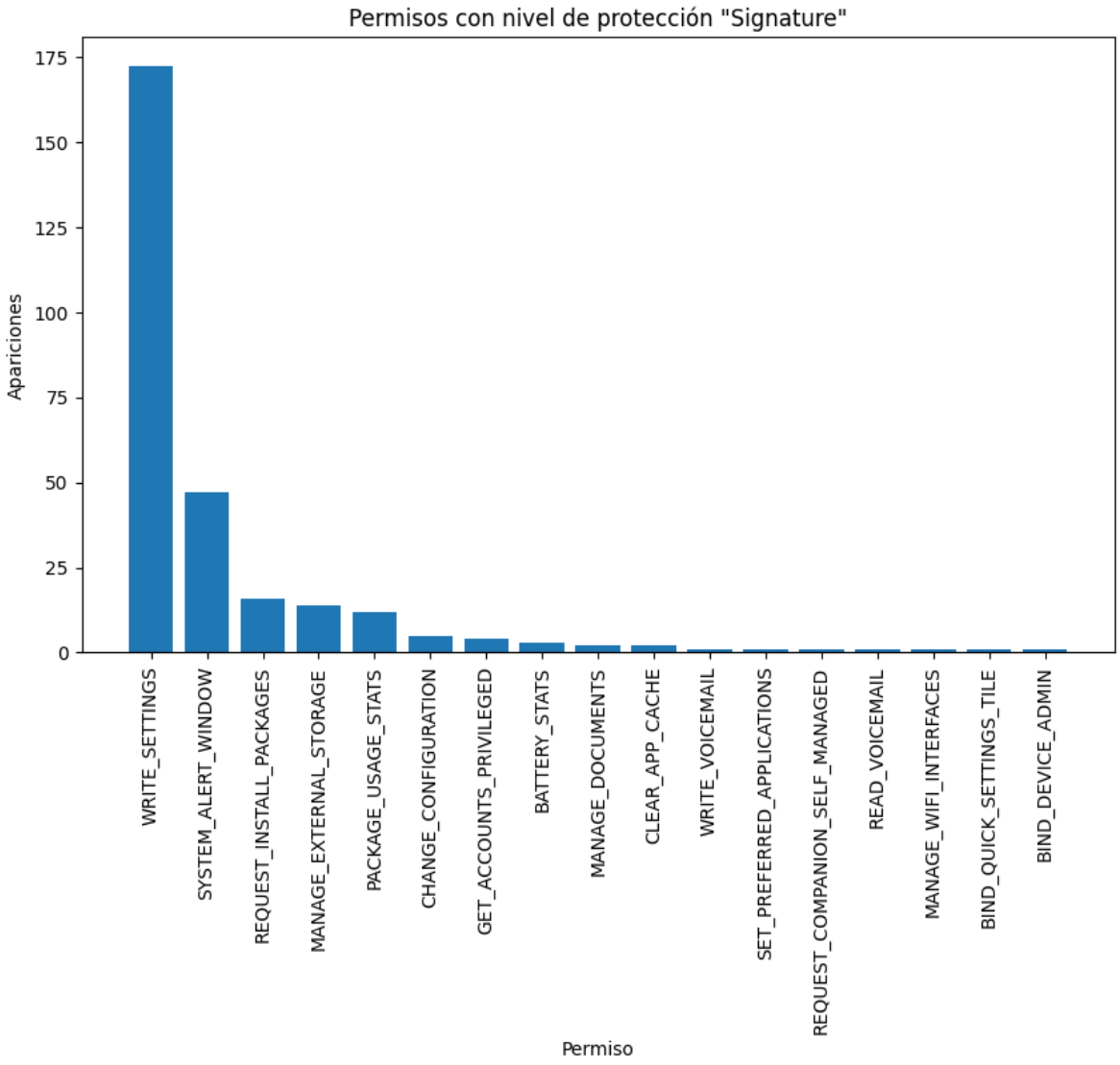


Figura 31: Permisos más comunes de categoría *Signature*

Fuente: Elaboración Propia

Los permisos *Signature* son aquellos que son otorgados a la aplicación al momento de ser instalada por el usuario, el permiso más común es WRITE_SETTINGS, el cual tiene sentido dado que su funcionalidad es permitir a la aplicación leer o escribir en la configuración del sistema lo que podría ser un paso necesario para la instalación de la aplicación, pero esta misma funcionalidad puede ser utilizada para la modificación de los ajustes del sistema y

la instalación de aplicaciones maliciosas.

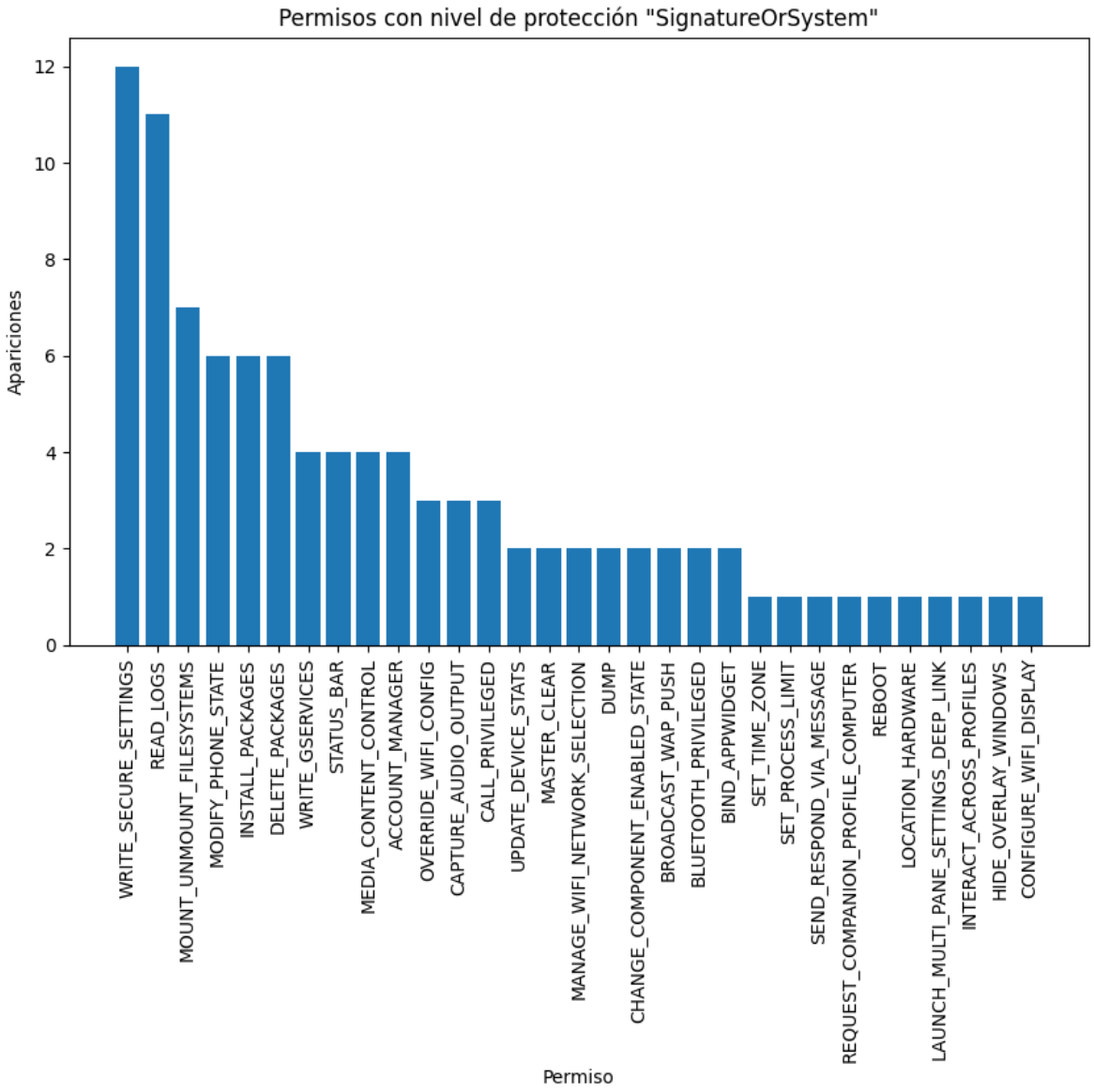


Figura 32: Utilización de permisos de protección *SignatureOrSystem*

Fuente: Elaboración Propia

En el caso de *SignatureOrSystem,* el más utilizado es WRITE_SECURE_SETTINGS que cumple una función similar a WRITE_SETTINGS pero en este caso deben ser aplicaciones dedicadas en una carpeta Android, pero puede dar lugar a modificaciones no autorizadas de configuraciones de seguridad. Algo que destacar es que la utilización de estos permisos

es inferior al resto, dado que la mayoría de las aplicaciones tienen acceso suficiente a permisos mediante la declaración de permisos *Signature.*

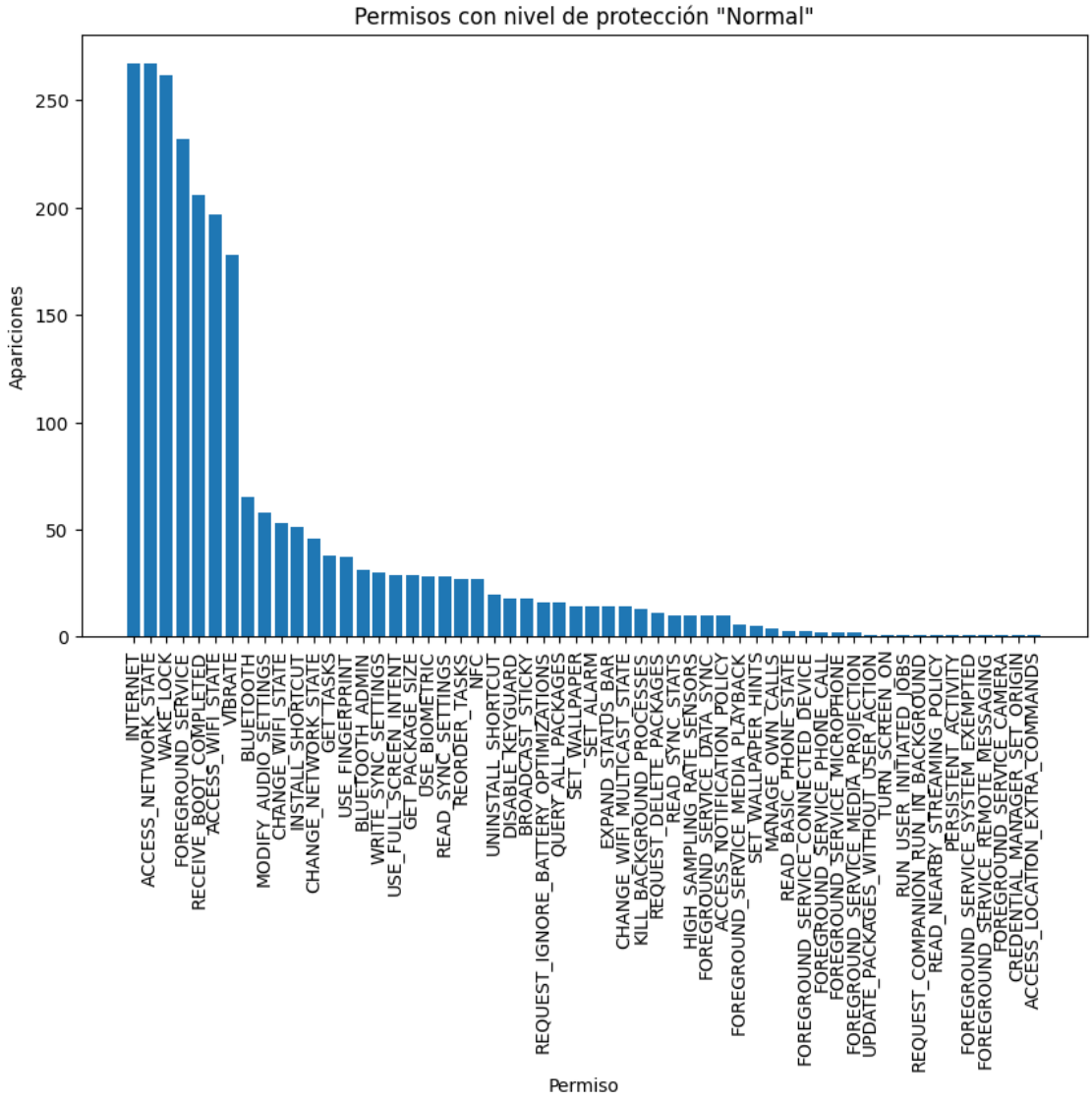


Figura 33: Permisos más comunes de nivel de protección Normal

Fuente: Elaboración propia

Los permisos de tipo Normal no representan un riesgo importante para el sistema, dado que se le dan funciones aisladas del sistema a la aplicación. Los dos más populares son INTERNET y ACCESS_NETWORK_STATE que están relacionados al acceso de la aplicación a internet como a la red doméstica de internet, como puede ser el Wi-fi del hogar. Por otro

lado WAKE_LOCK también está entre los más utilizados dado que este evita que la pantalla se bloquee o se oscurezca mientras la aplicación está en uso, lo que podría ser considerado un comportamiento esperado para las aplicaciones.

5.1.4 Resultados de análisis mediante inteligencias artificiales de procesamiento de texto

Para la técnica que utiliza inteligencia artificial para realizar el análisis de las aplicaciones se vuelve difícil poder procesar resultados, por diversas limitaciones que la implementación que permite obtener el código fuente posee, las limitaciones que las mismas inteligencias artificiales poseen tanto en su capa gratuita como en sus suscripciones y el alto factor que juega el usuario al pasar el código obtenido a la inteligencia artificial con el fin de obtener un resultado.

En primer lugar se poseen las limitaciones del sistema implementado, donde al obtener las clases en común o clases distintas entre 2 APK's se debe poner un límite de clases a agregar en el listado o el espacio de memoria usado por los distintos reportes aumenta considerablemente, dificultando o imposibilitando a FASTApi la capacidad de procesarlos, como también llenando el espacio de memoria del contenedor Docker.

En segundo lugar, las tecnologías de inteligencia artificial tienen un límite de caracteres por consulta, por lo que si el threshold explicado en el primer punto no existiera, no sería posible de todas maneras el crear un prompt capaz de procesar todo el código obtenido de las clases. Una de las posibilidades que se plantea es la posibilidad de ingresar el código mediante múltiples consultas, pero esto genera en la inteligencia artificial problemas para el procesado debido a la falta de contexto en el código otorgado por las distintas consultas.

Por último el hecho de que es la dependencia de que el usuario sea quien haga las consultas a la inteligencia artificial es un factor considerable, dado que dependiendo de la consulta que realice los resultados varían considerablemente a pesar de ser pasado el mismo código, como también hace imposible la recopilación de información respecto al rendimiento como puede ser los tiempos de ejecución para obtener un resultado, dado que se realiza en un sistema externo.

A partir de estos puntos, es importante considerar que la técnica en sí, no es inviable y que podría ser optimizada de diversas formas para obtener mejores resultados, entre estas optimizaciones se podría considerar la opción de automatizar las consultas a pesar de que siga existiendo la limitación por caracteres. Otro punto que podría verse mejorado, es el implementar la técnica fuera de un contenedor y en una API, dando la opción de acceder a todos los recursos de la máquina donde es ejecutado. Estas limitaciones causan que la idea de poder procesar código fuente mediante inteligencias artificiales no otorgue mucha información de valor, pero sí la sensación de ser una idea emergente que podría lograr

mejores resultados que los obtenidos.

5.2 Análisis de técnicas según su rendimiento

5.2.1 Análisis de tiempos de ejecución

Respecto a los tiempos de ejecución de las técnicas, se concluye que la técnica de análisis sintáctico es la más lenta de las tres, dado que recorre las aplicaciones línea por línea lo que genera un proceso bastante lento para realizar el análisis. Posteriormente viene el análisis mediante grafos que fue relativamente más rápido, pero una detalle importante a considerar es que la técnica se vuelve más lenta a medida que aumentan los grafos almacenados en la base de datos con los que debe ser comparada la aplicación a evaluar, por lo que es posible que la comparación de grafos en órdenes de magnitud mayores genere tiempos de ejecución más largos. Finalmente la técnica más rápida de ejecutar es la de análisis de permisos, lo que es dado que los permisos son identificados y su información se obtiene desde un catálogo ya hecho, por lo que no hay información que deba ser procesada sino que solo obtenida.

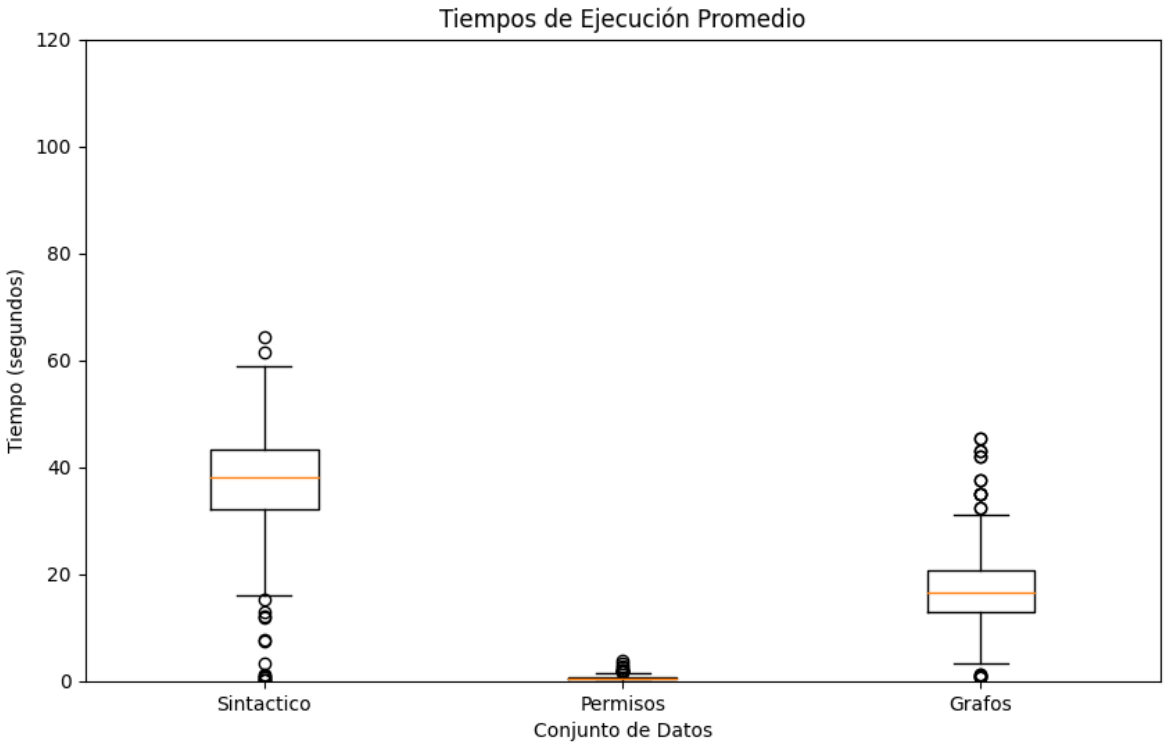


Figura 34: Tiempos de ejecución promedios por técnica.

Fuente: Elaboración propia.

Algunas consideraciones que deben ser tomadas desde el gráfico es que los tiempos de ejecución que tienden a cero en el análisis sintáctico y en el de grafos es principalmente causado por aplicaciones que no pudieron ser analizadas por las técnicas, por lo que generaron reportes vacíos con solamente el nombre y la categoría de la aplicación, pero que de todas formas fueron analizados dado que el tiempo entre que la técnica obtenía el APK y reconoció que no era una aplicación analizable también nos da información respecto a cuánto tiempo le toma a la técnica reconocer este problema.

5.2.2 Análisis de aplicabilidad de las técnicas

Para el análisis, se revisó específicamente 300 aplicaciones. De las cuales no todas pudieron ser analizadas por las técnicas implementadas como se mencionó anteriormente, a continuación se presenta la cantidad de reportes que pudieron ser generados por cada técnica.

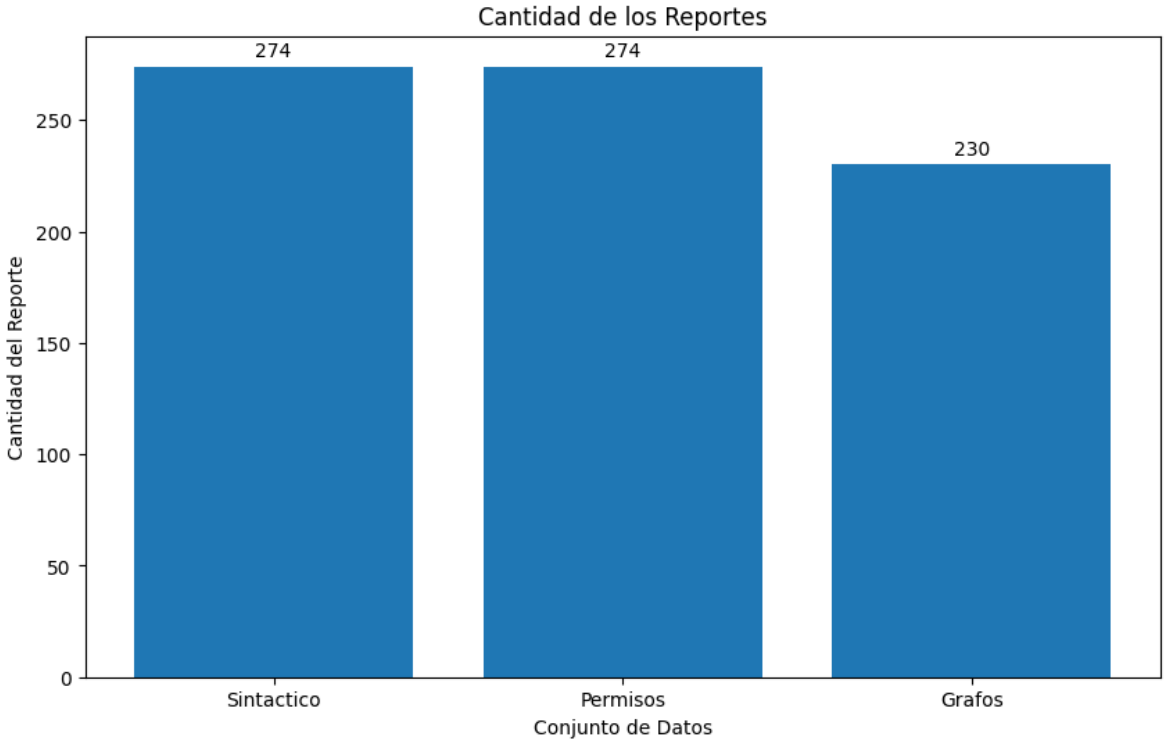


Figura 35: Cantidad de reportes analizados por técnica.

Fuente: Elaboración propia.

Como se puede ver, la técnica de análisis sintáctico y de permisos fueron capaces de

analizar la misma cantidad de aplicaciones del conjunto, mientras que la de grafos fue incapaz de analizar 70 aplicaciones. Entre los principales problemas que se encontraron al momento de analizar fueron aplicaciones que realmente estaban con extensión XAPK en lugar de APK, debido al peso que estas tienen como ocurrió con algunos juegos, y principalmente aplicaciones desarrolladas por empresas como Samsung para los teléfonos que ellos mismos construyen, lo que generaba errores con AndroGuard debido a la arquitectura para desempaquetar y la diferencia de bits para generar la traducción a código Smali.

En el caso específico de la técnica de grafos uno de los factores que más influyó en la cantidad de reportes generados fue la ejecución de la técnica en un entorno *dockerizado,* dado que esto dio lugar a que aplicaciones muy pesadas generan grafos que no eran soportados por la cantidad de recursos de la API y del Docker, lo que dio lugar a una mayor cantidad de errores al momento de aplicar la técnica implementada a un conjunto de datos.

5.2.3 Análisis del nivel de automatización de la técnica

Respecto a los niveles de automatización de cada una de las técnicas se debe recalcar que ninguna de las técnicas es 100% automática y todas requieren un intervención por parte de quien implementa la técnica.

La técnicas de análisis sintáctico y análisis de permisos requieren niveles similares de intervención dado que ambas necesitan que se elaboran catálogos o expresiones para la obtención de información, en el caso del análisis sintáctico es para la elaboración de las expresiones para el reconocimiento de vulnerabilidades, mientras que en el de permisos es en el desarrollo del catálogo con la información que será posteriormente entregada a los usuarios.

En el análisis por grafos la técnica en sí se encarga de la creación y análisis de los grafos, pero requiere que el investigador genere todos los grafos que serán almacenados en la base de datos de manera previa a los análisis con el fin de que este sea capaz de otorgar información, por lo que una fase crucial de la técnica requiere intervención humana en sus fases iniciales.

Finalmente, la técnica que analiza mediante inteligencia artificial es completamente dependiente del usuario, dado que este debe obligatoriamente pasar el código obtenido por la API a alguna inteligencia artificial capaz de procesar texto mediante consola.

5.3 Análisis de la información otorgada por cada técnica

Respecto a la información que cada técnica otorga se puede ver que esta varía considerablemente entre las distintas técnicas.

La técnica de análisis sintáctico nos da más información respecto a posibles vulnerabilidades o malas prácticas que pueden estar presente en el código Smali de la aplicación, pero debido a que el análisis es linea por linea es incapaz de darnos información como la posibilidad de que la aplicación sea un malware dado que esta es más desde un punto de vista de la aplicación y su comportamiento como un todo en lugar de instrucciones carentes de contexto.

La técnica de análisis mediante grafos en este caso específico mediante el árbol de características y la similitud con otras aplicaciones o malware conocido es capaz de darnos una idea más general del comportamiento de la aplicación en base a qué tan parecida es a otras, sirve sustancialmente para identificar posibles aplicaciones con un comportamiento sospechoso, para identificar aplicaciones que pudieron haber sido desarrolladas por el mismo equipo de desarrollo o incluso para ver si la misma versión de una aplicación descargada de distintos *mirrors* de aplicaciones presentan diferencias en su código, siendo el principal valor que esta técnica otorga la capacidad de identificar APK's que podrían ser de interés para analizar más a fondo.

La técnica de análisis de permisos otorga información relevante que el *AndroidManifest* no nos da y que en el caso de buscar de manera independiente tomaría un mayor tiempo, por lo que el principal aporte de información que esta técnica nos otorga es el procesamiento automático de los permisos que la aplicación ha manifestado e información como el nivel de riesgo que significa darle acceso a los recursos especificados a la aplicación, como también una breve descripción de lo que hace el permiso, lo que nos podría indicar si tiene sentido que el permiso está presente según la categoría de la aplicación.

Finalmente, la técnica de análisis mediante inteligencias artificiales tiene como principal objetivo el facilitar el análisis de los reportes para un usuario sin conocimientos técnicos y dar una explicación más contextual al usuario de lo que está haciendo la aplicación en sí y qué significa la información encontrada en las otras técnicas. Siendo información bastante interesante si se toma en consideración que la API tiene como objetivo la disponibilización de los datos para informar a los usuarios respecto a posibles vulnerabilidades, malas prácticas o comportamiento sospechoso en aplicaciones que pueden ser de su interés.

5.4 Análisis de la información al combinar técnicas

Finalmente al ver la diferencia considerable en la información que cada técnica era capaz de otorgar acerca de la misma aplicación es que se consideró la opción de mezclar técnicas con el fin de obtener reportes más completos y de una manera más eficiente a que si aplicamos los tres por separado a la misma aplicación.

Esta propuesta es la que se implementó en cierta medida en la técnica de análisis mediante inteligencias artificiales, donde en el primer paso se aplica un análisis de grafos a

la aplicación en cuestión para verificar si esta tiene un comportamiento similar alguna aplicación que posea malware dentro de ella, a partir de este análisis podemos identificar aplicaciones en común y aplicaciones distintas lo que nos permite optimizar el análisis estático dado que en lugar de analizar todo el código se puede hacer un filtrado y analizar clases en específico, para finalmente obtener el código de las clases en común y ver qué vulnerabilidades se encuentran en estas o como se realiza en el análisis mediante inteligencias artificiales se pasan a alguna herramienta como lo puede ser Chat GPT.

Con el análisis combinado nombrado anteriormente seríamos capaces de obtener un análisis de comportamiento de la aplicación que podría indicarnos presencia de malware a como también de código específico que podría dar origen a vulnerabilidades, lo cual puede ser complementado con un análisis de permisos debido a que el rendimiento de este no impacta mayormente al tiempo de generación y nos otorga información de mayor relevancia sobre la aplicación.

5.5 Evaluación del pipeline de análisis

Respecto al *pipeline* de trabajo utilizado se pueden realizar distintos análisis paso por paso.

1. **Descargar el APK:** Este paso presento una de las más grandes dificultades en cuanto a la implementación de las técnicas, el cual es que la *Play Store* de Android no permite la automatización de descargas mediante *bots.* Por lo que la obtención de aplicaciones fue mediante *marketplaces* de terceros, lo cual presenta el problema de que nada asegura que las APKs obtenidas sean las originales.
2. **APK es Obtenido:** En este caso a través de los *marketplaces* de aplicaciones fue posible obtener el 100% de las aplicaciones seleccionadas, por lo que respecto a disponibilidad de aplicaciones no hubo problemas.
3. **Procesamiento del APK:** Con el fin de procesar el APK y obtener información a partir de este es que se utilizó la librería Androguard en Python, siendo su aplicabilidad de un 91% siendo aplicable a 273 de 300 aplicaciones. La principal problemática que se encontró fueron limitaciones de la librería en sí y la arquitectura de 8 bits que utilizaba para la extracción de información cuando se encontraba con arquitecturas distintas como ocurría con aplicaciones de Samsung.
4. **Separación del APK:** Este proceso era realizado de fondo por Androguard, por lo que su aplicabilidad y problemas son los mismos.
5. **Obtención del Código:** En cuanto a la obtención del código debido a las limitaciones que se tuvo por trabajar en un contenedor y en específico a través de una API de Python, es que solo se trabajo con el código Smali que Androguard otorgaba por lo que en cuanto a la obtención del código esto fue una limitación importante. Por otro lado, su aplicabilidad y dificultades son las mismas que los 2 pasos anteriores por la utilización de la misma librería.

6. **Depuración del código obtenido:** En cuanto a la depuración del código, esto fue utilizado principalmente en la técnica que analizaba mediante inteligencias artificiales ya que se otorgaba solamente el código de interes ya sea el comun o el distinto en el grafo de características, por otro lado en el resto de técnicas este paso no fue aplicado dado que se analizaba todo el código en el análisis sintáctico o se generaba un árbol completo en el análisis por grafos.
7. **Aplicación de técnicas de análisis estático:** Este paso depende de cada técnica pero tanto el análisis sintáctico como el de permisos tienen una aplicabilidad del 91%, mientras que el de grafos tenía una aplicabilidad del 76%. Siendo la principal limitación la utilización de Python como lenguaje, que sea ejecutado en un contenedor y los problemas que presentaba Androguard.
8. **Análisis de resultados:** En todos los casos fue posible obtener resultados que otorgan valor y donde era posible realizar algún tipo de análisis, por lo que este paso fue aplicable al 100% de las técnicas.

5.6 Discusión de Resultados

En el siguiente capítulo se discuten las decisiones respecto a la implementación de las técnicas, como también de los hallazgos que fueron encontrados al analizar resultados, como también analizar la validez de estos.

En primer lugar, respecto a la implementación uno de los factores más limitantes para el conjunto de técnicas fue el hecho de desarrollar el sistema en una API dentro de un contenedor, lo que limitó el uso de tecnologías a solamente Python por la imagen utilizadas, como también las limitaciones de recursos que el contenedor posee como memoria RAM o disco duro, a lo que se sumaron las restricciones que FastAPI de por sí posee respecto a rendimiento y cantidad de información que era capaz de procesar. Considerando estas limitaciones se podría considerar que el trabajar en un entorno donde cada código es ejecutado en la máquina directamente y a través de la consola podría dar origen a resultados de mayor calidad respecto a información al no tener que trabajar necesariamente con el código Smali y al mismo tiempo podrían tener un mejor rendimiento al ser capaz de acceder a todos los recursos del equipo en el que sean ejecutados.

Posteriormente respecto a las implementaciones y resultados obtenidos el principal punto a considerar es la dependencia de los resultados de la calidad de información que le provee quien implementa las técnicas, en el caso del análisis sintáctico se pueden obtener resultados notoriamente distintos según las expresiones utilizadas para identificar las vulnerabilidades, en el análisis de permisos depende completamente del catálogo de permisos generado por el investigador, en la técnica de grafos si no se poseen grafos de malware dentro de la base de datos el reporte no será capaz de identificar ningún

comportamiento sospechoso y para la técnica de análisis mediante inteligencia artificial depende completamente del *prompt* que se le pase a la inteligencia artificial. Cada uno de estos factores humanos generará reportes de distintas calidad para las mismas técnicas por lo que los resultados que se obtuvieron pueden variar notoriamente al replicar el experimento y generar modificaciones en los puntos nombrados anteriormente, por lo que esta posible diferencia de los resultados podría generar que al implementar las mismas técnicas por otra persona los resultados obtenidos por los experimentos nombrados anteriormente pierdan algún tipo de validez.

CAPÍTULO 6: CONCLUSIÓN

Una vez finalizado el trabajo es posible discutir respecto al cumplimiento de los objetivos, alcances y limitaciones de la solución planteada, calidad de los resultados que este otorga, conocimientos que fueron adquiridos en el desarrollo de este y trabajo a futuro que puede ser desarrollado a partir de lo realizado.

6.1 Impacto y limitaciones de la solución propuesta

Este trabajo responde a una necesidad de transparentar las prácticas de código, en el contexto de las aplicaciones móviles ante la masividad de estas y la información sensible que manejan de los usuarios que las consumen. Esto debido a la masificación del sistema operativo Android en la población teniendo aproximadamente 2.000 millones de usuarios activos quienes tienen acceso a un catálogo de 2.5 millones de aplicaciones[39]. A continuación se plantean el cómo se abordaron las principales problemáticas que existen en el contexto de las aplicaciones móviles y el cómo la solución propuesta las aborda.

- **Ausencia de un estándar para realizar análisis de código estático:** Debido a la implementación de técnicas diametralmente distintas fue necesaria la elaboración de una metodología de trabajo que funcionara de manera transversal, la cual facilitará la implementación y obtención de resultados independiente de la técnica, siendo esta lograda en el *pipeline* presentado en la sección 3.1.
- **Identificación de malas prácticas y vulnerabilidades:** El sistema fue capaz de identificar de manera exitosa las malas prácticas y vulnerabilidades dentro del código fuente de la aplicación, a través de la utilización de un catálogo de instrucciones sospechosas generado a partir del listado de vulnerabilidades de la CWE.
- **Identificación de malware:** El sistema fue capaz de analizar aplicaciones y obtener porcentajes de similitud con otras aplicaciones, lo cual puede ser aplicado de manera que se puede comparar cualquier aplicación con un malware y obtener un indicador de si las estructura de esta posee similitudes con código maliciosos, que puede dar como resultado la aplicación de un análisis más en profundidad.
- **Disponibilización de la información para los usuarios:** La solución propuesta abarcaba también la necesidad de dar a conocer los comportamientos sospechoso, malas prácticas o posibles fugas de información a los usuarios que presentará un interés por esta información, para esto el sistema fue desarrollado en una API que al ser disponibilizada puede ser accedida por los usuarios que presentan un interés por ver los reportes de las diversas aplicaciones o para la utilización de terceros

[39] https://www.businessofapps.com/data/android-statistics/

que deseen implementar los análisis en sus propios desarrollos.

Por otro lado, las limitaciones que presentó la solución propuesta están vinculadas a las tecnologías seleccionadas para el desarrollo de esta y los requisitos que eran solicitados para la implementación. Las principales limitaciones fueron:

- **Contenedores:** El hecho de que el sistema fuera levantado dentro de un contenedor de Docker limitó el acceso a recursos del equipo como disco duro o memoria RAM, lo que limitó la capacidad del sistema de procesar información de la aplicación como se planteó en la sección 5.6.
- **Desarrollo de una API:** Otra limitante en cuanto a recursos fue la disponibilización del sistema en una API de FastAPI, dado que en el caso que los reportes excedieran los límites de caracteres o de memoria permitidos por el framework, este dejaba de funcionar o devolvió mensajes de error cuando la aplicación había sido analizada con éxito.
- **Lenguajes de Programación:** La utilización de Python para la implementación del sistema limitó las librerías a utilizar para la obtención del código fuente de las APK's, esto causó que la principal librería a utilizar fuera Androguard lo que generó por su parte la limitante a realizar solo análisis en Smali, el cual puede dar resultados distintos a si se hubiera analizado Java.

6.2 Objetivos

6.2.1 Objetivo General

El objetivo general del trabajo realizado era la implementación de distintas técnicas de análisis de código estático para analizar diversas aplicaciones que estén disponibles para Android, con el fin de analizar la calidad de información, rendimiento y aplicabilidad de las distintas técnicas

El cual fue logrado exitosamente en el desarrollo de este documento a través de la creación de una metodología de trabajo aplicable de manera transversal para las distintas técnicas, implementación de estas y análisis de los resultados obtenidos con el fin de comparar las distintas métricas a evaluar de cada técnica.

6.2.2 Objetivos Específicos

Los 6 objetivos específicos planteados en este documento fueron cumplidos exitosamente.

- **Identificar las vulnerabilidades a analizar a partir de la literatura existente:** A partir del estudio de la literatura existente se definió que las vulnerabilidades a analizar corresponden a las del top 25 del CWE y algunas más asociadas

específicamente al desarrollo de software, la información respecto a los criterios de selección pueden ser vistos en la sección 3.5.

- **Definir los parámetros de inclusión/exclusión que serán utilizados para seleccionar las técnicas de análisis de código estático a implementar:** Respecto al análisis de técnicas el primer paso fue el estudio y categorización de diversos *frameworks* o técnicas utilizados en la literatura para el análisis de código estático, el cual fue realizado en la sección 2.10, posteriormente en la sección 3.3 se definieron exitosamente las métricas para la selección de técnicas, como lo son la aplicabilidad de estas y facilidad en su implementación.
- **Listar los parámetros de inclusión/exclusión que serán utilizados para elegir las aplicaciones que serán analizadas:** Este objetivo fue logrado en la sección 3.6, donde se presentan las métricas seleccionadas como lo son la popularidad de la aplicación, categoría de esta, clasificación, número de actualizaciones y peso de la aplicación.
- **Esquematizar un *pipeline* de análisis que permita obtener el código fuente de las aplicaciones a partir de su APK:** Este objetivo fue logrado a través del *pipeline* presentado en la sección 3.1, el cual fue aplicado en la implementación de las distintas técnicas seleccionadas por lo que podría considerarse como una metodología que es aplicable de manera transversal a las técnicas.
- **Implementar las técnicas seleccionadas en un conjunto de aplicaciones seleccionadas a partir de ciertos criterios de inclusión/exclusión:** Este objetivo fue logrado con éxito a través del capítulo 4 en el cual se plantea en profundidad como fueron desarrolladas cada una de las técnicas, mientras que en el capítulo 5 se pueden ver los resultados obtenidos al ser aplicadas en el conjunto de aplicaciones seleccionadas en la sección 3.6.
- **Realizar un análisis de la información que se obtenga con las distintas técnicas y comparar la calidad de estas, las variables que se vieron involucradas en la calidad de las técnicas y la aplicabilidad de estas, así como identificar las brechas de seguridad y malas prácticas más comunes en aplicaciones móviles:** Respecto al análisis y comparación de las diversas técnicas se puede ver que fue logrado con éxito en el capítulo 5 en donde se valida la solución, viendo las diferencias entre la información que cada técnica provee, rendimiento de estas y como las distintas técnicas pueden ser utilizadas en conjunto para complementar la información, con el fin de obtener un reporte más completo.

6.3 Resultados obtenidos

Respecto a los resultados obtenidos por técnicas en el caso del análisis sintáctico se pudo encontrar que la vulnerabilidad más encontrada al realizar un análisis de instrucciones linea por linea son las asociadas a la CWE-319 de Transmisión de información sensible a través de texto, lo que quiere decir que muchas de las conexiones a bases de datos o servicios de internet pueden ser interferidos por terceros y darles la capacidad de obtener información sensible de los usuarios; en cuanto a la técnica de análisis mediante grafos se encontró que la información que el árbol de características otorga es bastante similar al obtenido en el estudio realizado por [QiLi2015] donde se utilizaba para identificar a que familia de *malware* se acerca un APK desconocido, lo cual fue utilizado en nuestro caso para identificar si el APK analizada tiene alguna semejanza con algún desarrollador ya identificado lo que nos podría permitir identificar si el APK es de origen de algún desarrollador que ha sido identificado malicioso; en cuanto al análisis de permisos se logro obtener los permisos más comunes en las aplicaciones y asociarlos a las distintas categorias, lo que permitiria identificar si alguna de las aplicaciones tiene permisos que no son comunes en su categoria y por lo tanto podria dar señales de un comportamiento sospechos; finalmente el método propuesto de análisis estático mediante inteligencias artificiales de procesamiento de texto presento resultados prometedores pero que no eran suficientes para realizar un análisis debido a las distintas limitaciones que los procesadores poseen como lo es el número de caracteres, como también las limitaciones propias del sistema como la imposibilidad de obtener una gran cantidad de código sin que genere problemas en la API por lo que existia la posibilidad de que las inteligencias artificiales carecieran de contexto para realizar un analisis correcto del código.

Finalmente, respecto a la información que cada una de estas otorga se puede ver que la técnica de análisis sintáctico es capaz de identificar posibles vulnerabilidades en el código mediante instrucciones susceptibles previamente definidas, por lo que otorga información más específica y no un enfoque global de la aplicación, que es lo que sí otorga el análisis por grafos de características que es capaz de dar una visión más global de la aplicación desde donde se puede deducir si tiene o no un comportamiento sospechoso, por lo que la combinación de ambas al obtener las clases que podrían ser problemáticas en el análisis por grafos y analizar estas clases en específico en búsqueda de vulnerabilidades con analisis sintactico ha demostrado dar una imagen más completa de la aplicación analizada. Por otro lado el análisis de permisos da información distinta a las 2 técnicas anteriores dado que nos permite identificar riesgos a los que accedemos como usuarios y al mismo tiempo identificar si alguna aplicación podría poseer un comportamiento sospechoso si es que se desvía de los permisos comunes para su categoría.

Lo que es importante de recalcar de estos resultados es que las técnicas no permiten asegurar con certeza que una aplicación posee o no un comportamiento de malware, sino que da indicios de que aplicaciones tienen características más sospechosas dado que para

asegurar que una aplicación es un malware es necesario un análisis más contextual de la información que posee con el fin de diferenciar una aplicación que es maliciosa intencionalmente de una que posee un funcionamiento negligente debido a malas prácticas en su desarrollo.

6.4 Conocimientos adquiridos en este trabajo

El desarrollo de este trabajo requirió una investigación en diversas áreas del desarrollo de software para aplicaciones móviles, posibles vulnerabilidades que pueden surgir a partir de este, malas prácticas en el desarrollo y malware del que pueden ser víctimas, tanto desarrolladores como usuarios.

En cuanto al desarrollo de aplicaciones móviles fue necesario el comprender como funcionaba el sistema operativo Android, las capas que este posee para proveer seguridad al kernel del sistema, la modalidad con la cual se le conceden permisos a las aplicaciones y como cada aplicación corre de manera aislada en el sistema para evitar fugas de información entre aplicaciones. Por otro lado, el entender las diferencias entre las distintas formas de desarrollar aplicaciones, ya sean nativas, *webapp* o híbridas, es un gran aporte a mi formación dado que me permite definir los pros y los contras de cada una, como también el entender las políticas de permisos que el *marketplace* de Android posee.

Respecto a la ciberseguridad el conocimiento adquirido respecto a las diferencias entre vulnerabilidad, malas prácticas y malware, es bastante interesante dado que estas se encuentran estrechamente asociadas, siendo fácilmente confundibles. Mientras que uno de los descubrimientos que más me llamó la atención fue la existencia de la ofuscación y las distintas técnicas que existen de esta, dado que no era una práctica de la cual poseía conocimiento a pesar de que es un proceso clave para aumentar la seguridad de las aplicaciones y de la propiedad intelectual de estas, en cuanto a código.

6.5 Trabajo Futuro

Dentro del trabajo desarrollado en el documento es importante destacar el trabajo o nuevos enfoques que se deberían desarrollar en trabajos futuros. La primera consideración que se debe tomar en cuenta es las limitaciones que este trabajo tuvo por la necesidad de que el sistema funcionara en una API, por lo que se debería evaluar como el implementar las mismas técnicas como código ejecutable mediante consola afecta los resultados obtenidos o la utilización de otros lenguajes como podría ser con Java. También es necesario ver como la modificación de los factores humanos como las expresiones utilizadas en el análisis sintáctico o el catálogo de permisos pueden ser mejorados para hacerlos independientes del conocimiento del investigador o estudiar el impacto que estos tienen en los resultados obtenidos al analizar como modificaciones de estos podrían dar resultados crucialmente distintos.

Otro factor a considerar que podría ser de interés es aumentar la muestra de aplicaciones a analizar lo que nos podría dar resultados distintos si los criterios de selección varían, como podrían ser tomar una muestra más similar de categorías, aplicaciones con tamaños similares o algún otro criterio que podría ser de interés para estudios específicos.

REFERENCIAS BIBLIOGRÁFICAS

[HuiXu2015] H. Xu, Y. Zhou, C. Gao, Y. Kang and M. R. Lyu, "SpyAware: Investigating the privacy leakage signatures in app execution traces," 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), 2015, pp. 348-358, doi: 10.1109/ISSRE.2015.7381828.

[WenyunDai2017] W. Dai, L. Chen, M. Qiu, A. Wu and M. Liu, "DASS: A Web-Based Fine-Grained Data Access System for Smartphones," 2017 IEEE International Conference on Smart Cloud (SmartCloud), 2017, pp. 238-243, doi: 10.1109/SmartCloud.2017.45.

[GokhanKul2018] G. Kul, S. Upadhyaya and V. Chandola, "Detecting Data Leakage from Databases on Android Apps with Concept Drift," 2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), 2018, pp. 905-913, doi: 10.1109/TrustCom/BigDataSE.2018.00129.

[QiLi2015] Q. Li and X. Li, "Android Malware Detection Based on Static Analysis of Characteristic Tree," 2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, 2015, pp. 84-91, doi: 10.1109/CyberC.2015.88.

[YuZhang2020] Y. Zhang and B. Li, "Malicious Code Detection Based on Code Semantic Features," in IEEE Access, vol. 8, pp. 176728-176737, 2020, doi: 10.1109/ACCESS.2020.3026052.

[AsafShabtai2010] A. Shabtai, Y. Fledel and Y. Elovici, "Automated Static Code Analysis for Classifying Android Applications Using Machine Learning," 2010 International Conference on Computational Intelligence and Security, 2010, pp. 329-333, doi: 10.1109/CIS.2010.77.

[AidenPolese2022] A. Polese, S. Hassan and Y. Tian, "Adoption of Third-party Libraries in Mobile Apps: A Case Study on Open-source Android Applications," 2022 IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MobileSoft), 2022, pp. 125-135, doi: 10.1145/3524613.3527810.

[HsuMyatWin2022] H. M. Win, "Complement of Dynamic Slicing for Android Applications with Def-Use Analysis for Application Resources," 2022 IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MobileSoft), 2022, pp. 100-101, doi: 10.1145/3524613.3527808.

[IbrahimSalihu2018] I. A. Salihu, R. Ibrahim and A. Usman, "A Static-dynamic Approach for UI Model Generation for Mobile Applications," 2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), 2018, pp. 96-100, doi: 10.1109/ICRITO.2018.8748410.

[KristiinaRahkema2021] K. Rahkema and D. Pfahl, "GraphifyEvolution - A Modular Approach to Analysing Source Code Histories", 2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft), 2021, pp. 24-27, doi: 10.1109/MobileSoft52590.2021.00009.

[KristiinaRahkema2022] K. Rahkema and D. Pfahl, "SwiftDependencyChecker: Detecting Vulnerable Dependencies Declared Through CocoaPods, Carthage and Swift PM," 2022 IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MobileSoft), 2022, pp. 107-111, doi: 10.1145/3524613.3527806.

[UmmeMannan2016] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig and C. Jensen, "Understanding Code Smells in Android Applications," 2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), 2016, pp. 225-236, doi: 10.1145/2897073.2897094.

[KewelShah2019] K. Shah, H. Sinha and P. Mishra, "Analysis of Cross-Platform Mobile App Development Tools," 2019 IEEE 5th International Conference for Convergence in Technology (I2CT), 2019, pp. 1-7, doi: 10.1109/I2CT45611.2019.9033872.

[RojasPoblete2016] Rojas Poblete, Cristián. “Evaluación de la seguridad de aplicaciones móviles bancarias.”, 2016., Santiago, Chile: Universidad de Chile - Facultad de Ciencias Físicas y Matemáticas. Disponible en https://repositorio.uchile.cl/handle/2250/144529

[AnirbanSarkar2019] A. Sarkar, A. Goyal, D. Hicks, D. Sarkar and S. Hazra, "Android Application Development: A Brief Overview of Android Platforms and Evolution of Security Systems," 2019 Third International conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 2019, pp. 73-79, doi: 10.1109/I-SMAC47947.2019.9032440.

[AmukelaniNgobeni2019] A. Ngobeni and S. Mhlongo, "Towards Enhancing Security in Android Operating Systems – Android Permissions & User Unawareness," 2019 2nd International Conference on Computer Applications & Information Security (ICCAIS), 2019, pp. 1-6, doi: 10.1109/CAIS.2019.8769531.

[ZhengyangQu2014] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14). Association for Computing Machinery, New York, NY, USA, 1354–1365. https://doi.org/10.1145/2660267.2660287

[Uma2019]K. V. Uma and E. S. Blessie, "Survey on Android Malware Detection and Protection using Data Mining Algorithms," 2018 2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 2018 2nd International Conference on, 2018, pp. 209-212, doi: 10.1109/I-SMAC.2018.8653720.

[Geunha2022] G. You, G. Kim, S. Han, M. Park and S. -J. Cho, "Deoptfuscator: Defeating Advanced Control-Flow Obfuscation Using Android Runtime (ART)," in IEEE Access, vol. 10, pp. 61426-61440, 2022, doi: 10.1109/ACCESS.2022.3181373.

[Dharmalingam2022] B. Dharmalingam, A. Liu, S. Ganesan and S. Roy, "FineObfuscator: Defeating Reverse Engineering Attacks with Context-sensitive and Cost-efficient Obfuscation for Android Apps," 2022 IEEE International Conference on Electro Information Technology (eIT), 2022, pp. 368-374, doi: 10.1109/eIT53891.2022.9837111.

[HuiXu2017]H. Xu, Y. Zhou, Y. Kang and M. R. Lyu, "On secure and usable program obfuscation: A survey", *CoRR*, vol. abs/1710.01139, 2017, [online] Available: http://arxiv.org/abs/1710.01139.

[Arnatovich2018] Y. L. Arnatovich, L. Wang, N. M. Ngo and C. Soh, "A Comparison of Android Reverse Engineering Tools via Program Behaviors Validation Based on Intermediate Languages Transformation," in IEEE Access, vol. 6, pp. 12382-12394, 2018, doi: 10.1109/ACCESS.2018.2808340.

ANEXOS

1. Categorías de Aplicaciones

Categoría	Descripción
Action	Juegos y aplicaciones donde los usuarios participan en desafíos centrados en acción en tiempo real.
Adventure	Juegos y aplicaciones que se centran en sumergir al usuario en historias.
Arcade	Juegos con características más antiguas.
Beauty	Aplicaciones relacionadas a maquillaje, cuidado de la piel y consejos para mejorar la apariencia personal.
Books & Reference	Aplicaciones que permiten acceder a libros electrónicos o recursos en línea, como lo pueden ser publicaciones científicas.
Business	Aplicaciones de gestión empresarial, productividad y herramientas profesionales.
Casual	Juegos enfocados en sumergir al usuario en una experiencia relajante.
Communication	Aplicaciones centradas en la comunicación entre usuarios, como lo pueden ser aplicaciones de mensajería instantánea.
Education	Aplicaciones destinadas a herramientas de aprendizaje.
Educational	Juegos y aplicaciones que tienen como objetivo educar al usuario respecto a un tema.
Entertainment	Juegos y aplicaciones diseñados para entretener al usuario, pueden ir desde videos hasta música y más.
Finance	Aplicaciones enfocadas en mejorar la gestión financiera de los usuarios.
Health & Fitness	Aplicaciones que promueven estilos de vida saludable, abarcan desde rutinas de ejercicios hasta consejos alimenticios.

Maps & Navigation	Aplicaciones que permiten a los usuarios navegar y encontrar direcciones en su entorno.
Music & Audio	Aplicaciones que permiten la gestión de música.
Photography	Aplicaciones que permiten capturar, editar y mejorar fotos.
Productivity	Aplicaciones que están diseñadas para aumentar la eficiencia en el trabajo y la gestión de tareas.
Puzzle	Juegos que buscan sumergir al usuario en experiencias donde es necesaria la resolución de acertijos o completar objetivos.
Racing	Juegos principalmente de carreras de autos.
Role Playing	Juegos donde los usuarios deben asumir el rol de algún personaje dentro de estos.
Shopping	Aplicaciones que permiten a los usuarios la compra y venta de productos desde sus teléfonos.
Simulation	Juegos donde los usuarios simulan diversas experiencias que pueden ir desde gestión de ciudades hasta supervivencia.
Social	Aplicaciones que permiten al usuario conectarse con amigos o familiares, chatear o compartir fotografías.
Sports	Aplicaciones que permiten al usuario conocer de eventos deportivos, noticias o rutinas de ejercicio.
Strategy	Juegos donde el usuario requiere planificar y tomar decisiones para alcanzar la victoria.
Tools	Aplicaciones que ofrecen diversas herramientas a los usuarios como lo pueden ser calculadoras, editores de texto u otros..
Travel & Local	Aplicaciones que permiten a los usuarios la planificación de viajes, lugares de interés o recomendaciones de destinos.
Video Players	Aplicaciones que permiten reproducir y gestionar videos.

Tabla Anexa 1: Categorías de aplicaciones

Fuente: Elaboración propia