Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Un estudio empírico sobre la eficacia de las herramientas de medición de cobertura en la plataforma Android

Tesis de Licenciatura en Ciencias de la Computación

Gustavo Giráldez

Director: Iván Arcuschin Moreno

Codirector: Juan Pablo Galeotti

Buenos Aires, 2022

UN ESTUDIO EMPÍRICO SOBRE LA EFICACIA DE LAS HERRAMIENTAS DE MEDICIÓN DE COBERTURA EN LA PLATAFORMA ANDROID

Resumen

La medición de cobertura de código de caja negra es una práctica con aplicaciones en seguridad y análisis de defectos de software. Al ser de caja negra puede ser realizada por terceras partes no vinculadas con quienes lo desarrollan. Por estos motivos, los análisis y pruebas relacionadas con esta técnica resultan de interés particular en plataformas de software de consumo masivo. Se estima que el número de dispositivos móviles en la actualidad supera los 15 mil millones, y el sistema operativo Android opera en una gran proporción de estos dispositivos. Realizaremos un estudio empírico de las herramientas disponibles para realizar mediciones de cobertura en software diseñado para el sistema Android, analizando su eficacia en la modalidad de operación de caja negra y sin acceso al código fuente.

En un mercado tan dinámico como el de dispositivos móviles los avances en prestaciones y características de los mismos obligan a una mejora, evolución y revisión continua de las capacidades ofrecidas por las plataformas de software correspondientes. Las herramientas a analizar fueron en principio desarrolladas con fines académicos y tienen escaso mantenimiento. Nos interesa saber si se mantienen vigentes y pueden realizar mediciones de cobertura en software que ha sido desarrollado usando técnicas modernas y siguiendo los requerimientos actuales de la plataforma Android.

Palabras claves: Cobertura, Android, COSMO, ACVTool, Ella, instrumentación, DEX, bytecode.

Índice general

1.. Introducción 1
1.1. Motivaciones 1
1.2. Cobertura de código 1
1.3. Instrumentación de código 2
1.4. Instrumentación para análisis de cobertura 2
1.5. Granularidad de instrumentación 3
1.6. Objetivos 3

2.. Preliminares sobre Android 4
2.1. Aplicaciones en Android 4
2.2. Componentes de una aplicación Android 4
2.3. La plataforma Android 5
2.4. Bytecode y código nativo 6
2.5. Dalvik bytecode y Java bytecode 6
2.6. Formato binario DEX 7
2.7. Smali 7
2.8. Empaquetado de una aplicación Android 8
2.9. El manifiesto de una app Android 8
2.10. XML binario y recursos 9
2.11. Empaquetado de código nativo 9
2.12. Integridad de un APK 9

3.. Herramientas a analizar 11
3.1. Estrategias de instrumentación y medición 11
3.2. Ella 11
3.3. ACVTool 12
3.4. COSMO 13
3.5. Modificaciones a COSMO 14

4.. Descripción de los experimentos 16
4.1. Fases de los experimentos 16
4.2. Criterios de aceptación 16
4.3. Obtención de los APKs 16
4.4. Caracterización de las aplicaciones 17
4.5. Monkey y software de testing 18
4.6. Emulador Android 18
4.7. Versiones de Android para las pruebas 18
4.8. Validación inicial de los APKs 19

5.. Resultados 20
5.1. Resultados de la instrumentación 22
5.2. Limitaciones *a priori* en la instrumentación 23
5.3. Causas de falla en la instrumentación 24

5.4. Resultados de medición de cobertura 26
5.5. Limitaciones *a priori* en la medición de cobertura 28
5.6. Causas de falla en la medición de cobertura 28
5.7. Correlación con la caracterización de los sujetos 31

6.. Conclusiones 34
6.1. Trabajo futuro 35

Apéndice 37

A.. Listado de APKs 38

1. INTRODUCCIÓN

1.1. Motivaciones

El testeo de caja negra es una de las herramientas disponibles para el análisis de vulnerabilidades y estabilidad que se pueden realizar sobre el software disponible en el mercado. Para evaluar la efectividad de los mecanismos de testeo, una de las principales métricas a utilizar es la cobertura de código alcanzada durante la ejecución de las pruebas. Para ello es necesario instrumentar la aplicación, antes o durante la corrida de las pruebas, de tal forma de determinar qué porciones de código son ejercitadas por dichas pruebas.

La plataforma Android es una de las más importantes en el mercado de dispositivos móviles (teléfonos celulares, tablets, etc.) y es utilizada por una porción muy importante de la población mundial. Entonces, resulta de mucho interés poder realizar pruebas de caja negra sobre aplicaciones de software escritas para dicha plataforma ya que vulnerabilidades u otro tipo de problemas puede afectar potencialmente a un número muy grande de usuarios. Poder realizar testeos efectivos en la modalidad de caja negra y sin necesidad de tener acceso al código fuente del software testeado, permite que éstos sean realizados por terceras partes independientes. Esto es, no necesariamente por los autores de las aplicaciones.

1.2. Cobertura de código

La cobertura de código es una métrica que indica qué porcentaje de todo el código que constituye una aplicación es ejercitado (ie. ejecutado) durante una corrida de un experimento o una suite de tests. Es interesante medirla ya que da una idea de qué tan completa es una suite de tests o un experimento particular, y esto a su vez es importante para estimar la confianza sobre si el objetivo de la suite de tests o el experimento es alcanzado. Por ejemplo, si construimos una suite para verificar que la aplicación no tiene errores de tipo crash, cuanto mayor sea la cobertura de código medida, con mayor certeza podremos asegurar que no existe un camino de ejecución que conduzca a un crash (entiéndase por crash a un error fatal que detiene la ejecución del programa y no permite continuar con su ejecución).

Para medir la cobertura de código, necesitamos tener la capacidad de monitorear la ejecución de alguna manera para así saber qué partes del código fueron ejercitadas. Hay varias maneras de hacerlo, a saber:

1. Modificando (instrumentando) el código para que registre los lugares por los cuales se ejecuta.

2. Realizando un muestreo periódico de la CPU que ejecuta el código.

3. Ejecutando el código en un emulador o intérprete que nos permita saber en todo momento cuál es el estado del programa.

El método de muestreo periódico suele ser el más eficiente en términos de costo en tiempo de ejecución, aunque el menos preciso, pues determinar o no si cierta parte del código

fue ejecutada dependerá de la frecuencia de muestreo y de las condiciones de monitoreo en general.

En el otro extremo, ejecutar el código en un emulador o intérprete tiene un alto grado de *overhead* que puede hacer la medición impracticable o cambiar sustancialmente las condiciones de la prueba. Sobre todo si se trata de aplicaciones interactivas, ejecutarlas en un emulador puede significar una importante pérdida de interactividad que, a su vez, puede alterar los resultados de la medición.

El punto intermedio es la instrumentación, que implica modificar el código a ejecutar para que, como parte de esa ejecución, se registre qué partes se ejercitaron.

1.3. Instrumentación de código

Instrumentar código implica modificarlo de tal forma que se ejecute concurrentemente y en la misma traza el código original y algún código arbitrario insertado oportunamente (*probe*). El código agregado por la instrumentación tiene infinidad de aplicaciones. Podemos monitorear alguna métrica de la ejecución, auditar ciertas operaciones, registrar parámetros de ejecución con fines estadísticos, conectar con un *debugger*, permitir la introspección arbitraria y en tiempo real de los datos de la aplicación, modificar dinámicamente el comportamiento del programa, medir tiempo de ejecución u otros parámetros de performance, etc. Los usos de la instrumentación son múltiples y variados.

La instrumentación debe hacerse de tal forma de no alterar la semántica del programa original. Dado que los recursos disponibles durante la ejecución son escasos (eg. la CPU tiene una cantidad de registros finita) esto puede resultar sumamente difícil e implica un delicado balance entre funcionalidad de la instrumentación y *overhead* de ejecución.

El proceso de instrumentación en sí puede realizarse en distintos momentos. Si se realiza previo a la ejecución hablamos de instrumentación estática y generalmente implica modificar el programa en forma completa. Esto tiene la ventaja de poder analizar la totalidad del código a instrumentar. Su principal desventaja es que modificaremos sustancialmente el programa insertando fragmentos a lo largo de todo el texto, con lo cual será necesario reescribir el código ejecutable resultando en un programa de tamaño mayor. Adicionalmente, la integridad del programa original será vulnerada. Si el programa original depende de alguna forma de esto (eg. realizando verificaciones criptográficas) estaremos modificando la semántica original.

La instrumentación también puede hacerse en forma dinámica durante la ejecución. Esta modalidad *just-in-time* (JIT) requerirá de algún tipo de soporte del sistema operativo. Otra importante desventaja es que no resultará fácil hacer un análisis global de todo el programa. La instrumentación es incremental y debe ser eficiente de tal forma de no alterar aún más las condiciones de ejecución. En contrapartida, no es necesario modificar de antemano el programa, permitiendo mantener su integridad intacta. Una vez instrumentada una porción del código original usualmente no es necesario volver a hacerlo, amortizando el costo total de la instrumentación.

1.4. Instrumentación para análisis de cobertura

Para medir la cobertura de código, la instrumentación consistirá en insertar en puntos claves del código fragmentos (*probes*) que registren que la ejecución pasó por ese lugar. En términos de cantidad de información, un bit o un valor booleano es suficiente, y no nos in-

teresa ninguna otra variable. En términos de *overhead* de tiempo ejecución es relativamente liviano y no debería requerir gran cantidad de recursos de la CPU para implementarse.

1.5. Granularidad de instrumentación

Un parámetro importante para la instrumentación para medir cobertura consiste en decidir con qué granularidad instrumentar. El máximo nivel de granularidad que tiene sentido es el bloque básico. Un bloque básico es una secuencia lineal de instrucciones, que tiene un único punto de entrada y un único punto de salida. Cualquier programa ejecutado en un procesador tradicional es descomponible en bloques básicos. Usualmente la última instrucción de la secuencia es un salto condicional o incondicional, permitiendo múltiples posibles siguientes bloques básicos.

La disponibilidad del código fuente al momento de la instrumentación, determina qué tan posible es mapear la cobertura a elementos del programa original que tengan sentido semántico. Esto significa que si el código fuente no está disponible, puede por ejemplo no ser posible determinar la estructura original del programa. Aquí entran en juego otros factores como ser el lenguaje de programación utilizado, el compilador y cuál es la configuración utilizada en la compilación. Por ejemplo, un compilador puede aplicar la optimización de *inlining* lo que puede hacer invisible una función, eg. si la función *inlineada* no tiene condicionales desaparecerá dentro del bloque básico donde esté el código que la invoca.

En ausencia de código fuente no será posible entonces obtener un análisis de cobertura por líneas de código. Y dependiendo del lenguaje y optimizaciones aplicadas por el compilador, podría no ser posible una granularidad por función o método. Siempre será posible medir cobertura por bloque básico, pero generalmente esta medición resultará difícil de interpretar, de correlacionar semánticamente con la aplicación, y de comparar con otras mediciones del mismo programa. Desde luego que esto es irrelevante si lo que nos interesa es el número final (ie. el porcentaje de cobertura).

1.6. Objetivos

Queremos evaluar las herramientas existentes para medir cobertura de código de aplicaciones para Android. Nos limitaremos a herramientas que permitan realizar estas mediciones sin tener acceso al código fuente de las aplicaciones y trabajando directamente sobre el código compilado. En el caso de la plataforma Android, el código ejecutable se distribuye en archivos de formato APK, que se pueden descargar desde páginas web de distribución.

Lo que buscamos es comparar la efectividad de las herramientas existentes en cuanto a la cantidad de aplicaciones sobre las que funcionan correctamente. Trabajaremos sobre el conjunto de aplicaciones disponibles de la categoría “Top Gratis” para Argentina de Google Play Store [23]. También intentaremos caracterizar las aplicaciones según criterios de construcción para intentar correlacionar la aptitud de las herramientas con dichas características. Por ejemplo, es posible que algunas herramientas tengan dificultad en instrumentar aplicaciones escritas en el lenguaje Kotlin.

2. PRELIMINARES SOBRE ANDROID

2.1. Aplicaciones en Android

Android es un sistema operativo basado en el kernel Linux con una arquitectura orientada a dispositivos móviles. Una aplicación o app Android es un software programado para ser ejecutado en un sistema operativo Android. Dicho sistema especifica la manera en que esas aplicaciones deben ser empaquetadas y distribuidas, definiendo un formato de archivo denominado APK (por Android Package o Android Package Kit) que es una extensión del formato JAR (por Java Archive) de la plataforma Java.

Para instalar una aplicación es necesario obtener un archivo .apk. Este archivo contiene el programa en sí en formato binario y todos los recursos necesarios para su funcionamiento (imágenes, iconos, sonidos y otros archivos de datos).

Las aplicaciones Android se suelen obtener en Internet a través de sitios que facilitan su distribución y comercialización. El sitio más popular es Google Play Store administrado por Google (que es la principal empresa dando respaldo al sistema operativo), pero existen otros stores, eg. F-Droid que distribuye exclusivamente aplicaciones de código abierto. Además de facilitar la distribución de los binarios de las aplicaciones, los stores pueden prestar otro tipo de servicios a los desarrolladores y a los usuarios. A los primeros les otorga una plataforma de distribución (servidores, ancho de banda para descargas, etc), exposición y opciones de comercialización. Para los segundos las ventajas de usar un store provienen de la centralización (un sólo lugar de donde obtener múltiples apps), la facilidad para encontrar distintas aplicaciones (los stores generalmente dividen el contenido en categorías funcionales y permiten realizar búsquedas en su catálogo) y seguridad ya que el store puede garantizar la integridad de los archivos obtenidos y generalmente revisan y auditan las aplicaciones antes de ser distribuidas.

2.2. Componentes de una aplicación Android

Android especifica 4 tipos de componente [6] que una aplicación puede implementar, pudiendo definirse múltiples instancias de cada uno de estos tipos de componente según la funcionalidad ofrecida.

- *Activity*: conceptualmente representa una interacción con el usuario. Permite a la aplicación ofrecer una interfaz visual y acceso a dispositivos de entrada. Es el único componente visual de las aplicaciones Android y el único con el cual los usuarios interactúan directamente. Es por lo tanto el más utilizado, y la gran mayoría de las apps definen al menos un *activity*. Por ejemplo, una aplicación de email puede ofrecer un *activity* para navegar los mails del usuario, y un *activity* diferente para componer un nuevo mensaje. Esta modularidad permite que la función de composición de un email sea utilizada fácilmente de forma independiente (eg. mediante un acceso directo o desde otras aplicaciones).

- *Broadcast receiver*: es un componente que permite recibir en forma asincrónica e independiente eventos del sistema o de otras aplicaciones. No tiene componente visual, pero a través de *intents* puede lanzar un *activity*. Por ejemplo una aplicación

puede utilizar un receiver para recibir una alarma agendada al sistema operativo para ser disparada a una determinada hora. Esto permite que la aplicación no esté necesariamente ejecutándose (y consumiendo recursos) permanentemente.

- *Service*: un componente de ejecución en segundo plano de propósito general. Puede ejecutarse de manera independiente o estar vinculado con una aplicación, lo que implica que su tiempo de vida también está vinculado con dicha aplicación. Por ejemplo, una aplicación de reproducción de música utilizará un servicio para realizar la reproducción y esto le permitirá continuar con la misma aún si el usuario no tiene la interfaz (*activity*) activa en primer plano.
- *Content provider*: permite exponer a otras aplicaciones almacenamiento y acceso a recursos de algún tipo en forma controlada. Por ejemplo el sistema Android provee un content provider para consultar y manipular los contactos agendados del usuario. El acceso a los recursos está reglado por un sistema de permisos configurable por el propio *content provider*.

Una aplicación debe declarar las instancias de cada uno de los distintos tipos de componentes en el manifiesto del APK (ver sección 2.9). Los componentes así declarados pueden ser invocados o utilizados desde otras aplicaciones, típicamente el gestor de aplicaciones (application launcher), o servicios del sistema operativo.

2.3. La plataforma Android

El sistema operativo Android está basado en el kernel Linux. Provee un modelo de *sandbox* para la ejecución de las aplicaciones: cada aplicación se instala y se ejecuta en un usuario UNIX diferente y en procesos distintos. De esta forma ninguna aplicación puede interferir directamente con otras aplicaciones ni leer o escribir sus archivos. Cualquier interacción entre aplicaciones debe ocurrir a través del sistema operativo. Para ello Android introduce el concepto de *intent*, que representa la intención de interacción entre distintos componentes o aplicaciones del sistema.

El lanzamiento de aplicaciones se implementa a través de *intents* que activan los componentes de las mismas. Los *intents* están dirigidos principalmente a activities, pero también es posible enviarlos a broadcast receivers. Los *intents* son creados a partir de acciones del usuario, de eventos del sistema o programáticamente desde las propias aplicaciones.

Como parte del modelo de *sandbox*, las aplicaciones Android se ejecutan en una máquina virtual. Ésta y todo el entorno de ejecución se llama Android Runtime (ART), que es el sucesor del runtime Dalvik, reemplazándolo a partir de la versión Android 5.0. ART es compatible con Dalvik y el formato del código ejecutable para la máquina virtual se denomina Dalvik bytecode. La ejecución en máquina virtual permite además la independencia de la arquitectura de la CPU específica del dispositivo donde ejecuta la aplicación. Así es que Android tiene implementaciones para diversas arquitecturas de CPU (x86, ARM, MIPS) y las aplicaciones suelen ser portables entre las mismas sin necesitar ningún tipo de modificación.

El sistema Android ha ido evolucionando en el tiempo desde su versión inicial 1.0 del año 2008 y actualmente está disponible la versión 12. Con cada nueva versión se han agregado nuevas características y funcionalidades y el modelo de programación fue evolucionando también. Esto significa que ciertas aplicaciones pueden requerir una versión

de sistema operativo mínima para poder ejecutarse, y que versiones más modernas del sistema pueden no ejecutar aplicaciones implementadas para versiones anteriores. Si bien el runtime de las últimas versiones de ART es compatible con el formato binario de ejecución bytecode Dalvik original, no significa que las aplicaciones lo sean ya que pueden requerir funcionalidad del entorno de ejecución que fue declarado obsoleto y ya no está disponible.

2.4. Bytecode y código nativo

El formato ejecutable de Android es el bytecode Dalvik, que a través de una API Java ofrecida en la máquina virtual permite a las aplicaciones interactuar con el sistema operativo y el hardware del dispositivo. Esto provee:

- una capa de abstracción sobre el hardware permitiendo independencia tanto de la CPU como otros dispositivos periféricos del sistema; y
- una interfaz de programación compatible con lenguajes de alto nivel (Java, Kotlin y otros).

La API Java permite no sólo utilizar el lenguaje de programación Java para programar aplicaciones Android, sino que además permite aprovechar una buena parte del ecosistema que rodea al lenguaje. Es así que con pocas o ninguna modificación es posible utilizar en la plataforma muchas de las bibliotecas escritas para Java.

A la vez también es posible utilizar otros lenguajes de programación del ecosistema Java. Kotlin es uno de estos lenguajes, que desde el año 2017 tiene soporte oficial para desarrollar en Android, y desde 2019 es el lenguaje preferido por Google para esta actividad. Kotlin es completamente compatible con bibliotecas escritas para Java y ofrece ventajas ergonómicas y de seguridad deseables (eg. null-safety, lambdas, funciones de extensión, etc) y es por este motivo que ganó rápidamente popularidad en el dominio de las aplicaciones Android.

Si bien la API Java es la forma predeterminada y preferida para implementar aplicaciones en Android, también es posible escribirlas en otros lenguajes que compilen a código nativo de la CPU. Para ello, Android ofrece el Android Native Development Kit (NDK) que brinda una interfaz escrita en el lenguaje C++. La funcionalidad ofrecida nativamente en C++ es menor que la API Java, pero a través de JNI (Java Native Interface) siempre es posible (aunque mucho más laborioso) acceder y utilizar las interfaces Java desde código nativo.

Implementar una aplicación en código nativo tiene la desventaja que se pierde la independencia con la arquitectura de la CPU. Es así que en general estas aplicaciones deben distribuir dentro del APK el código binario compilado para todas las arquitecturas de CPU que se deseen soportar.

2.5. Dalvik bytecode y Java bytecode

Existen similitudes semánticas entre la máquina virtual Java y la máquina virtual Dalvik. Ambas brindan gestión de memoria automática (garbage collection) y manejo de excepciones. A más bajo nivel son más las diferencias que las similitudes. La JVM (Java Virtual Machine) es una VM basada en pila, mientras que la DVM (Dalvik Virtual

Machine) está basada en registros. El bytecode que interpretan ambas es entonces completamente diferente. Al ser una VM basada en registros, la DVM es más eficiente para ejecutar aplicaciones en sistemas con pocos recursos (el mapeo a registros de la CPU es más directo), y está diseñada para ejecutar múltiples instancias de la máquina virtual y el runtime simultáneamente y en paralelo en un mismo sistema.

La VM Dalvik [9] permite la utilización de hasta 64K registros, aunque la mayor parte de las instrucciones sólo pueden referenciar los primeros 256 (e inclusive hay formatos de instrucción que permiten usar sólo los primeros 16). Los métodos definen cuántos registros utilizarán, y los parámetros formales se reciben en registros, indexados en el final del bloque de registros solicitados.

El bytecode Java se puede transformar a bytecode Dalvik (y viceversa). Esta operación la realiza el programa `dx` que es parte del Android SDK, y este paso es parte fundamental del proceso de compilación de una aplicación Android. Luego este bytecode, al igual que el bytecode Java, es interpretado y compilado a demanda (just in time, o JIT) por la Dalvik VM. Versiones más modernas de Android introducen el Android Runtime (ART) [3] que reemplaza a la VM Dalvik. En este sistema el bytecode es compilado a código nativo al momento de instalar la aplicación (ahead of time compilation). Esto es a efectos de mejorar la eficiencia durante el uso normal de las aplicaciones. El ART mantiene el soporte de la semántica de la VM Dalvik, y ofrece la misma API de programación y las funcionalidades de gestión de código y memoria (que han sido mejoradas en las sucesivas versiones de ART).

Dado el diseño basado en registros de la DVM, es posible encontrar bytecode Java que no sea transformable en bytecode Dalvik, pero son casos extremos y usualmente no representan un problema.

2.6. Formato binario DEX

El bytecode Dalvik se almacena y transporta en archivos de formato DEX. Estos son equivalentes en función a los `.class` de Java, pero albergan múltiples clases. En las primeras versiones de Android todas las clases definidas por una aplicación se guardaban en un único archivo `classes.dex`. Sin embargo, el formato tiene limitaciones que con el correr del tiempo y la evolución de la plataforma se volvieron restrictivas. En particular, en un archivo DEX sólo se pueden almacenar hasta 64K métodos (referenciables mediante un entero de 16 bits). Para los strings existe una limitación similar, aunque ya es fácilmente subsanada mediante la utilización del formato de instrucción const-string/jumbo que utiliza referencias de 32 bits.

La limitación en la cantidad de métodos se mantiene, y esto limita a priori el tamaño de las aplicaciones. Este límite se alcanza muy rápidamente sobre todo cuando se utilizan bibliotecas de servicios (eg. Google Play services) que contienen una gran cantidad de clases y métodos. La solución que ofrece la plataforma es separar las definiciones de las clases y métodos en múltiples archivos `.dex` (nombrados `classes2.dex`, `classes3.dex`, etc) para conformar una aplicación denominada multidex.

2.7. Smali

El bytecode Dalvik describe operaciones a realizar en la máquina virtual. Hay herramientas [10] que permiten su manipulación mediante la decodificación y reescritura de los

archivos `.dex`.

Para facilitar el análisis y comprensión por humanos, también se puede decompilar el bytecode a una representación textual que se denomina Smali. Las herramientas [12] para realizar esta decompilación también permiten el camino inverso y compilan texto Smali a bytecode Dalvik. Esto hace posible otra forma de manipulación del bytecode.

2.8. Empaquetado de una aplicación Android

Una aplicación Android se empaqueta y se distribuye en el formato APK. Un archivo `.apk` es una extensión del formato `.jar` (Java Archive, a su vez una extensión del formato `.zip`) que permite adosarle una firma digital que certifica la integridad del contenido (ver sección 2.12).

Un APK usualmente contiene:

- el bytecode Dalvik en un archivo `classes.dex`, y opcionalmente otros `.dex` numerados a partir de `classes2.dex` para aplicaciones multidex
- código nativo, dependiente de la arquitectura de CPU, en subdirectorios `lib/x86`, `lib/x86_64`, `lib/armeabi-v7a`, `lib/arm64-v8a`, etc.
- un archivo de manifiesto `AndroidManifest.xml` que describe los componentes de la aplicaciones, permisos y otros requerimientos (ver sección 2.9)
- un archivo binario, `resources.arsc`, que contiene los recursos de la aplicación compilados
- otros archivos de recursos no incluidos en `resources.arsc`

Un requisito más que el sistema operativo suele requerir de los archivos APKs es que se encuentren alineados. Esta característica corresponde en realidad al formato ZIP y es una característica que permite que el contenido de los archivos dentro del `.zip` estén alineados en múltiplos de 4 bytes, lo que a su vez permite que si no están comprimidos puedan ser mapeados por el sistema operativo con `mmap()` y puedan ser consumidos directamente.

2.9. El manifiesto de una app Android

El manifiesto de una aplicación Android [5] es un archivo XML que contiene la definición de todos los componentes de la misma (activities, services, broadcast receivers, content providers, ver sección 2.2), los permisos que ésta requiere, y otros requerimientos sobre el sistema y el dispositivo donde se instala la aplicación.

Para cada uno de los componentes, además de definir atributos básicos como su descripción, icono para representación visual y clase que implementa su funcionalidad, encontraremos una serie de filtros de *intent* que activan dicho componente. Esta es la manera de indicar qué eventos del usuario o del sistema deben recibir y activar la aplicación.

Existe además un tipo de componente adicional a los 4 antes definidos: un elemento `<instrumentation>` [13] permite definir una clase que sirve de agente de instrumentación cuando la aplicación se ejecuta en este modo (esto se logra usando la herramienta `am` del sistema). En ese caso el sistema operativo carga primero que nada la clase declarada como de instrumentación y permite que esta monitoree la interacción entre la aplicación y el sistema. Una de las herramientas que analizaremos hace uso de esta funcionalidad.

2.10. XML binario y recursos

Si bien el archivo de manifiesto es un XML, no se distribuye en el APK en formato textual. En su lugar se utiliza un formato binario no documentado [25].

Android utiliza este formato binario por razones de eficiencia. El formato está mejor optimizado para su lectura en sistema con bajos recursos y está segmentado facilitando su acceso directo por bloques.

Es usual encontrar que desde el manifiesto se referencian recursos adicionales: los iconos antes mencionados, etiquetas visibles al usuario, opciones de configuración, etc.. Todos (o casi todos) los recursos son compilados en un único archivo `resources.arsc` y las referencias en el manifiesto son punteros a elementos dentro de este archivo. En este paquete de recursos encontramos codificados listas de strings, archivos de imagen y otros documentos XML, también convertidos al formato binario para su consumo más eficiente.

Algunas características del dispositivo u opciones del usuario condicionan qué recursos son seleccionados. Así por ejemplo, encontramos distintas listas de strings, una para cada idioma de sistema soportado por la aplicación. Otros atributos de selección son resolución de pantalla del dispositivo, versión del sistema operativo y otros similares.

Por último, el sistema operativo contiene un conjunto de recursos predefinidos que pueden ser referenciados desde el manifiesto u otros archivos XML de recursos.

La compilación del manifiesto XML al formato binario y el empaquetado y codificación de los recursos es un proceso que implica cierta pérdida de información. Esto sumado a la ausencia de documentación oficial del formato y la dependencia de recursos definidos por el sistema pueden implicar que sea difícil una alteración del manifiesto mediante decodificación y recompilación.

2.11. Empaquetado de código nativo

Estando Android basado en el kernel Linux, el código binario para la plataforma se consume bajo el formato ELF en forma de biblioteca compartida (archivos `libXXXX.so`). Al igual que la máquina virtual Java, la DVM puede cargar dinámicamente estos archivos e invocar el código por ellos definido usando JNI (Java Native Interface).

Como dijimos antes lo que sí es necesario es que el código corresponda a la arquitectura de CPU del dispositivo donde se instala la aplicación. Por este motivo en los APKs es frecuente encontrar distintos archivos para las distintas arquitecturas que la aplicación soporta. Encontramos las bibliotecas de código nativo dentro del directorio `lib`, en el subdirectorio correspondiente a la arquitectura: `x86`, `x86_64`, `armeabi-v7`, `arm64-v8a`, `mips` ó `mips64` (estas últimas dos están consideradas obsoletas en versiones recientes de Android).

Cabe mencionar que los APKs descargados desde Google Play Store pueden estar limitados en las arquitecturas nativas soportada a la del dispositivo con que se realiza la descarga. Esta reducción se realiza para optimizar el tamaño de descarga y el espacio que utiliza luego la aplicación instalada en el dispositivo.

2.12. Integridad de un APK

Como mencionamos en la sección 2.8, los APKs cuentan con una firma digital [7] que garantiza su integridad e identifica el origen de los mismos. Dependiendo de dónde se

obtenga el APK, puede estar firmado por el desarrollador original o por el store desde donde se descargó (Google Play Store ofrece esta modalidad desde 2017).

La firma digital permite que el sistema operativo pueda verificar que el contenido del APK no fue alterado desde su compilación y que el origen corresponde a un desarrollador confiable. Un dispositivo Android puede rechazar una instalación si detecta una firma inválida o proveniente de un desarrollador desconocido. La restricción de la verificación de origen está usualmente relajada en un dispositivo emulado y se puede desactivar en uno físico si se habilita el modo desarrollador.

3. HERRAMIENTAS A ANALIZAR

En esta tesis analizaremos las herramientas Ella [28], ACVTool [26] y COSMO [27]. Todas estas herramientas realizan instrumentación estática del bytecode de la aplicación bajo análisis. Es decir, modifican los archivos `.dex` que están dentro del APK, alterando los métodos de las clases existentes y posiblemente agregando nuevas clases y métodos. Ninguna de las herramientas realiza instrumentación del código nativo que las aplicaciones puedan tener, con lo cual no será posible analizar la cobertura en el mismo.

En todos los casos, hay pasos del procedimiento que son necesarios independientemente de la herramienta utilizada. Todas las herramientas modifican de una forma u otra los archivos `.dex` dentro del APK. Con lo cual todas deben reconstruir el APK y proceder a firmarlo, obviamente con un certificado diferente del original.

3.1. Estrategias de instrumentación y medición

Para medir la cobertura las herramientas crean algún tipo de estructura de datos para registrar si la ejecución pasó por una sección de código o no. Generalmente y por cuestiones de eficiencia, esta estructura será un arreglo de valores booleanos. Luego el código inyectado durante la instrumentación modifica el valor booleano correspondiente a la sección instrumentada. Es decir que el fragmento inyectado debe, de alguna forma obtener una referencia a la estructura de datos de registro y mutarla para indicar la visita por la sección de código. Esta está identificada usualmente con una constante numérica que es generada al momento de la instrumentación.

Para efectuar la instrumentación las herramientas deben modificar el bytecode de la aplicación, y para realizar esto cada una de ellas utiliza una estrategia diferente de manipulación de los archivos `.dex`.

Asimismo, la estrategia para obtener los resultados de la medición de cobertura varía de herramienta en herramienta.

3.2. Ella

Analizaremos la versión modificada de Ella [28] con soporte para aplicaciones multidex. La misma fue publicada en septiembre de 2018, y la herramienta original es del año 2016. El último *commit* relevante de código encontrado en el repositorio es de diciembre de 2018.

Ella instrumenta la aplicación manipulando directamente el bytecode DEX. Utiliza la biblioteca `dexlib2` que permite parsear archivos `.dex`, modificar su contenido y escribir el nuevo bytecode. Ella únicamente permite medir cobertura con granularidad de método.

Para esto agrega a la aplicación clases utilizadas en tiempo de ejecución para registrar las mediciones de cobertura y realiza las siguientes modificaciones en cada método del código original:

1. Aumenta el tamaño del *frame* (ie. la cantidad de registros) en 1 más los necesarios para duplicar los parámetros del método.

2. Invoca un método estático en una de las clases de runtime para notificar el paso por el método instrumentado, identificado por un valor constante determinado en

tiempo de instrumentación. El registro adicional solicitado se utiliza para almacenar este identificador.

3. Copia los parámetros del método (que son siempre los últimos del *frame*) a otros de tal forma que el índice de cada parámetro se vea inalterado (respecto del original) en el resto del código del método.

Por ejemplo si el método necesitaba originalmente 5 registros y 1 de ellos era un parámetro, el mismo es referenciado en el registro `v4`. Luego de la instrumentación el método pasa a solicitar 7 registros y el parámetro (como se ubica siempre en las últimas posiciones) pasa a recibirse en `v6`. El fragmento inyectado por Ella entonces usa `v5` para guardar la constante que identifica el método y como parte de la inicialización copia `v6` en `v4` de forma tal que el resto del bytecode del método pueda funcionar sin otras modificaciones.

Las clases de runtime agregadas tienen además una sección de inicialización estática (es decir que se ejecuta automáticamente al cargarse el código de la aplicación) que inicia un nuevo thread para subir periódicamente los datos de medición de cobertura. Éste abre un *socket* TCP a un servidor Ella, que debe estar en ejecución y disponible al momento de realizar la medición, y periódicamente con un intervalo de 1 segundo envía un *payload* JSON con los identificadores de los métodos que fueron ejecutados.

El registro de cuáles métodos fueron ejecutados se almacena en un set concurrente que permite que cualquier thread inserte nuevos identificadores de método (*probes*).

3.3. ACVTool

ACVTool [26] fue publicada en octubre de 2018. Recibió luego de esa fecha algunas modificaciones, siendo la última relevante la actualización de la dependencia `apktool` en octubre de 2020.

Para instrumentar la aplicación, ACVTool realiza una manipulación del bytecode similar a Ella, pero en lugar de trabajar directamente sobre el archivo `.dex`, lo decompila primero a su representación Smali utilizando la herramienta apktool [4]. Luego utiliza una biblioteca Python que permite parsear el código Smali y manipular el AST del mismo.

ACVTool permite medir cobertura con granularidad variable desde clase a instrucción (con algunas limitaciones mencionadas en el paper). A efectos de esta tesis utilizaremos la granularidad de método para poder compararla objetivamente con las demás herramientas que no admiten mayor detalle.

Además de duplicar los parámetros de cada método de forma similar a Ella, ACVTool solicita 3 registros adicionales para poder mutar un arreglo de booleanos en forma directa en lugar de realizar una invocación estática a una clase de soporte de runtime. Esto reduce el *overhead* de la instrumentación y es de particular importancia en el caso que se instrumente con granularidad de instrucción. Los tres registros guardan: una referencia al arreglo de booleanos; el identificar numérico del *probe*; y el valor *true* 1. Con esto, la mutación del arreglo se realiza mediante una instrucción `aput-boolean` que recibe los 3 registros como parámetros.

Luego ACVTool agrega dos clases de soporte de runtime:

1. `AcvReporter` es generada dinámicamente y tiene el sólo propósito de concentrar los arreglos de booleanos para cada clase instrumentada, que se sostienen en campos

estáticos que son accedidos por el fragmento de inicialización de cada método instrumentado. Cada arreglo tiene el tamaño que permite almacenar todos los *probes* inyectados en la clase en cuestión. Adicionalmente implementa un método que permite serializar el contenido de todos estos arreglos y que se utiliza para generar los archivos de reporte de cobertura.

2. `AcvInstrumentation` (y clases internas) es fija y es una implementación de un componente `Instrumentation` de Android. Esto permite que su código se ejecute antes de cargar el resto de la aplicación y poder monitorear excepciones y errores fatales y guardar los datos de medición antes de finalizar la ejecución. Adicionalmente instala un `BroadcastReceiver` con un filtro de *intent* para capturar el mensaje de sistema que se envía para detener la medición y generar el reporte en momentos arbitrarios.

Luego de modificar los archivos `.smali` y de agregar las nuevas clases de soporte, ACVTool recompila estos fuentes modificados generando nuevamente los archivos `.dex`. Para esto utiliza nuevamente la herramienta `apktool`.

Cuando se generan los archivos de reporte se guardan en el directorio `/mnt/sdcard` con lo cual la aplicación necesita el permiso de escritura en almacenamiento externo. A tal efecto, durante la instrumentación ACVTool decompila y modifica el manifiesto agregando este permiso (si no está ya declarado para la aplicación) y declarando el componente de instrumentación en la clase `AcvInstrumentation` mencionada previamente. La decompilación del manifiesto, y su posterior recompilación al formato binario XML también se realiza usando la herramienta `apktool`.

Para realizar la medición de cobertura es necesario ejecutar la aplicación en modo de instrumentación para permitir que el código inyectado en `AcvInstrumentation` se ejecute primero antes de iniciar el resto de la aplicación. ACVTool provee un comando que hace esto usando el comando `adb` del SDK, en combinación con `am` (por *Activity Manager*) del sistema operativo Android.

Para detener manualmente la medición de cobertura, también utilizando `adb am` se dispara un *intent* que es recibido por el `BroadcastReceiver` instalado por `AcvInstrumentation` y este genera el reporte en el directorio de almacenamiento externo que luego se puede descargar del dispositivo.

3.4. COSMO

COSMO [27] fue publicada en abril de 2021 (aunque el primer *commit* registrado es de noviembre de 2020) y no ha recibido actualizaciones significativas desde su publicación.

COSMO permite tanto la instrumentación de un APK como la instrumentación de una aplicación a partir de su código fuente. En ambos casos utiliza la misma biblioteca JaCoCo que trabaja sobre bytecode Java. A efectos de esta tesis utilizaremos únicamente la funcionalidad de instrumentación de APK.

Como JaCoCo actúa sobre bytecode Java, el primer paso para JaCoCo es reconvertir el código DEX. Para esto utiliza la herramienta `dex2jar` [16] que recibe el `.apk` y escribe un `.jar` con todas las clases convertidas a archivos `.class`.

JaCoCo entonces procede a instrumentar cada una de esas clases. En cada clase inyecta:

1. Una variable estática referencia a un arreglo de booleanos `$jacocoData`.

2. Un método estático `$jacocoInit()` que inicializa por única vez `$jacocoData` con un arreglo obtenido de una de las clases de soporte en runtime de JaCoCo y luego devuelve siempre esta misma referencia.

Luego los *probes* se agregan en todos los métodos de la clase y el fragmento consiste en obtener una referencia al arreglo de booleanos invocando el método `$jacocoInit()` y mutando el índice correspondiente.

A diferencia de Ella o ACVTool, y como el código instrumentado es ahora bytecode Java, no es necesario modificar la cantidad de registros ni duplicar los parámetros, ya que la VM de Java está basada en un *stack*.

Luego COSMO reconvierte los archivos `.class` al formato DEX utilizando la herramienta `dx` del SDK Android. Junto con las clases instrumentadas, también se agregan a la aplicación las clases pertenecientes al runtime de JaCoCo. Aquí está implementada la funcionalidad para obtener y mantener los arreglos de booleanos para almacenar los *probes* y el código para escribir un reporte de la medición de cobertura a un archivo.

Para reconstruir el APK instrumentado COSMO utiliza el módulo `zipfile` del lenguaje Python.

3.5. Modificaciones a COSMO

Nótese que en la descripción anterior no hicimos mención a cómo disparar la generación del reporte de cobertura. Esto es porque la herramienta COSMO no implementa ningún mecanismo para realizarlo. En el modo de instrumentación a partir del código fuente sí se inserta un BroadcastReceiver y se agrega su declaración en el manifiesto. Pero no ocurre nada similar en el caso de instrumentar un APK.

Usualmente JaCoCo se utiliza en Android con un APK de testing que se ejecuta como agente de instrumentación y es quién notifica al runtime que debe escribir el archivo con el resultado de la medición de cobertura. No encontramos en la documentación de COSMO o en el paper ninguna mención a este requerimiento. Por este motivo y para obtener algún tipo de resultado de COSMO, aprovechando la existencia de los mecanismos que ofrecen las clases de runtime para generar y guardar el reporte de cobertura en una ruta determinada, realizamos las siguientes modificaciones:

1. Agregamos un componente `BroadcastReceiver` a la aplicación instrumentada, cuyo único propósito es invocar el método de generación de reporte cuando recibe un *intent* del sistema. Esto consiste en agregar los `.class` correspondientes al conjunto de clases a convertir a DEX.

2. Registramos el *receiver* anterior en el manifiesto del APK. Para esto decompilamos el manifiesto binario XML utilizando la herramienta `apktool`, modificamos el XML y lo recompilamos a formato binario con la misma herramienta.

3. Agregamos un archivo de `.properties` con una configuración nula para que el runtime de JaCoCo se inicialice correctamente.

De esta forma es posible ejecutar normalmente la aplicación instrumentada y en el momento en que se decide finalizar la medición, enviar un *intent* usando `adb am` como en el caso de ACVTool. El archivo de reporte generado se almacena en el directorio privado de la aplicación, con lo cual no es necesario otorgar nuevos permisos a la aplicación.

Para la ejecución de las pruebas y a partir de este punto en lo que resta de esta tesis, cuando nombramos COSMO nos referimos a la versión modificada.

4. DESCRIPCIÓN DE LOS EXPERIMENTOS

Evaluaremos las herramientas antes mencionadas con el objetivo de comparar su efectividad para medir cobertura de código de aplicaciones para Android, realizando las pruebas en la modalidad de caja negra y sin acceso al código fuente.

4.1. Fases de los experimentos

El experimento a realizar consiste en la instrumentación para medición de cobertura, instalación en un dispositivo Android, ejecución y ejercitación de funcionalidad, y posterior obtención de los resultados de medición, sobre un conjunto de aplicaciones disponibles en forma pública.

El resultado final esperado es el registro de medición y posterior reporte de cobertura obtenido a partir de su análisis utilizando cada una de las herramientas antes mencionadas.

Sin embargo recopilaremos y analizaremos los resultados parciales de: instrumentación del código, reconstrucción del APK modificado, y factibilidad de instalación y ejecución del mismo.

4.2. Criterios de aceptación

Consideraremos que una herramienta instrumenta y mide correctamente la cobertura de código de una aplicación cuando es posible ejecutar la aplicación y ejercitar su funcionalidad durante un período de tiempo no menor a 15 segundos y es posible obtener como resultado final los registros de cobertura de los métodos visitados durante la prueba. Todo el proceso se realiza en forma automática, incluída la ejercitación de las funcionalidades de la aplicación.

4.3. Obtención de los APKs

Los APKs de las aplicaciones ejercitadas fueron obtenidos del Google Play Store para la región Argentina. Elegimos las primeras 200 aplicaciones de la sección Top Gratis. La fecha de acceso y descarga del ranking fue el 16 de marzo del 2022. Estas son las aplicaciones más descargadas del Play Store para la región elegida y es un ranking que por supuesto varía en forma diaria. El listado completo de los APKs analizados se encuentra en el Apéndice A.

El conjunto de aplicaciones se compone de una multitud de categorías, desde aplicaciones para mensajería y redes sociales hasta aplicaciones bancarias o financieras. Si bien la mayoría de las misma necesitarán credenciales de acceso para exponer la totalidad de su funcionalidad, a efectos de esta tesis consideramos suficiente que cumplan con los criterios de aceptación para la funcionalidad parcial ofrecida en modo anónimo.

Prácticamente todas las aplicaciones obtenidas del Google Play Store tienen carácter comercial, con lo cual es usualmente imposible obtener el código fuente de las mismas. Aún cuando fuera posible, analizaremos las herramientas instrumentando únicamente el artefacto binario distribuido en el store.

4.4. Caracterización de las aplicaciones

Intentamos caracterizar las aplicaciones obtenidas a partir de una serie de particularidades observables a partir del análisis del contenido del APK. Las dimensiones de dicha caracterización son variadas pero tendientes a buscar rasgos que nos permitan correlacionarlos con los resultados obtenidos con cada herramienta. Por otro lado, otras clasificaciones son interesantes porque son una expresión de tendencias en el desarrollo de aplicaciones para dispositivos móviles, y en particular para Android.

Por ejemplo, detectamos aplicaciones que contienen partes implementadas con los frameworks Flutter [11] o React Native [17]. Este aspecto interesa a los efectos de medición de cobertura ya que para estas aplicaciones las funcionalidades están implementadas en lenguajes que no compilan a Dalvik bytecode. En el caso de Flutter, se utiliza el lenguaje Dart para escribir la funcionalidad y éste compila a código nativo. En aplicaciones escritas con React Native la funcionalidad de escribe en Javascript que es optimizado y ofuscado, para ser incluído en el APK en un archivo de *bundle*. En ambos casos sabemos de antemano que las mediciones de cobertura no expresarán cabalmente cuánto del código de la aplicación fue ejercitado, independientemente de la exhaustividad de las pruebas realizadas, ya que el código de la funcionalidad de la aplicación no será instrumentado por las herramientas utilizadas.

Otro caso más extremo es la tecnología Superpack [22] de Facebook que utilizan todas las aplicaciones de la compañía Meta: Facebook, Instagram, WhatsApp, etc.. Superpack es una tecnología de compresión que reduce el tamaño final del APK resultante. Para lograrlo reescribe los archivos incluídos en el archivo distribuible lo que hace irreconocible el bytecode. Por lo tanto será imposible de instrumentar de manera estática el código final del la aplicación, pudiendo hacerlo sólo para el programa de descompresión.

En los casos mencionados hasta el momento no encontramos razones a priori para suponer que sea imposible la instrumentación de las aplicaciones. Pero ciertamente el número medido de porcentaje de cobertura no será indicativo en ninguno de los casos acerca de qué tanto del código original de la aplicación estamos ejercitando con las pruebas, sin importar qué tan exhaustivas sean éstas (ie. porque no es posible instrumentar la totalidad del código relevante). En este sentido, e independientemente de frameworks detectados, también medimos el tamaño relativo entre el bytecode Dalvik y el código nativo encontrado dentro del APK.

Hay otros rasgos que analizamos que sí pueden interferir en la instrumentación o en la viabilidad de ejecución de las aplicaciones una vez instrumentadas. Una descripción más detallada de las limitaciones de las herramientas en este sentido se puede encontrar en las secciones 5.2 y 5.5. Caracterizamos las aplicaciones según si el bytecode está escrito en Kotlin (al menos parcialmente), o si la aplicación utiliza las bibliotecas de Google Play Services o Firebase (ambas bibliotecas poseen funcionalidad que permite que la aplicación realice un chequeo de integridad del código).

La caracterización de las aplicaciones no es rigurosa y surge de la observación del contenido del APK. Aún así, resulta de interés el análisis de correlación entre los resultados obtenidos y los rasgos de las aplicaciones, así como también pueden indicar tendencias o proveer puntos de referencia en cuanto a estrategias de desarrollo de las aplicaciones dirigiendo posibles trabajos futuros sobre las herramientas de análisis.

4.5. Monkey y software de testing

Para ejercitar la funcionalidad de las aplicaciones instrumentadas utilizaremos la herramienta `monkey` que está incluída en la distribución estándar del sistema operativo Android. Se trata de una herramienta que inyecta estímulos de entrada aleatorios: acciones de navegación, movimientos y pulsaciones en el dispositivo de entrada de pantalla, etc. Si bien los eventos sintetizados son aleatorios, es posible elegir una semilla haciendo posible la reproducibilidad del flujo de estímulos enviados a la aplicación.

Para estos experimentos, elegimos una semilla diferente por cada aplicación, que utilizaremos para las pruebas con el APK instrumentado por cada una de las herramientas.

Configuramos la ejecución de `monkey` con una cantidad de eventos finita y una cadencia fija tal que la aplicación se ejercite al menos durante el tiempo mínimo establecido según el criterio de aceptación.

4.6. Emulador Android

Para la instalación y ejercitación de las aplicaciones con la consecuente medición de cobertura utilizaremos un emulador de un dispositivo Android, para garantizar la reproducibilidad y facilitar la automatización de las pruebas. El emulador utilizado se ejecuta en el sistema operativo Linux en una arquitectura Intel x86 de 64 bits.

El dispositivo emulado también tiene una arquitectura x86 por cuestiones de eficiencia de emulación. Esta arquitectura no es la encontrada habitualmente en dispositivos físicos móviles. En teléfonos o tablets usualmente la CPU tiene una arquitectura ARM. Es por esto que alguno de los APK obtenidos sólo son instalables en arquitecturas ARM. Obviamente esto es relevante si la aplicación en cuestión tiene partes implementadas con código nativo, ya que como dijimos previamente el bytecode Dalvik es portable entre distintas arquitecturas de CPU.

Una alternativa para sortear este problema es realizar emulación del hardware de otra arquitectura (ARM en este caso). Pero esto es altamente ineficiente y ejecutar la aplicación con esta limitación de desempeño puede influir de maneras no esperadas en las mediciones.

4.7. Versiones de Android para las pruebas

En cuanto al software del sistema Android de los dispositivos emulados utilizamos dos versiones [20] diferentes:

- Android 11.0 (API versión 30)
- Android 7.0 (API versión 24)

La versión del sistema operativo y la versión de API ofrecida no el mismo número pero tienen una correlación directa. El SDK de desarrollo se designa en función de la versión de API para la cual fue liberado (ie. SDK 30 ofrece las características y funcionalidades de la API versión 30).

Android 11 fue lanzado en el año 2020 [24]. Tiene una importante ventaja: el sistema operativo provee una capa de emulación de arquitectura ARM a nivel de sistema [1]. Es decir que aún cuando el dispositivo emulado tenga una arquitectura x86 es posible instalar

y ejecutar nativamente APKs que sólo soporten arquitecturas ARM. La principal desventaja para este experimento es que es un sistema relativamente nuevo y no representativo del universo de diferentes versiones de Android instaladas a nivel global. Por otro lado, y como las herramientas analizadas fueron escritas hace unos años es posible encontrar incompatibilidades específicas con versiones posteriores del sistema operativo Android (ver sección 5.6).

Android 7 fue elegido justamente por este motivo. Fue lanzada en el año 2016 [2] y las tres herramientas que analizamos en esta tesis fueron escritas luego de ese año, con lo cual es esperable que todas sean compatibles con esa versión.

4.8. Validación inicial de los APKs

Realizaremos un paso inicial de validación de los APKs obtenidos en la sección 4.3. Consideraremos válidos los APKs que puedan ser correctamente instalados y que finalicen la secuencia de eventos inyectados por `monkey`, configurado con la misma semilla que utilizaremos durante el experimento. Probaremos todos los APKs descargados en ambas versiones de Android. Si la aplicación no se puede instalar, o deja de responder durante las pruebas, o termina con un error fatal, la descartaremos para el resto de los experimentos a ejecutar en la versión Android dada.

5. RESULTADOS

En la figura 5.1 vemos la progresión en las distintas etapas del experimento mostrando cuántas aplicaciones finalizaron exitosamente cada etapa con al menos una de las herramientas, para alguna de las dos versiones del sistema operativo Android utilizadas.

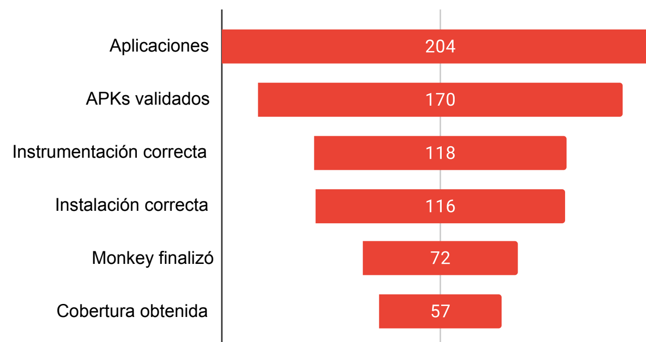


Fig. 5.1: Resultados del experimento para al menos una herramienta

Podemos ver que de las 204 aplicaciones descargadas, sólo 170 pasaron el proceso de validación. De ese conjunto, 118 pudieron ser instrumentados por al menos una de las herramientas. Analizaremos las causas de fallo en esta etapa con mayor detalle en la sección 5.3. Con un par de excepciones, todos los APKs instrumentados pudieron ser instalados para ser ejercitados con `monkey`, pero aproximadamente dos tercios finalizan con éxito ese paso. Y por último aparecen problemas en la etapa de obtención del reporte de cobertura. Analizaremos los fallos en estas 3 últimas fases en la sección 5.6.

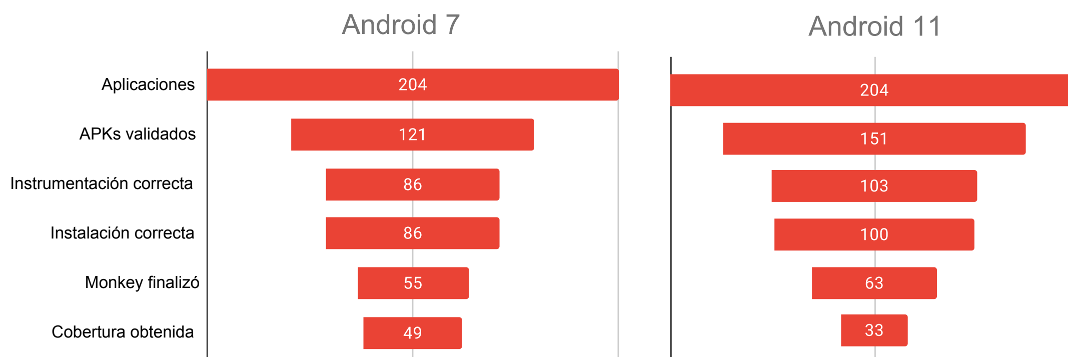


Fig. 5.2: Resultados del experimento para al menos una herramienta, por versión Android

En la figura 5.2 vemos los resultados pero distinguiendo lo ocurrido en las dos versiones de Android por separado. Aquí podemos observar una notable diferencia entre ambas

versiones: la cantidad de APKs validados para Android 11 es mayor que para Android 7. Esto se debe fundamentalmente a que como varios de los APKs obtenidos están en parte implementados en código nativo pero sin soporte para la arquitectura del *host* del emulador, x86, no pueden ser instalados en Android 7, y por lo tanto fueron descartados del conjunto válido para esta versión. Como dijimos en la sección 4.7, Android 11 ofrece emulación para la arquitectura ARM, lo que permite que estas *apps* se puedan instalar sin inconvenientes.

La proporción de APKs instrumentados correctamente es aproximadamente la misma para los conjuntos validados en ambas versiones (70 %). La instalación presenta algunos problemas en Android 11. Veremos más adelante (en la sección 5.6) que se debe sobre todo a nuevas restricciones sobre la integridad del APK que introduce la nueva versión y que no son satisfechas por Ella.

La siguiente etapa, la finalización correcta de `monkey`, muestra otra reducción importante entre las aplicaciones que la completan con éxito, pero en igual proporción para ambas versiones de Android (63 %). Por último la obtención de la cobertura medida que representa la última etapa muestra que para Android 11 los resultados son peores que para Android 7. Podemos adelantar que se debe a una falla crítica en la herramienta ACVTool, pero analizaremos en mayor detalle las causas en la sección 5.6.

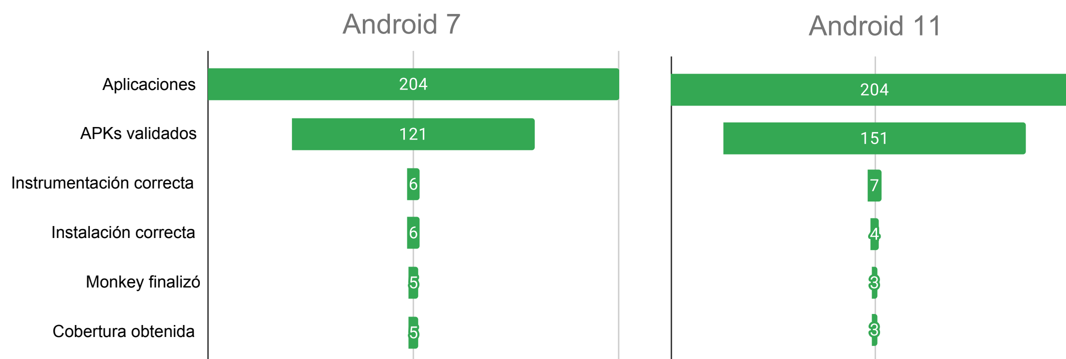


Fig. 5.3: Resultados del experimento para Ella

En las figuras 5.3, 5.4 y 5.5 vemos la evolución en las sucesivas etapas para cada herramienta por separado.

Ella, siendo la herramienta más antigua sólo logra instrumentar una muy pequeña porción de las aplicaciones (figura 5.3). Veremos en la sección de análisis que esto en buena parte se debe a la utilización de dependencias que tienen errores que ya fueron subsanados. Como ya mencionamos antes, también tiene una limitación en el firmado del APK instrumentado que reduce sensiblemente los *subjects* en el escenario de Android 11.

ACVTool (figura 5.4) logra instrumentar aproximadamente un tercio de las aplicaciones, pero luego muestra buenos resultados en la instalación y ejecución de `monkey`. Sin embargo y como mencionamos antes, para Android 11, prácticamente todos los APKs fallan en la última etapa al intentar obtener los resultados de la medición. Los fallos en Android 7 en esta instancia se deben en buena medida a otros problemas inherentes a la herramienta.

Por último COSMO (ver figura 5.5) es la herramienta que logra instrumentar la mayor

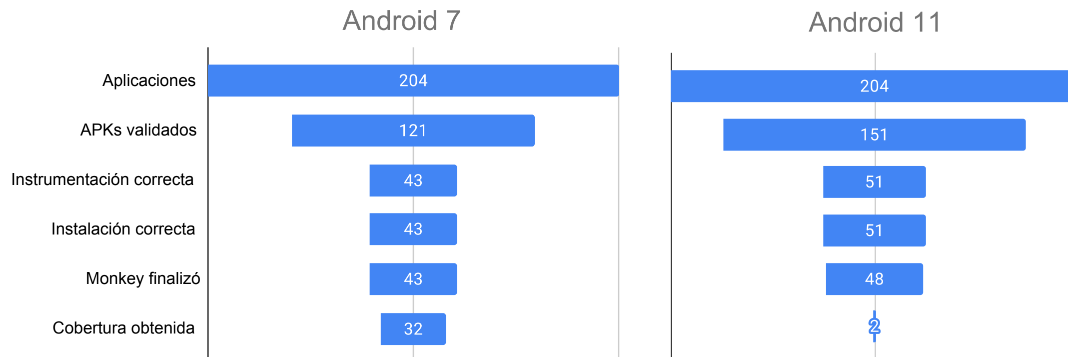


Fig. 5.4: Resultados del experimento para ACVTool

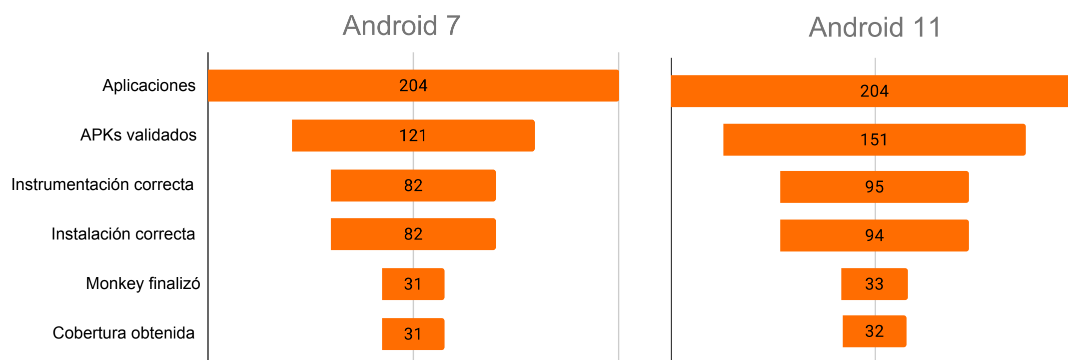


Fig. 5.5: Resultados del experimento para COSMO

cantidad de aplicaciones, cerca de dos tercios del total, y éstas se instalan correctamente en la mayoría de los casos. Sin embargo se puede apreciar una gran merma en los resultados en el paso de ejercitación con `monkey`. Esto se debe a que el proceso de instrumentación introduce errores en el código que llevan a fallas críticas que no permiten la ejecución correcta de la aplicación.

5.1. Resultados de la instrumentación

En la tabla 5.1 vemos un resumen de las cantidades y porcentajes de las aplicaciones instrumentadas exitosamente por cada una de las herramientas sobre el conjunto unión de los APKs válidos para ambas versiones de Android.

Es interesante ver en la figura 5.6 la intersección entre los conjuntos de aplicaciones instrumentadas por cada herramienta. Observamos que existen APKs que sólo son instrumentados por una única herramienta: 3 por Ella, 5 por ACVTool y 58 por COSMO. Y hay un número significativo, 52, que no puede ser instrumentado por ninguna herramienta.

Otra observación interesante es que a pesar de utilizar estrategias sustancialmente distintas, la mayoría de los APKs que instrumenta ACVTool, también los instrumenta COSMO, pero no se dá el caso inverso y hay un número importante de aplicaciones que

exclusivamente COSMO puede instrumentar.

Herramienta	Instrumentaciones exitosas	
Ella	7	(4.1 %)
ACVTool	57	(33.5 %)
COSMO	110	(64.7 %)

Tab. 5.1: Instrumentaciones exitosas sobre una base de 170 APKs

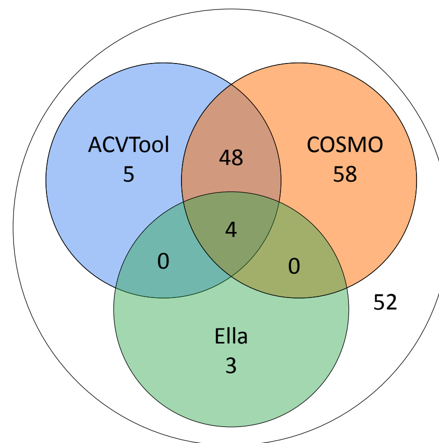


Fig. 5.6: Instrumentaciones según herramientas

5.2. Limitaciones *a priori* en la instrumentación

Los papers y la documentación para cada herramienta describen una serie de limitaciones conocidas que aplican tanto a la fase de instrumentación como a la de ejecución y obtención de resultados. Algunas de estas limitaciones ya no son relevantes, pués las herramientas han sufrido algunas modificaciones y adaptaciones desde el momento de su publicación.

Ella menciona una única limitación entre la documentación del repositorio Git y se refiere a que un archivo DEX puede contener sólo 65.535 métodos. Dado que Ella inyecta nuevos métodos en cada binario `.dex` para su uso en runtime, es posible que no pueda ser instrumentado.

Para ACVTool, las limitaciones mencionadas incluyen: posibles errores y problemas con `apktool`, la herramienta de decompilación y re-empaquetado de los APKs, que forma parte central del flujo de trabajo de ACVTool; no se pueden instrumentar métodos con más de 253 registros, pues la instrumentación requiere 3 nuevos registros que deben estar indexados debajo del límite de 256 utilizables por la instrucción `aput-boolean`; existe una limitación en la cantidad total de instrucciones que pueden componer un método aunque en la práctica y aún instrumentando con granularidad de nivel de instrucción no es alcanzado.

El paper de COSMO menciona que JaCoCo en modo *offline* no es compatible con aplicaciones escritas en el lenguaje Kotlin y que la dependencia `dex2jar` no funciona con

aplicaciones multidex. No observamos ninguna de estas limitaciones en la práctica (ver sección 5.7). Podemos suponer que existirán limitaciones intrínsecas a la utilización de `apktool`, similar a lo que ocurre con ACVTool.

5.3. Causas de falla en la instrumentación

A continuación detallamos las causas de los fallos durante la fase de instrumentación encontradas. La caracterización de las mismas la realizamos haciendo un análisis semi-automático de los *logs* de instrumentación para encontrar los grupos principales, seguido de una posterior evaluación más exhaustiva de los mismos.

- **Ella**

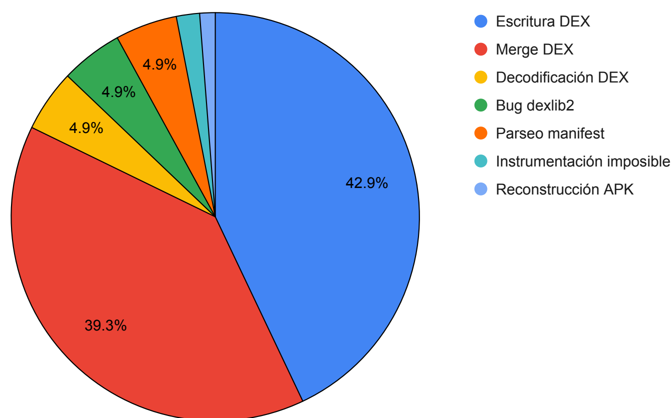


Fig. 5.7: Causas de falla en la instrumentación con Ella

La instrumentación con Ella (descripta en la sección 3.2) falla principalmente por dos motivos (ver figura 5.7):

- al intentar escribir el DEX instrumentado, más precisamente la información de *debugging* asociada, probablemente porque la inyección del fragmento para instrumentación no la ajusta
- al unir el DEX de soporte en runtime propio de Ella, pués si bien contiene unas pocas nuevas referencias (strings, campos, métodos) se termina superando los límites de 64K (por ejemplo en cantidad de métodos) en archivos DEX grandes.

Luego y en mucha menor cantidad de ocurrencias observamos 3 motivos más de falla significativos: la biblioteca utilizada `dexlib2` no soporta versiones recientes del formato DEX y falla acusando un archivo inválido[1]; la misma biblioteca contiene un error (ya solucionado en versiones posteriores[2]) que no interpreta correctamente información de *debugging*; y errores al parsear el manifiesto, muy probablemente debido a que para su decodificación Ella utiliza una versión antigua de `apktool`.

[1] `dexlib2` 2.0.3 incluída en Ella admite hasta la versión 036, pero hemos observado archivos DEX en versiones 037 o posteriores

[2] `dexlib2` 2.1.3 incluye el commit `https://github.com/JesusFreke/smali/commit/c347e68b46da4d78d1f5c7a3704ee59aa5e87ef5`, pero Ella utiliza la versión 2.0.3.

Únicamente en 3 casos el fallo de instrumentación reportado es porque no es posible la instrumentación, porque el método en cuestión no tiene registros disponibles para la utilización del *probe*.

- **ACVTool**

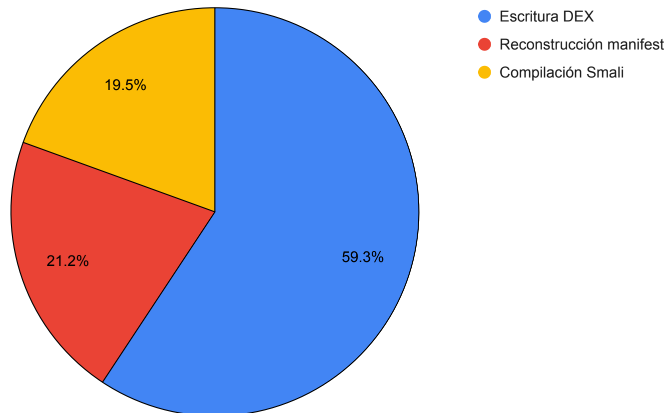


Fig. 5.8: Causas de falla en la instrumentación con ACVTool

Para ACVTool (ver sección 3.3) la gran mayoría de los casos de fallos en la instrumentación se deben a 3 causas (ver figura 5.8), en orden de ocurrencia:

- falla la escritura de los archivos DEX a partir del código Smali instrumentado
- falla la reconstrucción del manifiesto
- falla la compilación del Smali instrumentado

El fallo en la escritura del DEX se debe a que se intenta escribir un operando de 16 bits que no cabe en ese tamaño. Esto es porque la instrumentación agrega nuevas referencias que no estaban presentes en el DEX original, y probablemente también ocasiona un reordenamiento en las mismas. El formato Dalvik admite para algunas instrucciones operandos de mayor tamaño (modo Jumbo) pero no está siendo seleccionado correctamente para las instrucciones que así lo necesitan.

La reconstrucción del manifiesto la realiza la herramienta `apktool`. Encontramos varios problemas en esta fase también en COSMO (como veremos más adelante) y las razones parecen deberse a que el proceso de decodificación del manifiesto en formato binario no es realizado correctamente. Luego entonces es imposible volver a recodificar el manifiesto luego de ser modificado por ACVTool. Esto es una limitación en la implementación de `apktool`.

La tercera razón de fallos se debe a un procesamiento incorrecto por parte de ACVTool del código Smali. Encontramos dos errores que se repiten en todos estos casos: un manejo incorrecto de la directiva `.source` y la escritura de operaciones con registros que no pueden ser codificados en 8 bits. Cabe aclarar que esta segunda familia de casos es una de las limitaciones conocidas *a priori* para la herramienta.

- **COSMO**

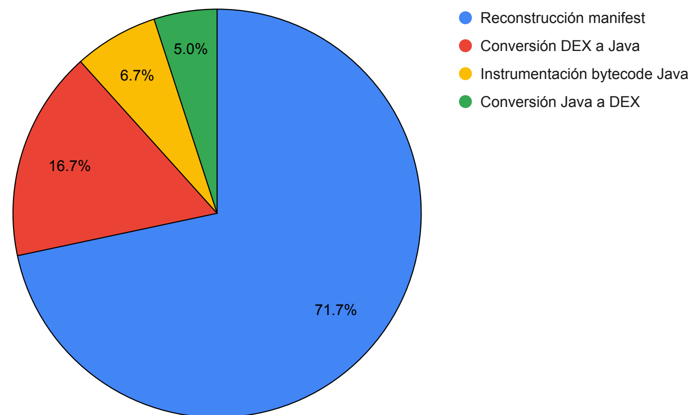


Fig. 5.9: Causas de falla en la instrumentación con COSMO

La inmensa mayoría de los fallos de instrumentación con COSMO (ver figura 5.9) se deben a errores durante la reconstrucción de manifiesto. Este problema ya descripto más arriba en los fallos de ACVTool se debe a limitaciones de la herramienta `apktool`. Cabe aclarar aquí que la introducción de `apktool` forma parte de las modificaciones realizadas a COSMO para lograr obtener resultados relevantes para el experimento (ver sección 3.5).

En menor cantidad, la segunda causa de fallo ocurre durante la conversión del formato DEX a bytecode Java (parte de la estrategia de instrumentación de COSMO, ver sección 3.4), y se debe exclusivamente al agotamiento de memoria asignada a la herramienta `dex2jar`.

Luego en tercer lugar ocurren errores al instrumentar el bytecode Java, posiblemente por limitaciones de la biblioteca JaCoCo. Y por último en 3 casos la causa de fallo es un error durante la conversión de bytecode Java a DEX, presumiblemente debido a una utilización incorrecta de la herramienta `dx`.

5.4. Resultados de medición de cobertura

Presentaremos y analizaremos los resultados de medición de cobertura para cada versión de Android por separado. Esto implica que el conjunto base de APKs válidos es diferente para cada versión (ver figura 5.2) como ya mencionamos al inicio del capítulo.

En la tabla 5.2 encontramos los resultados finales de los APKs instrumentados por cada herramienta para los cuales el experimento se pudo ejecutar y finalizar correctamente según lo descripto en la sección 4.2.

Como se puede apreciar, en esta segunda fase la herramienta que funciona mejor es Ella a pesar de haber sido la que menor cantidad de APKs logra instrumentar. Esto tiene que ver con las estrategias elegidas por cada herramienta para instrumentar y reportar los resultados. Ella utiliza una estrategia muy conservadora: modifica directamente el bytecode, los *probes* son pequeños, y no necesita modificar el manifiesto. Analizaremos en la sección 5.6 las causas concretas de fallas en esta etapa con mayor detalle, pero la mayor sofisticación de ACVTool y COSMO influyen en las mismas.

Herramienta	Mediciones exitosas	
	Android 7	Android 11
Ella	5 de 6	3 de 7
ACVTool	32 de 43	2 de 51
COSMO	31 de 82	32 de 95

Tab. 5.2: Mediciones de cobertura correctas para las 3 herramientas sobre APKs instrumentados del conjunto base, para las dos versiones de Android utilizadas.

Asimismo vemos que los resultados son mejores para Android 7 que para Android 11 para Ella y ACVTool. Esto es esperable ya que la versión 11 impone requerimientos adicionales que no se conocían al momento de desarrollar las herramientas. COSMO parece sufrir de este problema en menor medida.

Hay sólo 2 APKs de los 4 que lograron instrumentar las tres herramientas que además finalizaron correctamente el experimento. Para ellos analizaremos con mayor detalle los resultados de la cobertura medida.

Comparación de cobertura medida

En la tabla 5.3 se pueden observar los resultados de cobertura para los únicos 2 APKs que fueron medidos correctamente por las tres herramientas.

Aplicación	Métodos cubiertos / totales		
	Ella	ACVTool	COSMO
com.MiANSES	298/1023	301/1023	276/915
com.sube.app	199/4214	199/4214	199/4038

Tab. 5.3: Comparación de cobertura obtenida en los APKs con las tres herramientas en Android 7

En primer lugar cabe la siguiente aclaración: Ella excluye en su configuración predeterminada algunas clases consideradas de sistema (por ejemplo, todas las clases pertenecientes a los paquetes Java bajo el prefijo `android.*`), pero esto no es así para ACVTool y COSMO. Para hacer la comparación consistente, todos los métodos excluídos por Ella fueron también excluídos en este análisis para ACVTool y COSMO.

En segundo lugar, se puede observar que las cantidades totales de métodos detectados por Ella y ACVTool coinciden, pero son diferentes al número detectado por COSMO (que son siempre menores). Esto se explica porque COSMO utiliza JaCoCo, que es una herramienta que funciona sobre bytecode Java, y por consiguiente instrumenta y mide métodos Java. Los métodos sintéticos [14] creados durante la compilación a bytecode (por ejemplo para el acceso a campos privados desde clases anidadas) nos son considerados por JaCoCo, pero son métodos que luego se transforman a Dalvik y por consiguiente son instrumentados por Ella y ACVTool.

Por último, a pesar de que los estímulos son siempre los mismos para cada aplicación (ie. la semilla de `monkey` es constante y única para cada APK, independientemente de la

herramienta utilizada), se observan muy pequeñas diferencias en la cantidad de métodos cubiertos en una de las aplicaciones. Esto puede deberse a ligeras variaciones en las condiciones de ejecución de las distintas pruebas que pueden influir en los tiempos de respuesta de la aplicación. Y por supuesto a que el código ejecutado resultante de la instrumentación con cada herramienta es diferente.

5.5. Limitaciones *a priori* en la medición de cobertura

De antemano sabemos que la aplicación de las herramientas de instrumentación y la consecuente modificación de la integridad del APK puede traer problemas para instalar o ejecutar la aplicación. Por empezar el APK instrumentado debe ser firmado con un certificado diferente de aquel con el que fue publicado.

Google ofrece dos servicios de chequeo de integridad: Play Integrity [15] y SafetyNet Attestation API [19] que las aplicaciones pueden usar para verificar la integridad de la instalación. Si una aplicación utiliza alguno de estos servicios e interrumpe su ejecución (o inhabilita determinadas características) será imposible obtener un resultado consistente y completo de medición de cobertura.

5.6. Causas de falla en la medición de cobertura

Identificamos 3 momentos importantes en que puede fallar el experimento en esta fase: durante la instalación, mientras se está ensayando la aplicación con `monkey`, y al momento de obtener el reporte de cobertura.

Las fallas de instalación pueden deberse a:

- la instrumentación generó un APK inválido, o con una firma inválida
- el APK contiene código nativo pero no para la arquitectura de CPU donde estamos instalando la aplicación
- el sistema no alcanza los requerimientos mínimos requeridos por la aplicación

Sólo la primera causa la podemos atribuir a un error de la herramienta. En los demás casos el problema es intrínseco a la aplicación original, y por lo tanto el APK no debe considerarse válido. El paso inicial de validación detecta y elimina esta clase de fallos completamente.

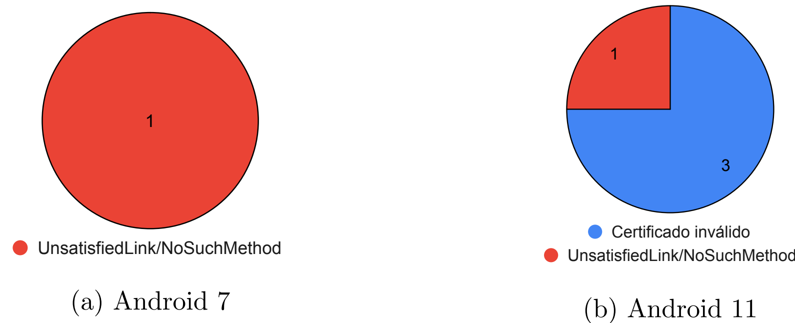
Los fallos durante el ensayo de la aplicación con `monkey` son atribuibles a:

- El proceso de instrumentación: si la instrumentación altera el código de tal forma de producir errores fatales o disparar chequeos de integridad de la aplicación, esto se verá reflejado en errores durante esta fase.
- Errores propios de la aplicación: no podemos descartar que la aplicación contenga errores de origen que produzcan fallas criticas desencadenadas por los eventos inyectados con la semilla elegida.
- El entorno de ejecución: ejercitamos las aplicaciones en un emulador y en una arquitectura que no es la usual en dispositivos móviles (aunque esto está mitigado en Android 11 con la emulación de ARM).

Nuevamente, nos interesa analizar las fallas originadas por la primera causa dado que las otras dos son debidas a problemas inherentes a la aplicación original y el entorno. Por lo tanto, estos casos son detectados y filtrados en el paso de validación.

Por último los fallos al obtener los resultados de cobertura son atribuibles casi exclusivamente a errores de la herramienta en sí.

- **Ella**



(a) Android 7

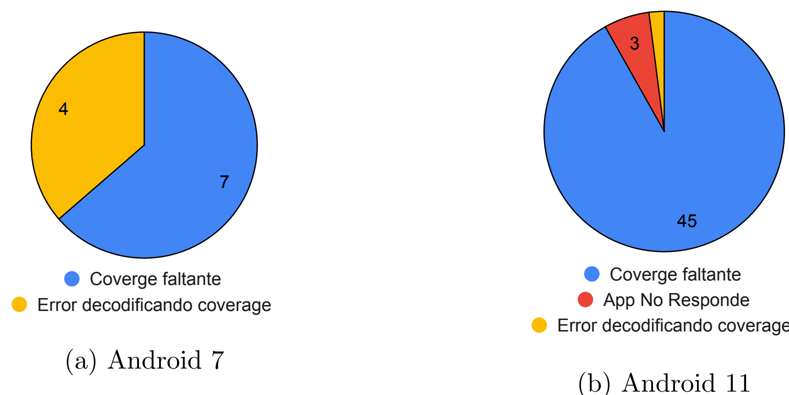
(b) Android 11

Fig. 5.10: Causas de falla en la ejecución de pruebas en APKs instrumentados con Ella

Para los APKs instrumentados con Ella (ver figura 5.10) observamos 3 casos de instalación fallida en Android 11 por un problema con la firma del APK. Esta falla se debe a que Ella firma los APKs instrumentados con `jarsigner` que crea una firma con versión 1 (firma JAR), pero Android 11 requiere firmas con el esquema de versión 2 o posterior [8]. En los tres casos, el *target SDK* indicado por los APKs es 30 (correspondiente a la versión Android 11) o superior, y por lo tanto el sistema operativo pone en vigor esta restricción.

La otra causa de falla es un error de excepción `java.lang.UnsatisfiedLinkError` que ocurre en la misma aplicación para ambas versiones de Android y es consecuencia del proceso de instrumentación.

- **ACVTool**



(a) Android 7

(b) Android 11

Fig. 5.11: Causas de falla en la ejecución de pruebas en APKs instrumentados con ACVTool

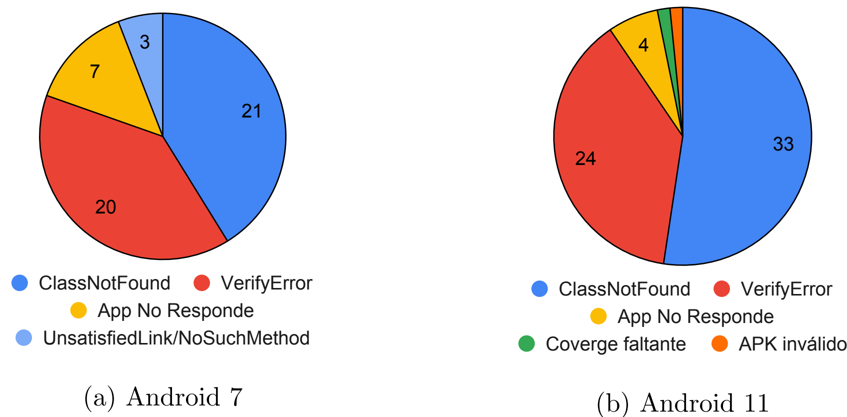
Con ACVTool la más importante causa de falla se produce al intentar obtener el reporte de cobertura (ver figura 5.11). En Android 7 observamos que representa casi

dos tercios del total y no está clara la razón del fallo: el archivo de cobertura no existe cuando la herramienta intenta obtenerlo. La otra causa es por errores al decodificar el archivo de resultados, y se debe a errores de programación de la herramienta.

También para Android 11, la gran mayoría de los fallos ocurren al momento de intentar obtener el reporte de cobertura. Sin embargo, en este caso el problema es inherente al funcionamiento de ACVTool, que intenta escribir un archivo en el almacenamiento externo del dispositivo. Para hacer esto asume que la ruta al mismo es `/mnt/sdcard/PACKAGE_NAME`, pero en Android 11 ya no es posible el acceso irrestricto al almacenamiento externo, independientemente del permiso `WRITE_EXTERNAL_STORAGE` que ACVTool introduce en el manifiesto de la aplicación [21].

El resto de las causas son minoritarias, pero vemos un error en la decodificación del archivo de resultados de cobertura (igual que antes por un error en la programación de ACVTool) y la aplicación deja de responder en los otros casos, mientras se está ejercitando con `monkey`.

- **COSMO**



(a) Android 7

(b) Android 11

Fig. 5.12: Causas de falla en la ejecución de pruebas en APKs instrumentados con COSMO

En los APKs instrumentados por COSMO las causas de fallo en la medición son un poco más variados (ver figura 5.12).

Por un lado para Android 7, observamos que más de tres cuartos de los fallos se deben a dos clases de error por excepciones fatales: `ClassNotFoundException` y `VerifyError`. El error de verificación es indicativo de un problema en el bytecode de la aplicación. Es decir que el proceso de transformación que sufre durante la instrumentación introduce algún tipo de error que hace fallar al runtime de Android. Algo similar ocurre con los errores sobre clases no encontradas: el procedimiento de instrumentación no procesa todas las clases originales y hay faltantes en el APK instrumentado. Este error es causado por la herramienta `dex2jar`. El resto de los fallos se producen durante la etapa de ensayo de la aplicación, y parecen ser también debidos a problemas introducidos por la instrumentación del código.

Para Android 11, observamos una situación similar con los fallos por las excepciones `ClassNotFoundException` y `VerifyError` significando casi el 90 % de las causas, y otros porque la aplicación deja de responder. Hay 2 casos adicionales de interés: en

uno el archivo de resultados de cobertura no se genera, y el otro es debido a que el APK usa una característica *anti-tampering* introducida en Android 10 [18] que `apktool` no considera al reconstruir el APK.

5.7. Correlación con la caracterización de los sujetos

Instrumentación

En la tabla 5.4 mostramos los porcentajes de éxito en la instrumentación según los distintos rasgos de caracterización con los que clasificamos los APKs.

Característica		APKs	Instrumentación exitosa		
			Ella	ACVTool	COSMO
MultiDEX	NO	23	22 %	65 %	70 %
	SI	147	1 %	29 %	64 %
Kotlin	NO	95	5 %	29 %	57 %
	SI	75	3 %	39 %	75 %
Código Nativo	NO	38	8 %	45 %	84 %
	SI	132	3 %	30 %	59 %
Firebase	NO	29	14 %	45 %	55 %
	SI	141	2 %	31 %	67 %
Google Play Services	NO	19	16 %	42 %	47 %
	SI	151	3 %	32 %	67 %
Flutter	NO	167	4 %	34 %	65 %
	SI	3	0 %	33 %	33 %
React Native	NO	163	4 %	33 %	64 %
	SI	7	0 %	43 %	71 %
Superpack	NO	167	4 %	32 %	65 %
	SI	3	0 %	100 %	67 %

Tab. 5.4: Comparación del éxito en la instrumentación según caracterización de los APKs.

Analizando estos números para diferencias de un 20 % o más, podemos observar que tanto Ella como ACVTool son muy sensibles a la característica multidex. Para el caso de Ella era algo esperable dada la estrategia de instrumentación (ver sección 3.2). El soporte para APKs multidex no estaba contemplado en la versión original de la herramienta y precisa hacer modificaciones aditivas a cada archivo `.dex` que no siempre son posibles dadas las limitaciones del formato. Algo similar ocurre con ACVTool (ver sección 3.3), que tiene la misma limitación, ya que a pesar de decompilar a Smali no altera la disposición de las clases dentro de los archivos `.dex`. COSMO (ver sección 3.4) por otro lado no se ve afectado por esta característica ya que reconstruye por completo los `.dex` al recompilar los archivos de bytecode `.class` y puede reubicar las clases y crear `classesN.dex` adicionales.

El rasgo identificado como Kotlin (ie. si todo o parte del código de la aplicación fue escrito en el lenguaje Kotlin) no presenta sesgos significativos para Ella o ACVTool. Con

COSMO observamos una ligera mejora en el resultado cuando el rasgo es detectado. Esto es una sorpresa, considerando que el paper [27] menciona que JaCoCo no es compatible con el lenguaje Kotlin.

La inclusión de código nativo en los APKs no parece afectar a Ella o ACVTool, pero si a COSMO negativamente. No encontramos una razón aparente que pueda ocasionar esta correlación. En el sentido opuesto observamos que el uso de las biblioteca Google Play Services parece mejorar la eficacia de COSMO, aunque nuevamente no encontramos una razón para explicar este resultado. Este rasgo no parece afectar a Ella o ACVTool, como tampoco lo hace la utilización de Firebase con ninguna de las herramientas.

Por último, desestimamos los sesgos por el uso de Flutter, React Native o Superpack ya que la cantidad de APKs afectados no es significativa.

Medición de cobertura

En las tablas 5.5 y 5.6 listamos los éxitos en la medición de la cobertura clasificados según las caracterizaciones de las aplicaciones, para Android 7 y Android 11 por separado. Omitimos las características Flutter, React Native y Superpack ya que la cantidad de APKs en los conjuntos de base no son representativas y por lo tanto no consideramos que se pueda extraer ninguna conclusión del análisis de esos resultados.

Característica		Medición exitosa en Android 7			
		APKs	Ella	ACVTool	COSMO
MultiDEX	NO	18	75 % de 4	91 % de 11	54 % de 13
	SI	103	100 % de 2	69 % de 32	35 % de 69
Kotlin	NO	73	75 % de 4	65 % de 20	47 % de 43
	SI	48	100 % de 2	83 % de 23	28 % de 39
Código Nativo	NO	35	100 % de 3	93 % de 15	45 % de 29
	SI	86	67 % de 3	64 % de 28	34 % de 53
Firebase	NO	24	67 % de 3	83 % de 12	60 % de 15
	SI	97	100 % de 3	71 % de 31	33 % de 67
Google Play Services	NO	14	100 % de 2	100 % de 7	88 % de 8
	SI	107	75 % de 4	69 % de 36	32 % de 74

Tab. 5.5: Comparación del éxito en la medición de cobertura según caracterización de los APKs para Android 7. Los porcentajes son en base al número de APKs instrumentados por cada herramienta del conjunto dado por la caracterización.

Los resultados para Ella están incluídos por completitud, pero dada la bajísima cantidad de APKs instrumentados correctamente y que pueden ser ejercitados, no consideramos que corresponda extraer ninguna conclusión de los mismos.

Para ACVTool en Android 7 parece haber una correlación negativa entre la caracterización multidex y el éxito del experimento. Algo similar ocurre con la utilización de librerías de código nativo y los Google Play Services. A priori no encontramos una razón aparente para ello, excepto la observación trivial que a más sencilla la *app*, más probable el éxito de medir su cobertura. En Android 11 no podemos concluir nada pués como

Característica		Medición exitosa en Android 11			
		APKs	Ella	ACVTool	COSMO
MultiDEX	NO	22	60 % de 5	14 % de 14	60 % de 15
	SI	129	0 % de 2	0 % de 37	29 % de 80
Kotlin	NO	87	40 % de 5	7 % de 27	47 % de 49
	SI	64	50 % de 2	0 % de 24	20 % de 46
Código Nativo	NO	32	67 % de 3	7 % de 15	48 % de 27
	SI	119	25 % de 4	3 % de 36	28 % de 68
Firebase	NO	25	50 % de 4	17 % de 12	69 % de 13
	SI	126	33 % de 3	0 % de 39	28 % de 82
Google Play Services	NO	18	67 % de 3	25 % de 8	89 % de 9
	SI	133	25 % de 4	0 % de 43	28 % de 86

Tab. 5.6: Comparación del éxito en la medición de cobertura según caracterización de los APKs para Android 11. Los porcentajes son en base al número de APKs instrumentados por cada herramienta del conjunto dado por la caracterización.

mencionamos en la sección 5.4 la gran mayoría de los APKs fallan por un problema en la implementación de la obtención del reporte de cobertura.

Analizando los resultados para COSMO, observamos una correlación negativa en el éxito de la medición con casi todas las caracterizaciones, y para ambas versiones del sistema operativo. En particular la utilización de Firebase y Google Play Services son las características que mayor diferencia en el éxito presentan según si son detectadas o no. En el análisis exhaustivo de las causas de fallo de medición (ver sección 5.6) encontramos que muchos de los errores por excepción `ClassNotFound` mencionaban clases relacionadas con estos servicios, con lo cual la correlación antes mencionada no es sorpresiva. En menor medida, la detección de Kotlin tiene una correlación negativa con el éxito del experimento, y esto es consistente con la advertencia que hacía el *paper* de COSMO. Sin embargo, la advertencia apuntaba a JaCoCo como el causante, pero por el análisis que realizamos de los resultados, la herramienta `dex2jar` parece el causante más probable de esta correlación. Por último mencionaremos que no encontramos razón aparente para la correlación con la caracterización multidex, más allá de la trivialidad antes mencionada sobre la complejidad de las aplicaciones.

6. CONCLUSIONES

Observamos que todas las herramientas presentan problemas para realizar la instrumentación y la obtención de la medición de cobertura de código. Considerando que la herramienta más reciente, COSMO, fue publicada a fines de 2020, el tiempo de utilidad de las mismas parece ser muy breve. Consideramos que esto tiene que ver fundamentalmente con tres aspectos: la velocidad de evolución la plataforma, la dificultad inherente de la tarea que se ve empeorada por una escasa disponibilidad de documentación del funcionamiento interno del sistema y sus componentes, y por último la poca aplicación industrial que redunda en falta de interés.

Esta observación se ve respaldada también por la diferencia en la eficacia de las distintas herramientas, que es inversamente proporcional a su antigüedad. Así es que Ella, que es del 2016 (con modificaciones en el 2018) tiene un desempeño sumamente bajo, superado marginalmente por ACVTool, del 2018, pero aún con rendimiento poco satisfactorio (instrumentando sólo un tercio de las aplicaciones, pudiendo finalizar el experimento en un número aún menor), y seguido finalmente por COSMO que a pesar de ser una herramienta relativamente moderna cumple el objetivo de instrumentación sólo para dos tercios de los APKs, y su eficacia cae abruptamente si consideramos el experimento completo.

La mayoría de las limitaciones que observamos en la fase de instrumentación, tienen que ver con defectos o limitaciones de implementación, y no son inherentes a la estrategia de instrumentación elegida. Esto aplica para las tres herramientas. Es más, algunas de las limitaciones a las estrategias de instrumentación planteadas en los papers ya no son relevantes. Por ejemplo, a partir de la adopción del ART (Android Runtime) en versiones modernas de Android, el soporte multidex es nativo a la plataforma, y las clases se pueden reubicar sin mayores problemas en nuevos archivos `classesN.dex` para evitar las limitaciones del formato. Sólo COSMO aprovecha esta libertad pero en forma indirecta, y debido a la estrategia de instrumentación seleccionada.

Sobre el *overhead* de instrumentación, dado que nuestro experimento se limitó a la granularidad de método, no observamos que sea un problema por la baja densidad de los *probes* insertados. Aún así, todas las estrategias seleccionadas pueden ser implementadas correctamente y no introducen una limitante real para las aplicaciones estudiadas.

Respecto de la segunda fase del experimento, la medición y obtención del reporte de cobertura, la observación inmediata es que la estrategia elegida por Ella muestra el mejor desempeño para el tipo de experimento desarrollado en esta tesis. La simpleza en el método (enviar los datos periódica y automáticamente mediante un *socket* TCP) brinda sus frutos otorgando el mejor porcentaje de éxito respecto de las aplicaciones instrumentadas. Sin embargo, como mencionamos antes, el número total de casos exitosos de instrumentación es el más bajo de las tres herramientas, con lo cual este buen resultado podría no mantenerse aumentando el conjunto de base.

Tanto para ACVTool como para COSMO la necesidad de introducir un *BroadcastReceiver* para controlar la emisión del reporte de cobertura implican que se debe modificar el manifiesto del APK, lo cual termina siendo problemático en ambos casos y el origen de un alto número de fallos de la fase de instrumentación. Esto es por limitaciones o defectos en la implementación de la dependencia que realiza esta tarea: `apktool`. En buena parte esto se puede atribuir a que debe manipular un archivo en un formato sin especificación

formal.

Por último, las tres herramientas sufren de problemas relacionados con la utilización de características o funcionalidad que fueron obsoletizados por la evolución de la plataforma: firmas digitales en formato perimido, acceso a recursos restringidos, modificaciones en el esquema de permisos del sistema, y otros motivos.

Por todo lo expuesto debemos concluir que en la actualidad no existe una buena solución para el objetivo planteado al inicio de esta tesis. El testeo de caja negra de aplicaciones comerciales y su correlación con cobertura de código es un área de carácter fundamentalmente académico, y por lo tanto no cuenta con un interés comercial que pueda financiar un desarrollo intensivo y continuo de las herramientas para realizarlo.

6.1. Trabajo futuro

Las acciones más inmediatas que consideramos se pueden realizar para intentar mejorar la eficacia de las herramientas consisten en actualizar y mejorar las dependencias utilizadas por las mismas. Como dijimos, si bien hay limitaciones intrínsecas a las estrategias de instrumentación elegidas, en la práctica la mayoría de los fallos ocurren por defectos y limitaciones en la implementación.

Algunos de los fallos de instrumentación para Ella se deben a defectos en `dexlib2` que ya fueron solucionados en versiones posteriores. Actualizar dicha dependencia podría solucionar algunos de estos fallos. La gestión de la información de *debugging* de los `.dex` también parece susceptible a errores en la inserción de los *probes* de instrumentación. No nos resulta clara la razón de este problema, pero creemos que tiene solución también realizando mejoras en `dexlib2`.

La principal causa de fallos de instrumentación en COSMO son debidos a errores en la manipulación del *manifest* por la dependencia `apktool`. Mejoras y soluciones a defectos en esta dependencia podrían eliminar buena parte de estos fallos. De la misma manera, encontramos que muchos errores durante la fase de ejecución con APKs instrumentados por COSMO se deben a limitaciones en la transformación inicial de archivos `.dex` a `.jar` que realiza la dependencia `dex2jar`. Eliminar esas limitaciones podría mejorar sustancialmente los resultados para COSMO.

El siguiente paso en profundidad implica revisar ligeramente las estrategias y realizar algunos cambios en ese sentido. Para Ella y ACVTool que modifican el bytecode DEX inyectando nuevo código, entendemos que es posible solucionar los problemas relacionados a este paso realizando una redistribución de clases en distintos archivos `.dex`. Esto se puede hacer pues las versiones modernas del sistema Android manejan nativamente las aplicaciones multidex. Con este cambio debería ser posible evitar las limitaciones inherentes al formato. En esa misma línea, en el caso de Ella ya no sería necesario inyectar un *wrapper* diferente en cada archivo `classesN.dex`. En ambos casos es necesario implementar nueva funcionalidad para referenciar clases externas en cada `.dex`.

En la muestra de APKs obtenida no encontramos muchos casos que utilicen tecnologías alternativas cuyo código de aplicación no se compile a bytecode. Nos referimos concretamente a Flutter y React Native, aunque existen otras que no intentamos caracterizar, como por ejemplo Cordova, Ionic, etc.. Realizar una medición de cobertura completa en APKs con cualquiera de estas tecnologías requiere estrategias especializadas para cada una. Por ejemplo, para React Native necesitaríamos instrumentar el motor Javascript para obtener la cobertura del código que éste ejecuta. Para Flutter necesitaríamos poder instrumentar

código nativo.

Afortunadamente, la mayoría de las aplicaciones siguen programándose en lenguajes que compilan a bytecode. Sí debemos observar que Kotlin como lenguaje *frontend* ha ganado mucha popularidad, con lo cual es sumamente importante asegurar una buena compatibilidad de las herramientas de instrumentación con el bytecode que genera el compilador de Kotlin.

Y como última idea, aunque ya bastante lejos de las herramientas estudiadas, aproximadamente un cuarto de los APKs obtenidos tienen un 30 % o más de código nativo (medido en bytes). Si bien esta proporción es baja, sería útil poder instrumentar y medir cobertura del mismo. Las dificultades para hacer esto en un escenario de caja negra son muchas y muy grandes, pero de poder desarrollarse alguna solución, sería factible estudiar su aplicación a la otra plataforma móvil prevalente iOS, donde no hay un *runtime* que ejecute bytecode.

Apéndice

A. LISTADO DE APKS

Tab. A.1: APKs descargados de Play Store, región Argentina, Top Gratis, 01/04/2022

App ID	Versión	SDK	MX	NA	KT	FB	PL	FL	RN	SP	Válido
app.source.getcontact	5.7.0	21/30	✓	✓		✓	✓				✓
ar.burgerking	4.2.0	21/30	✓	✓		✓	✓				✓
ar.com.bancoprovincia.CuentaDNI	6.3.9.44290	19/29	✓	✓			✓				
ar.com.cablevision.attv.android.myminerva	3.56.21	23/30	✓	✓	✓	✓	✓				✓
ar.com.claro.android	307	19/30									✓
ar.com.personal	8.6.10	21/30	✓	✓	✓	✓	✓				✓
ar.com.personalpay	0.1.835	21/30	✓	✓	✓	✓	✓		✓		✓
ar.com.progresar	1.0.25	21/29		✓		✓	✓				
ar.com.santander.rio.mbanking	3.71.5	19/29	✓	✓	✓	✓	✓				7
ar.gob.afip.mobile.android.contribuyentes.mi_afip	2.4.3	19/28	✓	✓		✓	✓				✓
ar.gob.argentinagobar	5.3.3	21/30	✓	✓	✓	✓	✓				
ar.gob.coronavirus	3.5.32	21/30		✓		✓	✓				✓
ar.onvideo	v8.1.	23/29	✓	✓	✓	✓	✓				✓
ar.openbank.modelbank	1.5.0	21/30	✓	✓		✓	✓				✓
ar.org.pami.app	2.1.0	21/31	✓								✓
bef.ayp.idea	1.1	22/30	✓			✓	✓				7
bloodpressure.bloodpressureapp.bloodpressuretracker	1.2.2	21/30			✓	✓	✓				
cn.xiaofengkj.fitpro	1.9.1	21/30	✓	✓		✓	✓				11
co.brainly	5.79.0	21/30	✓	✓		✓	✓				✓
com.accurate.local.weather.forecast.live	1.2.4	17/30	✓	✓	✓	✓	✓				✓
com.adobe.reader	22.3.0.21685.B	24/31	✓	✓	✓	✓	✓				11
com.agminstruments.drumpadmachine	2.14.0	21/30	✓	✓							✓
com.alibaba.aliexpresshd	8.44.0	21/30	✓	✓	✓	✓	✓				
com.amazon.avod.thirdpartyclient	3.0.303.19247	21/29	✓	✓	✓	✓	✓				11
com.ankapp	1.15.1	24/31	✓	✓	✓	✓	✓		✓		✓
com.antivirus	6.47.0	23/30	✓	✓		✓	✓				✓
com.antivirus.mobilesecurity.viruscleaner.applock	1.5.3	19/31	✓	✓		✓	✓				✓
com.antivirusmaster.oneboostcleaner	1.2.0	19/30	✓	✓		✓	✓				✓
com.applemoncash	2.6.23	21/30	✓	✓	✓	✓	✓		✓		✓
com.banconacion.bnamas	6.4.3.44448	19/28	✓	✓			✓				✓
com.bbva.nxt_argentina	2.8.2	22/29	✓	✓	✓	✓	✓				✓
com.bettertec.ravo.app	1.0.5	21/31	✓	✓		✓	✓				11
com.bigwinepot.nwdn.international	2.1.1.20211276	23/30	✓	✓		✓	✓				7
com.brave.browser	1.24.86	24/30	✓	✓							
com.brubank	1.38.4	21/29	✓	✓	✓	✓	✓				11
com.cabify.rider	8.27.0	21/31	✓	✓		✓	✓				✓
com.camerasideas.instashot	1.815.1352	21/30	✓	✓		✓	✓				11
com.candoit.mostaza	4.2.2	15/30	✓		✓	✓	✓				11
com.cbs.ca	12.0.16	21/30	✓	✓	✓	✓	✓				✓
com.cleanmaster.ultra	1.2.5	19/29	✓		✓	✓	✓				
com.cordial.iudu	2.8.4	23/30	✓	✓	✓	✓	✓				✓
com.crunchyroll.crunchyroid	3.7.0	23/29	✓	✓		✓	✓				✓
com.descargar.musica.gratismp3	15	21/30		✓		✓	✓				✓
com.descargarmusica.abb	1.0.3	16/30	✓	✓	✓	✓	✓				✓
com.didiglobal.passenger	7.2.78	21/30	✓	✓	✓	✓	✓	✓			
com.directv.dtvlatam	2.25.0	21/29	✓		✓	✓	✓				
com.disney.disneyplus	2.2.0-rc5	21/30	✓			✓	✓				✓
com.disney.starplus	2.3.2-rc1	21/30	✓			✓	✓				11
com.documentreader.documentapp.filereader	2.1.9	21/29	✓	✓	✓	✓	✓				7
com.documentscan.simplescan.scanpdf	3.7.3	21/30	✓	✓	✓	✓	✓				7
com.dots.lpf	2.5.7	22/29	✓	✓		✓	✓				✓
com.duolingo	5.50.2	22/30	✓	✓		✓	✓				✓
com.dywx.larkplayer	5.5.15	18/29	✓	✓	✓	✓	✓				11
com.ecvilib.cleanerultra	1.14	21/31	✓		✓	✓	✓				✓
com.facebook.katana	358.0.0.34.117	28/30		✓		✓	✓			✓	
com.facebook.lite	251.0.0.4.119	15/29		✓						✓	11
com.facebook.mlite	140.0.0.5.118	14/29		✓		✓	✓				11

Tab. A.1: APKs descargados de Play Store (continuación)

App ID	Versión	SDK	MX	NA	KT	FB	PL	FL	RN	SP	Válido
com.facebook.orca	352.0.0.9.116	28/30	✓	✓		✓	✓			✓	
com.fastcandy.freeandroid	1.0.2	21/31	✓	✓		✓	✓				✓
com.fontskeyboard.fonts	4.5.0.17537	23/30	✓			✓	✓				✓
com.free.speedfiy	1.0.4	21/30	✓	✓		✓	✓				11
com.freedownloader.videosaver.hdvideodownloader	7	19/30	✓		✓		✓				✓
com.ftw_and_co.happn	25.31.0	21/29	✓	✓	✓	✓	✓				7
com.fullplay.zetaplay_momoplay	1	21/30	✓	✓	✓	✓	✓				
com.gamma.scan	2.2.21	16/30	✓			✓	✓				✓
com.gb.whtasappna.version022	1.2	21/31	✓				✓				✓
com.google.android.apps.adm	2.4.065	16/31									✓
com.google.android.apps.chromecast.app	2.49.1.8	23/31	✓	✓							✓
com.google.android.apps.classroom	7.3.141.04.45	21/29	✓	✓							11
com.google.android.apps.docs	2.22.117.0.all	23/31	✓	✓							
com.google.android.apps.docs.editors.docs	1.22.102.02.90	24/31	✓	✓							✓
com.google.android.apps.meetings	2022.03.06.433	23/31	✓	✓							✓
com.google.android.apps.subscriptions.red	1.96.371729603	22/30	✓	✓							11
com.google.android.apps.youtube.kids	7.10.3	21/31	✓	✓							✓
com.google.android.apps.youtube.music	4.70.50	21/31	✓	✓							7
com.google.android.youtube	17.11.35	23/31	✓	✓							
com.h2Osoft.videoeditor.videorecorder.screenrecorder	1.0.2	21/30	✓	✓	✓		✓				✓
com.hbo.hbonow	52.10.0.91	21/30	✓	✓		✓	✓				✓
com.ibragunduz.applockpro	3.1.3	21/29	✓		✓	✓	✓				7
com.instabridge.android	21.9.0.0325172	21/30	✓	✓		✓	✓				✓
com.instagram.android	188.0.0.35.124	23/30	✓	✓		✓	✓			✓	
com.instagram.lite	296.0.0.7.111	15/31		✓		✓	✓			✓	11
com.intsig.camscanner	6.12.0.2203030	21/30		✓							
com.kwai.video	5.2.0.511042	19/29	✓	✓	✓	✓	✓				
com.lemon.lvoverseas	5.6.0	21/30	✓	✓		✓	✓				
com.levine.easy.clean.boost	1.0.9	21/31	✓	✓		✓	✓				7
com.linecorp.b612.android	11.1.10	23/30	✓	✓		✓	✓				
com.linkedin.android	4.1.679.1	23/31	✓	✓		✓	✓				✓
com.litatom.app	3.9.2.1	21/31	✓	✓	✓	✓	✓				11
com.livehousex.drawing	1.1	24/31	✓		✓	✓	✓				
com.marsvard.stickermakerforwhatsapp	1.0.4-24	16/31	✓	✓		✓	✓				7
com.maximo.painelled	1.6	21/28			✓	✓	✓				✓
com.mcdo.mcdonalds	3.7.0	23/31	✓			✓	✓				✓
com.mcpe.bedrock.maps.Mods.Master	2.1	19/32	✓		✓	✓	✓				✓
com.meetyou.intl	1.3.1	21/30	✓	✓	✓	✓	✓				11
com.memeandsticker.textsticker	3.4.28.1	21/30	✓	✓	✓	✓	✓				11
com.mercadolibre	10.157.4	19/29	✓	✓		✓	✓				11
com.mercadopago.wallet	2.162.3	19/29	✓	✓		✓	✓				11
com.MiANSES	25.9.0	16/28									✓
com.microblink.photomath	7.8.1	21/30		✓	✓	✓	✓				✓
com.microsoft.office.officehubrow	16.0.15028.201	26/31	✓	✓	✓	✓	✓				11
com.microsoft.office.word	16.0.14931.200	26/30	✓	✓	✓	✓	✓				11
com.microsoft.skydrive	6.5	23/30	✓	✓	✓	✓	✓				11
com.minsaludpba.vacunatePBA	6.5.3.44955	19/29	✓	✓		✓	✓				✓
com.mufumbo.android.recipe.search	2.241.0.0-andr	21/31	✓			✓	✓				✓
com.musicplayer.playermusic	1.88.2	21/30	✓	✓		✓	✓				✓
com.naver.linewebtoon	2.9.1	19/30	✓			✓	✓				✓
com.netflix.mediaclient	8.22.	24/31	✓	✓		✓	✓				✓
com.offline.bible	4.0.5	21/30	✓			✓	✓				✓
com.outfit7.talkingben	3.8.0.28	19/29	✓	✓		✓	✓				11
com.pedidosya	6.3.9.0	21/30	✓			✓	✓				
com.phone.optimizer.tool.cleaner	1.0.28	19/30	✓	✓	✓	✓	✓				11
com.picsart.studio	19.4.0	23/31	✓	✓		✓	✓				
com.pinterest	10.10.0	24/30	✓	✓		✓	✓		✓		✓
com.plato.android	3.4.2	21/30	✓	✓		✓	✓				✓
com.playrix.fishdomdd.gplay	6.32.0	19/30	✓	✓	✓	✓	✓				✓
com.rarlab.rar	6.10.build104	19/31		✓							✓
com.recorder.screenrecorder.capture	3.1.3	21/30	✓	✓	✓	✓	✓				11
com.reportlyAnd	2.7.0	28/30	✓	✓	✓	✓	✓		✓		11
com.resultadosfutbol.mobile	5.2.5	21/30	✓			✓	✓				✓
com.safe.antivirus.cleaner	1.3.5	21/30	✓	✓		✓	✓				

Tab. A.1: APKs descargados de Play Store (continuación)

App ID	Versión	SDK	MX	NA	KT	FB	PL	FL	RN	SP	Válido
com.sec.android.easyMover	3.7.28.7	14/31	✓	✓	✓		✓				✓
com.services.movistar.ar	12.0.5	21/30		✓	✓	✓	✓		✓		✓
com.seven.shang.jianqi.zui.kan.wallpaper	1.7	21/30	✓		✓		✓				7
com.shaiban.audioplayer.mplayer	v6.7.2	21/30	✓		✓	✓	✓				✓
com.shopee.ar	2.85.21	16/30	✓	✓		✓	✓				✓
com.simplemobiletools.gallery.pro	6.22.0	21/29	✓								✓
com.smartsecurityxzt	166	19/30	✓	✓	✓	✓	✓		✓		✓
com.snapchat.android	11.73.0.3	19/30	✓	✓							11
com.snowcorp.epik	3.0.1	23/30	✓	✓		✓	✓				11
com.snowcorp.stickerly.android	1.18.2	21/29	✓	✓	✓	✓	✓				11
com.sofascore.results	5.93.1	21/31	✓			✓	✓				✓
com.softinit.iquitos.mainapp	1.8.4	22/30	✓		✓	✓	✓				✓
com.speedfiymax.app	1.0.3	21/30	✓	✓		✓	✓				11
com.splendapps.voicerec	3.16	19/30				✓	✓				✓
com.spotify.music	8.6.4.971	16/29	✓	✓	✓	✓	✓				11
com.starcleaner	2.2.7	21/30	✓	✓	✓	✓	✓	✓			11
com.starmakerinteractive.starmaker	7.9.8	18/29	✓	✓	✓	✓	✓	✓			11
com.start.kaishi.dian.clice.shou.gou.enouge.wallpaper	1.4	21/30	✓		✓		✓				7
com.streema.simpleradio	4.8.0	19/30	✓		✓	✓	✓				
com.sube.app	20.06.01.0	15/28		✓							✓
com.sube.cargasube	1.5.10b	18/31		✓	✓	✓	✓				✓
com.supertower.speedfast	1.0.1	23/31		✓		✓	✓				✓
com.supervielle.fedevida	1.35.0	21/30	✓	✓	✓	✓	✓	✓			11
com.tarjetanaranja.ncuenta	3.17.1.3019	21/29	✓	✓	✓	✓	✓				✓
com.teacapps.barcodescanner	2.7.5-L	30/30				✓	✓				11
com.themausoft.wpsapp	1.6.57	16/29		✓		✓	✓				7
com.tinder	12.3.0	23/29	✓	✓		✓	✓				✓
com.tool.fast.smart.cleaner	1.1.3	19/30	✓		✓	✓	✓				
com.tranzmate	5.68.0.482	16/29	✓	✓	✓	✓	✓				
com.twitter.android	9.35.1-release	21/31	✓	✓		✓	✓				✓
com.ubercab	4.414.10002	21/30	✓	✓	✓	✓	✓				11
com.vast.vpn.proxy.unblock	2.2.1	21/30	✓	✓		✓	✓				11
com.videoeditorpro.android	2.1.6	21/29	✓	✓	✓	✓	✓				
com.whatsapp	2.21.9.15	16/29	✓	✓		✓	✓			✓	11
com.whatsapp.w4b	2.22.7.11	16/30	✓	✓		✓	✓			✓	
com.wkolarbizar.hyfhbZVn2010.is4you2014	1	16/29	✓								✓
com.ypf.jpm	3.5.5-release	21/30	✓	✓	✓	✓	✓				✓
com.zhiliao.musically.livewallpaper	22.4	16/30		✓							11
com.zhiliaoapp.musically	23.4.4	19/30	✓			✓	✓				
com.zhiliaoapp.musically.go	24.0.2	19/30	✓	✓		✓	✓				
com.ztnstudio.notepad	2.0.16921	19/30	✓	✓	✓	✓	✓				✓
cz.hipercalc	9.2.1	16/30	✓			✓	✓				✓
de.motain.iliga	14.27.0	23/30	✓	✓	✓	✓	✓				✓
fm.anchor.android	3.114.0	22/31	✓	✓		✓	✓				✓
free.daily.tube.background	3.1.52.003	19/30	✓	✓			✓				
io.walkietalkie	2.1.3	21/31	✓	✓		✓	✓				✓
itube.snaptube.videodervideoplayerall	VT.1.1.7	19/30	✓				✓				7
jp.ne.ibis.ibispaintx.app	8.1.1	16/29	✓	✓	✓	✓	✓				
kjv.bible.kingjamesbible	3.1.0	19/30	✓	✓	✓	✓	✓				7
me.bukovitz.noteit	1.0.4	21/31	✓			✓	✓				✓
me.pou.app	1.4.92	19/31	✓				✓				✓
mp3.tubeplay.descargar.musica.tube.music.downloader	1.0.7	21/30	✓	✓	✓	✓	✓				11
musicplayer.musicapps.music.mp3player	2.9.1.97	19/30	✓	✓	✓	✓	✓				11
net.dinglisch.android.taskerm	5.12.22	21/29			✓	✓	✓				✓
net.zedge.android	7.34.4	21/30	✓	✓		✓	✓				✓
org.prowl.torque	1.10.120	16/28		✓			✓				✓
org.telegram.messenger	8.1.2	23/29	✓	✓		✓	✓				11
pdfreader.pdfviewer.officetool.pdfscanner	1.26	21/31	✓	✓	✓	✓	✓				✓
phonecleaner.androidmaster.cleanupspace.phone.booste	1.0.20	21/30				✓	✓				✓
photo.editor.photoeditor.photoeditorpro	1.402.119	21/30	✓	✓	✓	✓	✓				11
photoeditor.layout.collagemaker	2.122.109	21/30	✓	✓	✓	✓	✓				7
qrcodescanner.barcodescanner.qrscanner.qrcodereader	1.1.8	21/30	✓	✓	✓	✓	✓				✓
telefe.app	5.0.3	21/30	✓	✓	✓	✓	✓				✓
tv.pluto.android	5.14.1	21/31	✓	✓		✓	✓				✓

Tab. A.1: APKs descargados de Play Store (continuación)

App ID	Versión	SDK	MX	NA	KT	FB	PL	FL	RN	SP	Válido
tv.twitch.android.app	12.7.0_BETA	21/30	✓	✓		✓	✓				✓
us.zoom.videomeetings	5.9.6.4756	21/30	✓		✓	✓	✓				
videoeditor.videorecorder.screenrecorder	2.2.0.6	21/30	✓	✓	✓	✓	✓				11
vpf.iwz.capitanplay	1	21/30	✓			✓	✓				✓
vpn.video.downloader	5.2.2	21/30	✓	✓	✓	✓	✓				✓

SDK = mínimo/objetivo, MX = MultiDEX, NA = Código nativo, KT = Kotlin
FB = Firebase, PL = Play Services, FL = Flutter, RN = React Native, SP = Superpack
Válido = 7/11 sólo esa versión de Android, ✓ambas versiones

Bibliografía

[1] Android Developers Blog: Run ARM apps on the Android Emulator. https://android-developers.googleblog.com/2020/03/run-arm-apps-on-android-emulator.html. Accedido el 06/07/2022.

[2] Android Developers Blog: Taking the final wrapper off of Android 7.0 Nougat. https://android-developers.googleblog.com/2016/08/taking-final-wrapper-off-of-nougat.html. Accedido el 06/07/2022.

[3] Android Runtime (ART) and Dalvik — Android Open Source Project. https://source.android.com/devices/tech/dalvik. Accedido el 09/08/2022.

[4] Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps. https://ibotpeaches.github.io/Apktool/. Accedido el 10/08/2022.

[5] App Manifest Overview — Android Developers. https://developer.android.com/guide/topics/manifest/manifest-intro. Accedido el 09/08/2022.

[6] Application Fundamentals — Android Developers. https://developer.android.com/guide/components/fundamentals. Accedido el 22/06/2022.

[7] Application Signing — Android Open Source Project. https://source.android.com/security/apksigning. Accedido el 09/08/2022.

[8] Behavior changes: Apps targeting Android 11 — Android Developers. https://developer.android.com/about/versions/11/behavior-changes-11#minimum-signature-scheme. Accedido el 12/08/2022.

[9] Dalvik bytecode — Android Open Source Project. https://source.android.com/devices/tech/dalvik/dalvik-bytecode. Accedido el 29/06/2022.

[10] DexPatcher/multidexlib2: Multi-dex extensions for dexlib2. https://github.com/DexPatcher/multidexlib2. Accedido el 09/08/2022.

[11] Flutter - Build apps for any screen. https://flutter.dev/. Accedido el 02/09/2022.

[12] Github JesusFrekesmali: smali/baksmali. https://github.com/JesusFreke/smali. Accedido el 29/07/2022.

[13] Instrumentation — Android Developers. https://developer.android.com/reference/android/app/Instrumentation. Accedido el 30/06/2022.

[14] Java Synthetic Methods — What are these? — by Vaibhav Singh — DataDrivenInvestor. https://medium.datadriveninvestor.com/java-synthetic-methods-what-are-these-52f77efb347a. Accedido el 04/08/2022.

[15] Overview of the Play Integrity API — Google Play — Android Developers. https://developer.android.com/google/play/integrity/overview. Accedido el 08/07/2022.

[16] pxb1988/dex2jar: Tools to work with android .dex and java .class files. `https://github.com/pxb1988/dex2jar`. Accedido el 10/08/2022.

[17] React Native · Learn once, write anywhere. `https://reactnative.dev/`. Accedido el 02/09/2022.

[18] Run embedded DEX code directly from APK — Android Developers. `https://developer.android.com/topic/security/dex`. Accedido el 29/08/2022.

[19] SafetyNet Attestation API — Android Developers. `https://developer.android.com/training/safetynet/attestation`. Accedido el 08/07/2022.

[20] SDK Platform release notes — Android Developers. `https://developer.android.com/studio/releases/platforms`. Accedido el 08/09/2022.

[21] Storage updates in Android 11 — Android Developers. `https://developer.android.com/about/versions/11/privacy/storage#app-specific-external`. Accedido el 12/08/2022.

[22] Superpack: Pushing the limits of compression - Engineering at Meta. `https://engineering.fb.com/2021/09/13/core-data/superpack/`. Accedido el 02/09/2022.

[23] Top Charts - Android Apps on Google Play. `https://play.google.com/store/apps/top?gl=AR`. Accedido el 01/04/2022.

[24] Turning it up to Android 11. `https://blog.google/products/android/android-11/`. Accedido el 06/07/2022.

[25] Where is Android binary XML format documented? - Reverse Engineering Stack Exchange. `https://reverseengineering.stackexchange.com/questions/21806/where-is-android-binary-xml-format-documented`. Accedido el 08/08/2022.

[26] Aleksandr Pilgun, Olga Gadyatskaya, Stanislav Dashevskyi, Yury Zhauniarovich, and Artsiom Kushniarou. An effective Android code coverage tool. In *CCS*, pages 2189–2191. ACM, 2018.

[27] Andrea Romdhana, Mariano Ceccato, Gabriel Claudiu Georgiu, Alessio Merlo, and Paolo Tonella. COSMO: code coverage made easier for Android. In *ICST*, pages 417–423. IEEE, 2021.

[28] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. An empirical study of Android test generation tools in industrial cases. In *ASE*, pages 738–748. ACM, 2018.