Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Comparación de herramientas de testing automático para detección de permisos en aplicaciones Android

Tesis presentada para optar al título de
Licenciada en Ciencias de la Computación

Pulice, Ignacio y Ravasi, Nicolás

Director: Juan Pablo Galeotti

Buenos Aires, 2017

COMPARACIÓN DE HERRAMIENTAS DE TESTING AUTOMÁTICO PARA DETECCIÓN DE PERMISOS EN APLICACIONES ANDROID

Una de las mayores preocupaciones tanto para los usuarios como para los desarrolladores de software es la seguridad. Cuando usamos un programa, no es trivial, incluso para un usuario avanzado, saber si dicho programa hace lo que dice y no más que eso. Poder aseverar esto sin acceder al código fuente implica probar todos los caminos posibles que puede tomar el programa, y está más allá de la capacidad de cualquier persona.

Ante esta imposibilidad, ha tomado relevancia el testing automático, donde es la misma computadora quien se encarga de probar diferentes secuencias de entrada para tratar de cubrir la mayor cantidad de comportamiento posible, ya sea para encontrar defectos, buscar comportamiento indebido o asegurar la correctitud del programa.

En Android, el sistema operativo más usado en la actualidad en el mundo, las aplicaciones deben como medida de seguridad declarar las funciones restringidas del sistema a las que desean acceder, a manera de permisos. Pero estos permisos están definidos de una manera demasiado amplia, por lo que no es fácil entender para qué son requeridos.

Como alternativa a este modelo, se creó un entorno de *sandbox*, **Boxmate**, en el cual se corren aplicaciones de forma segura, y así ejecutar funcionalidades restringidas que fueron previamente detectadas mediante testing automático. Para este fin se creó la herramienta **Droidmate**.

Pero cabe preguntarse si existen otras herramientas de testing automático que sean más efectivas a la hora de localizar el comportamiento restringido de una aplicación. Para tal fin, vamos a comparar a Droidmate con otras dos herramientas, **Monkey** y **Sapienz**.

A la vez, vamos a evaluar cual es la situación actual en materia de detección de comportamiento en Android, y si ésta nos permite aseverar que las aplicaciones son seguras luego de pasar dicha prueba.

Palabras claves: Android, detección de permisos, testing automático, Droidmate, Sapienz, Monkey, Mining Sandboxes.

AGRADECIMIENTOS

De parte de Ignacio:

Principalmente agradecer y dedicar este ultimo esfuerzo a mis padres Alicia y Pablo. Por haberme forjado como la persona que soy en la actualidad; muchos de los logros se los debo a ustedes, en los que incluyo este. Me formaron con reglas y ciertas libertades, pero al final de cuentas, me motivaron con constancia para alcanzar mis objetivos en la vida.

A mis amigos, los cuales me molestaron hasta el hartazgo con todo tipo de bromas para que termine, que le ponga pilas, y cierre un ciclo de una vez por todas.

A mis compañeros de la facultad por compartir todo este largo camino, de exámenes, tps, y findes y findes de estudio.

A mis compañeros de trabajo por bancarme los días de bajon, por las dudas eternas de donde iba esta tesis y alentarme en todo momento.

A mis profesores de la facultad, por todo lo aprendido durante estos años.

De parte de Nicolás:

A mi familia, porque sin ellos no hubiera llegado hasta acá

A mis amigos, por todo lo que compartí con ellos estos años

A todos los que me insistieron por años que terminara la carrera

A Alex, por haber hecho toda esta carrera mucho más amena

A Juli, por haber ayudado mucho a que podamos realizar la experimentación

A Nacho, por haber dado todo de sí en esta tesis, y por haber mantenido este barco a flote en muchas ocasiones

Y a Josh, por estar siempre junto a mí, soportarme durante este período y no haberme dejado que me rindiera

Índice general

1. INTRODUCCIÓN

1.1. Contexto

La presencia de dispositivos móviles ha incrementado notablemente con los años, como también el número de actividades que se puede hacer con ellos, siendo fundamental para nuestra vida y casi ofreciendo la misma funcionalidad que una computadora personal. Este crecimiento ha causado una proliferación de aplicaciones (apps). Como todo software, éstas deben ser testeadas correctamente para garantizar el correcto funcionamiento, evitando así que código malicioso ponga en peligro nuestros dispositivos y nuestra privacidad. Al instalar una aplicación en el teléfono, dado que en la mayoría de los casos no podemos acceder al código fuente, no tenemos forma de saber fehacientemente que una app haga lo que dice que va hacer, pudiendo contener funcionalidad indeseada o maliciosa que comprometa nuestra información personal.

Android es el sistema operativo más popular del mercado actualmente, con más de 2.8 millones de aplicaciones en su store (Google Play), por esta razón haremos foco en él. A diferencia de la App Store de Apple, Android es mucho más permisivo a la hora de dejar que desarrolladores suban sus creaciones en su mercado, trayendo aparejado un problema latente de seguridad para el usuario que descargue aplicaciones maliciosas. La forma que tiene Android de controlar la funcionalidad de las apps es mediante los permisos que solicita. Para que una aplicación pueda acceder a distintos recursos del celular necesita que el usuario le conceda esos permisos específicos. En el transcurso de los años, Android, ha mutado mucho en la administración de los mismos, y estos varían según la versión del sistema operativo (En la sección permisos profundizaremos más sobre esto).

Es entendible que por el número creciente de aplicaciones, Google Play como tantos otros stores que proveen aplicaciones al usuario, vayan incorporan nuevos mecanismos para la detección de aplicaciones maliciosas Este proceso está en constante evolución, ya que para poder garantizar un mejor análisis se necesitan herramientas que puedan testear a todas y cada una de las aplicaciones publicadas, y más aún, distintos tipos de versiones. Esto último generó que se empiece a aumentar la demanda de herramientas de testing automático, dado que estas ayudan en la detección tanto de fallas como de comportamiento no deseado o malicioso.

En este punto, existen distintas técnicas de testing automático, las cuales difieren en cómo generan sus inputs, las diferentes estrategias que utilizan para explorar el comportamiento de las apps bajo test y las heurísticas específicas que utilizan.

¿Pero cómo relacionamos permisos con testing automático?

Una herramienta de testing automático, a grandes rasgos, recorre la aplicación bajo test y genera pruebas automáticas donde se proveen datos al azar, pudiendo ser válidos o inválidos, dependiendo de su inteligencia. Para poder trasladarse por toda la aplicación, ejecutan posibles acciones que un usuario puede hacer en un dispositivo, como

ser *click*, *long-click*, *press home*, *press back*, *reset*, completar un input, etc. La manera de lograr encontrar un permiso que haya declarado y usado la aplicación bajo test, depende nuevamente de la inteligencia e ingeniería que utilice dicha herramienta de testing automático. Suponiendo que la aplicación peticione acceder periódicamente a la locación específica del dispositivo, tuvo que declarar de antemano en el manifest, el grupo o permisos individuales (grupo LOCATION, permisos individuales ACCESS_FINE_LOCATION o ACCESS_COARSE_LOCATION) para utilizar la clase android.location.LocationManager que provee el servicio de locación mediante su API. Para poder recolectar en ejecución, los llamados a estos métodos, utilizamos una herramienta que instrumentar a la aplicación, la cual sirve para poder agregar callbacks en cada método que nosotros deseemos.

1.2. Preguntas

A lo largo de esta tesis compararemos la efectividad y comportamiento de 3 herramientas de testing automático (Sapienz, Monkey y Droidmate) en materia de detección de permisos, para lograr responder las siguientes preguntas:

P1: ¿Qué herramienta logra cubrir la mayor cantidad de comportamiento de las aplicaciones?

Por comportamiento nos referimos a cantidad máxima de métodos que involucran permisos encontrados por una herramienta de testing automático sobre un conjunto de aplicaciones bajo test. Para verificar esto vamos a correr cada aplicación durante dos horas diez veces por herramienta, y vamos a verificar cuántos métodos sensibles se encuentran luego de estas dos horas, para luego interpretar qué herramienta logra obtener un mayor cubrimiento

P2: ¿Qué herramienta es más eficiente en encontrar permisos?

Si bien una herramienta puede encontrar más permisos que otra luego de dos horas, puede no hacerlo de una manera rápida. Para verificar esto, vamos a analizar los métodos encontrados por cada herramienta por minuto a lo largo de cada corrida, para quedarnos con el máximo de cada instante. De esta manera, podremos ver qué herramienta localiza métodos de manera más rápida y eficientemente.

P3: ¿Es el estado del arte apropiado para encontrar permisos en las aplicaciones?

Para responder esta pregunta propondremos un análisis basado en los permisos declarados en el Manifest de una aplicación contra los permisos encontrados de la misma cuando estuvo bajo test. Para esto necesitaremos extraer la lista de permisos declarados por cada aplicacion (apk). Luego basándose en las diferentes corridas de la sección de experimentación, obtendremos una lista de todos los distintos permisos que fueron encontrados por las herramientas de testing que comparamos (Monkey, Droidmate y Sapienz)

para cada apk. Con estas dos listas calcularemos el porcentaje de permisos encontrados y así responder si el estado del arte es apropiado para encontrar permisos en las aplicaciones.

P4: Hay permisos que son más sensibles en relación a otros. ¿Que herramienta es mejor para encontrarlos?

Los permisos de Android puede variar según su nivel de protección, siendo algunos más sensibles que otros, estos niveles son: *Normal*, *Danger*, *Signature*, *SignatureOrSystem*. En este punto estudiaremos qué herramienta logró obtener más cantidad de llamados a métodos involucrados a permisos con nivel de protección *Danger*, los más sensibles.

2. BACKGROUND

Para entender un poco el trasfondo de esta tesis, primero presentaremos una breve explicación del paper Mining Sandboxes [1], el cual nos sirvió como puntapié inicial para la comparación de las herramientas de testing automáticas en el mundo de detección de permisos y seguridad. Podríamos decir que esta tesis está basada en Mining Sandboxes, o mejor aún, que utiliza fundamentos y herramientas de la misma.

El paper basa su investigación en lograr que la incompletitud del análisis dinámico se vuelva en una garantía, mediante un sandbox que impone que todo lo que aún no se ha visto no ocurrirá en el futuro. Pero preguntas que no responde Mining Sandboxes es ¿Que herramienta de testing automático es la mejor para detectar permisos?

Decir cual es la mejor herramienta en este escenario sería un trabajo arduo ya que son muchas, y no es la idea. Nuestra tesis sólo comparara 3 de estas herramientas, 2 de las cuales son herramientas que fueron creadas con el propósito de encontrar bugs y la otra, Droidmate, que fue creada con el propósito de introducir la noción de combinar generación de test, testing dinámico y sandboxing en dicho paper.

2.1. Paper Mining Sandboxes - Boxmate

Este paper incorpora la idea de Boxmate, el cual tiene la finalidad de depositar a una aplicación Android en un sandbox restringiéndole el acceso al uso de recursos y servicios sensibles.

Boxmate es una técnica que tiene como primer fase extraer el comportamiento del software por medio de testing automático, obteniendo así el conjunto de llamadas a APIs y recursos accedidos durante el test. Este conjunto, luego, será utilizado en la segunda fase para limitar a la aplicación, solo permitiéndole usar las APIs que fueron encontrados en la primera fase.

En esta segunda fase se utiliza un sandbox el cual bloquea el acceso a los recursos que no fueron accedidos durante el testing, permitiendo proteger al usuario/sistema de cambios de comportamientos como ser software malicioso latente, infecciones, actualizaciones maliciosas, etc.

Como se ve en la Figura 1, la idea fundamental de Boxmate es proteger al sistema de programas maliciosos.

Este proceso automático permite al usuario como a comerciantes correr Boxmate requiriéndoles menos de una hora para obtener un sandbox de una app de Android.

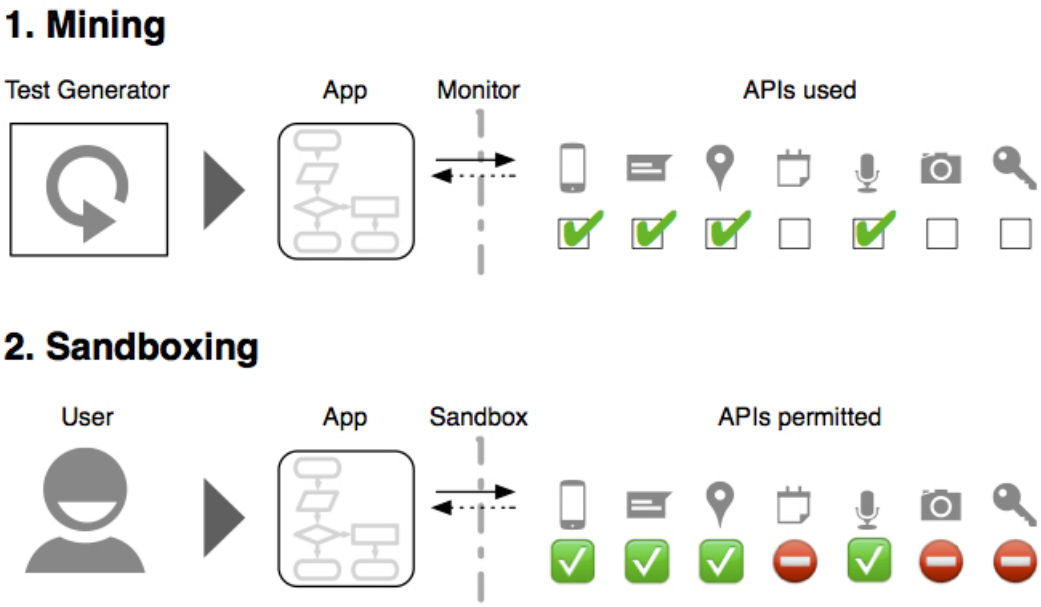

Fig. 2.1: Mining: Primera fase - Sandboxing: Segunda fase

Estructura de Boxmate

Para poder obtener una lista de las posibles APIs utilizadas por la aplicación bajo test, Boxmate se vale de la herramienta de testing automático Droidmate (la cual se verá más en detalle en la sección Droidmate) para generar tantas ejecuciones como sean necesarias y explorar el mayor comportamiento posible.

Todo este comportamiento encontrado durante la fase de Mining se considera el comportamiento normal de la aplicación. Cualquier uso de una API que no se haya descubierto durante esta fase se considera peligroso. Durante la etapa de Sandboxing, esto es, durante la ejecución día a día de la aplicación, cualquier comportamiento peligroso es bloqueado y es exhibido al usuario para que explícitamente indique si quiere ejecutarlo.

Este sistema se llama Boxify [2], que a grandes rasgos sirve para la virtualización de aplicaciones en Android, aprovechando la seguridad proporcionada por procesos aislados para encapsular de forma segura aplicaciones no confiables en un entorno de ejecución completamente sin privilegios dentro del contexto de una configuración regular de una aplicación Android. Boxify es desplegable como cualquier aplicación sin la necesidad de altos privilegios.

Siendo que Boxify instala la aplicación en un entorno aislado y encapsulado, la aplicación maliciosa puede correr tranquilamente en un entorno seguro, en consecuencia no hay necesidad de modificar el código de la aplicación bajo test.

De este componente, Boxmate saca provecho también para monitorear las APIs en forma más fina, utilizando el proceso broker de Boxify. Las operaciones sensibles de E/S se transmiten a través de este proceso privilegiado y separado, donde se lleva a cabo

la supervisión y la aplicación de políticas de seguridad.

Recapitulando, Boxmate trabaja en dos modos. Durante el minado de permisos, registra y distingue todas las llamadas sensibles a las APIs, estos registros incluyen el thread ID como también valores de recursos de seguridad relevantes. Durante la ejecución normal de la aplicación, Boxmate chequea si la llamada a la API está permitida, si no fue previamente vista en la fase de minado, este puede retornar un objeto mock, (simulando la ausencia de contactos, llamadas, etc) o notificar al usuario que la aplicación desea un permiso particular (Figura 2). Si el usuario rechaza el permiso, la llamada es denegada.

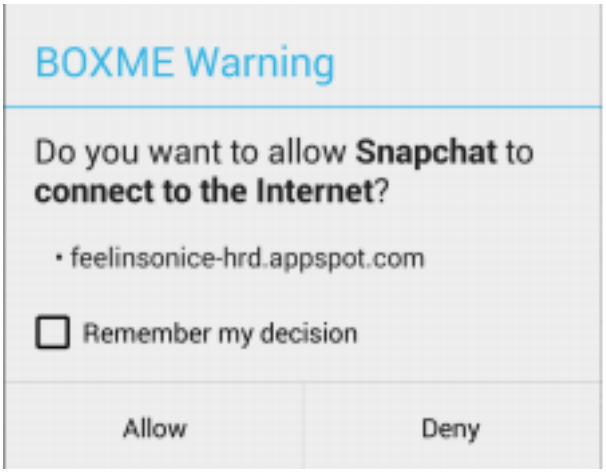

Fig. 2.2: Aquí se puede ver una notificación al usuario, donde la aplicación Snapchat solicita permiso para conectarse al sitio *feelinsonice-hrd.appspot.com* [1]

2.2. Análisis de Programas

Es el proceso de analizar automáticamente el comportamiento del software, respecto a una propiedad, como ser correctitud, optimización, seguridad, etc. Analizar un programa se focaliza principalmente en dos grandes áreas: optimización y correctitud.

El primero se centra en mejorar el rendimiento del programa mientras reduce el uso de recursos. El segundo se basa en asegurar que el programa hace lo que se supone que debe hacer.

La formas de analizar un programa pueden ser: sin ejecutarlo (análisis estático), en tiempo de ejecución (análisis dinámico) o en una combinación de ambos.

Análisis Estático

Esta categoría se basa en la inspección y análisis del código. El mismo establece una aproximación de lo que un programa puede hacer. En la mayoría de los casos, el análisis se realiza en alguna versión del código fuente y en otros casos se realiza en el código objeto. El término se aplica generalmente a los análisis realizados por una herramienta automática, el análisis realizado por un humano es llamado comprensión de programas.

Si el análisis estático pudiera establecer qué hace exactamente un programa, sería posible poder excluir esa parte del programa en los casos donde haya un uso maligno, pero esto no es posible, ya que es un problema indecidible.

Más aún el análisis estático se ve desafiado por código que está ofuscado, interpretado o es descargado en tiempo de ejecución. Programas con intenciones maliciosas cuentan con estas características, por esa razón el análisis estático no es suficiente y da lugar a la necesidad de análisis dinámicos.

Existen herramientas en Android que utilizan análisis estático para detectar vulnerabilidades en las aplicaciones, tales como CHEX [7] y FlowDroid [8].

Análisis Dinámico

El Análisis dinámico funciona en base a la ejecución del programa y observar su comportamiento (a diferencia de las técnicas estáticas de análisis que no ejecutan el software). Hay varias estrategias que ayudan al análisis del comportamiento del programa, como ser monitorear las llamadas al sistema, cobertura de código que se ejecutó o técnicas de testing automático, etc.

Pero aun así puede llegar a ser insuficiente este análisis, con lo cual para que resulte efectivo el programa a ser analizado se debe ejecutar con los suficientes casos de prueba como para producir un comportamiento interesante y a la vez tratando de abarcar un mayor cubrimiento. Las posibilidades de ejecuciones pueden ser mediante tests escritos por desarrolladores o por testers automáticos, este último es el que enfocaremos en nuestra tesis, puesto que las herramientas que compararemos son todas automáticas.

Testing automático

Dentro del espectro del análisis dinámico, el testing automático es utilizado hoy en día para poder facilitar el arduo trabajo de testear el funcionamiento de las aplicaciones, con fines de optimizar su calidad, y acelerar el proceso de puesta en el mercado. Si bien no reemplazan las tareas de testing manual, agiliza los tiempos de testeo y permite la reutilización de los mismos, haciendo el proceso más económico.

El problema de este enfoque es la incompletitud: si no se ha observado hasta el momento algún comportamiento, no hay garantía de que no pueda ocurrir en el futuro. Dado el alto costo de las falsos positivos, esto implica que se necesita un conjunto suficientemente grande de ejecuciones para cubrir todo el dominio de la aplicación.

En Android, debido al gran número de aplicaciones que se crean día a día, los distintos tipos de dispositivos a tener en cuenta a la hora de testear y la necesidad de detectar si una app está requiriendo más permisos de los necesarios, empresas dedicadas a la seguridad ven la necesidad de encontrar herramientas de testing automático que puedan ayudar con el labor de la detección de permisos que podría estar requiriendo una aplicación.

Dado que los programas vienen en código de byte interpretable, la plataforma ofrece la oportunidad de que se pueda supervisar el comportamiento dinámico, incluyen-

do llamadas al sistema (AASandbox [24]), flujo de datos (TAINTDROID [25]), CPU y actividad de red (ANDROMALY [26]); Todas estas plataformas pueden utilizarse tanto para supervisar el comportamiento de la aplicación (como para informar los resultados al usuario), así como para detectar comportamientos maliciosos.

2.3. Android

Android es un sistema operativo basado en el kernel de Linux. Fue diseñado principalmente para dispositivos móviles con pantalla táctil, como teléfonos inteligentes, tablets y también para relojes inteligentes, televisores, etc. Hasta la versión 4.4.4 Android utiliza Dalvik como máquina virtual con la compilación just-in-time (JIT) para ejecutar Dalvik "dex-code"(Dalvik ejecutable), que es una traducción de Java bytecode.

Android 4.4 introdujo el ART (Android Runtime) como un nuevo entorno de ejecución, que compila el Java bytecode durante la instalación de una aplicación. Se convirtió en la única opción en tiempo de ejecución en la versión 5.0

Desde sus comienzos la administración de permisos mutó mucho con el pasar de los dispositivos cambiando sistemas más falibles por otros más robustos para la protección del equipo e información del usuario sensible.

Permisos del sistema Android

Android es un sistema operativo con privilegios independientes, en el que cada aplicación se ejecuta con una identidad de sistema distinta (ID de usuario de Linux y ID de grupo). También se separan partes del sistema en identidades distintas. Así, Linux aísla las aplicaciones entre sí y del sistema operativo.

Se ofrecen funciones de seguridad adicionales más precisas mediante un mecanismo de "permisos" que aplica restricciones en las operaciones específicas para que un proceso en particular puede realizar y permisos por URI para la dar acceso ad-hoc a elementos específicos de datos.

Arquitectura de Seguridad

Un punto central del diseño de la arquitectura de seguridad de Android consiste en que ninguna aplicación, de manera predeterminada, tiene permiso para realizar operaciones que pudieran tener consecuencias negativas para otras aplicaciones, el sistema operativo o el usuario. Esto incluye leer datos privados de los usuarios o escribir en ellos (por ejemplo, los contactos o los mensajes de correo electrónico), leer archivos de otra aplicación o escribir en ellos, acceder a una red, mantener el dispositivo activo, etc.

Debido a que cada aplicación de Android funciona en una zona de pruebas de proceso, las aplicaciones deben compartir recursos y datos de manera explícita. Hacen esto declarando los permisos que necesitan para capacidades adicionales no previstas por la zona

de pruebas básica. Las aplicaciones declaran estáticamente los permisos que necesitan y el sistema Android solicita al usuario su consentimiento.

La zona de pruebas de aplicaciones no depende de la tecnología empleada para crear una aplicación. En especial, la VM Dalvik no supone un límite de seguridad y cualquier app puede ejecutar código nativo. Todos los tipos de aplicaciones (Java, nativas e híbridas) se colocan en zonas de pruebas de la misma manera y tienen el mismo grado de seguridad.

Firma de aplicaciones y Id de usuario

Todos los APK (archivos .apk) deben estar firmados con un certificado cuya clave privada esté en manos del desarrollador. Este certificado identifica al autor de la aplicación. No es necesario que el certificado lleve la firma de una autoridad de certificación; es perfectamente admisible, y común, que las aplicaciones de Android usen certificados autofirmados. El objetivo de los certificados de Android es distinguir a los autores de las aplicaciones. Esto permite al sistema otorgar o denegar a las aplicaciones el acceso a permisos de nivel de firma y otorgar o denegar a una aplicación una solicitud para usar la misma la identidad de Linux que otra aplicación.

En el momento de la instalación, Android asigna a cada paquete un ID de usuario de Linux distinto. La identidad se mantiene constante durante la vida útil del paquete en ese dispositivo. En un dispositivo diferente, el mismo paquete puede tener otro ID; lo importante es que mantenga ese ID dentro del mismo dispositivo.

Debido a que la seguridad de la aplicación tiene lugar en el nivel del proceso, el código de dos paquetes cualesquiera normalmente no puede ejecutarse en el mismo proceso, ya que deben funcionar como diferentes usuarios de Linux. Usando el mismo atributo *sharedUserId* en el AndroidManifest.xml de cada uno de los paquetes se puede hacer que se les asigne el mismo ID de usuario. Al hacer esto, por motivos de seguridad, los dos paquetes se tratan como si fueran una misma aplicación, con el mismo ID de usuario y permisos de archivo. Pero obviamente, a fin de mantener la seguridad, sólo se asignará el mismo ID de usuario a dos aplicaciones que tengan la misma firma.

Uso de permisos

Una aplicación básica de Android no tiene permisos asociados de manera predeterminada. Esto significa que no puede hacer nada que afecte negativamente la experiencia del usuario o los datos en el dispositivo. Para usar funciones protegidas del dispositivo se debe incluir u.na o más etiquetas *<uses-permission>* en el manifiesto de la app app (*AndroidManifest.xml*) Por ejemplo, una aplicación que tiene que controlar los mensajes SMS entrantes especificará lo siguiente:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.app.miAplicacion" >
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
```

```
</manifest>
```

Si se incluyen permisos normales (es decir, los que no presentan un gran riesgo para la privacidad del usuario o el funcionamiento del dispositivo), el sistema otorga automáticamente esos permisos. Si en el manifiesto de la aplicación se incluyen permisos riesgosos (es decir, los que podrían afectar la privacidad del usuario o el funcionamiento normal del dispositivo), el sistema solicita al usuario que otorgue explícitamente esos permisos. La manera en que Android realiza la solicitud depende de la versión del sistema y la versión del sistema objetivo de la app:

- Si el dispositivo tiene instalado Android 6.0 (nivel de API 23) o versiones posteriores y el atributo *targetSdkVersion* de la app es 23 o superior, la app solicita los permisos al usuario en el tiempo de ejecución. El usuario puede revocar los permisos en cualquier momento. Por ello, la app debe controlar si tiene los permisos cada vez que se ejecuta.
- Si el dispositivo tiene instalado Android 5.1 (nivel de API 22) o versiones anteriores, o el atributo *targetSdkVersion* de la app es 22 o un valor inferior, el sistema solicita al usuario que otorgue los permisos al instalar la app. Una vez que el usuario instale la app, la única manera que tiene de revocar el permiso es desinstalar la app.

Los permisos que proporciona el sistema Android se pueden encontrar en Manifest.permission. Cualquier aplicación también puede definir y aplicar sus propios permisos, por lo cual esta no es una lista completa de todos los permisos posibles.

Permisos normales y riesgosos

Los permisos del sistema se dividen en varios niveles de protección. Los dos niveles más importantes son el normal y el riesgoso. Los permisos normales abarcan áreas en las cuales la app tiene que acceder a datos o recursos fuera de su zona de pruebas, pero donde existe un riesgo mínimo para la privacidad del usuario o el funcionamiento de otras apps. Por ejemplo, el permiso para establecer el huso horario es un permiso normal. Si una app declara que necesita un permiso normal, el sistema le otorga automáticamente el permiso en el momento de la instalación. El sistema no pide al usuario conceder permisos normales como tampoco el usuario puede revocar estos permisos. Algunos de los permisos normales (clasificados como PROTECTION_NORMAL) son:

- ACCESS_LOCATION_EXTRA_COMMANDS
- ACCESS_NETWORK_STATE
- ACCESS_NOTIFICATION_POLICY
- ACCESS_WIFI_STATE
- BLUETOOTH, BLUETOOTH_ADMIN

- BROADCAST_STICKY
- KILL_BACKGROUND_PROCESSES
- MODIFY_AUDIO_SETTINGS
- NFC
- READ_SYNC_SETTINGS
- RECEIVE_BOOT_COMPLETED
- REORDER_TASKS
- SET_ALARM
- SET_TIME_ZONE
- SET_WALLPAPER
- UNINSTALL_SHORTCUT
- USE_FINGERPRINT
- VIBRATE
- WAKE_LOCK
- WRITE_SYNC_SETTINGS

Los permisos riesgosos abarcan áreas en las cuales la app requiere datos o recursos que incluyen información privada del usuario, o bien que podrían afectar los datos almacenados del usuario o el funcionamiento de otras apps. Por ejemplo, la capacidad de leer los contactos del usuario es un permiso riesgoso. Si una app declara que necesita un permiso riesgoso, el usuario tiene que otorgarle explícitamente el permiso.

Grupos de permisos

Todos los permisos riesgosos del sistema Android pertenecen a grupos de permisos. Si el dispositivo tiene Android 6.0 (nivel de API 23) instalado y el atributo targetSdkVersion de la app es 23 o un valor superior, el siguiente comportamiento del sistema tiene lugar cuando la app solicita un permiso riesgoso:

- Si una app solicita un permiso riesgoso incluido en su manifiesto y no tiene permisos actualmente en el grupo de permisos, el sistema muestra un cuadro de diálogo al usuario en el que se describe el grupo de permisos al cual la app desea acceder. En el cuadro de diálogo no se describe el permiso específico dentro de ese grupo. Por ejemplo, si una app solicita el permiso READ_CONTACTS, en el cuadro de diálogo del sistema se indica únicamente que la app necesita acceso a los contactos del dispositivo. Si el usuario brinda la aprobación, el sistema otorga a la app solamente el permiso que solicitó.

- Si una app solicita un permiso riesgoso incluido en su manifiesto y ya tiene otro permiso riesgoso en el mismo grupo de permisos, el sistema lo otorga de inmediato sin interacción con el usuario. Por ejemplo, si a una app ya se le otorgó el permiso READ_CONTACTS y luego esta solicita el permiso WRITE_CONTACTS, el sistema lo otorga de inmediato. Cualquier permiso puede pertenecer a un grupo de permisos; entre ellos, los normales y los que define una app. Sin embargo, el grupo de un permiso solo afecta la experiencia del usuario si es riesgoso. Si el dispositivo tiene instalado Android 5.1 (nivel de API 22) o versiones anteriores, o si el atributo *targetSdkVersion* de la app es 22 o inferior, el sistema solicita al usuario que otorgue el permiso en el momento de la instalación. Nuevamente, el sistema solo dice al usuario qué grupos de permisos necesita la app y no los permisos individuales.

Algunos de los permisos riesgosos son :

Grupo de Permisos	Permisos
CALENDAR	READ_CALENDAR WRITE_CALENDAR
CAMERA	CAMERA
CONTACTS	READ_CONTACTS WRITE_CONTACTS GET_ACCOUNTS
LOCATION	ACCESS_FINE_LOCATION ACCESS_COARSE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	READ_PHONE_STATE CALL_PHONE READ_CALL_LOG WRITE_CALL_LOG ADD_VOICEMAIL USE_SIP PROCESS_OUTGOING_CALLS
SENSORS	BODY_SENSORS
SMS	SEND_SMS RECEIVE_SMS READ_SMS RECEIVE_WAP_PUSH RECEIVE_MMS
STORAGE	READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE

Para aplicar los propios permisos, primero se deben declararlos en AndroidManifest.xml con uno o más elementos <permission>.

Muchas aplicaciones especifican más permisos de los que utilizan pero esto no significa, no obstante, que todas las aplicaciones que solicitan diversos permisos sean maliciosas por naturaleza. No existe una única lista de permisos para los dispositivos basados en Android. Por otro lado, algunas aplicaciones de carácter maliciosa solicitan permisos

para manipularlos y llevar a cabo acciones no esperadas, por ejemplo, grabar las conversaciones, enviar información sensible del dispositivo a un servidor externo, etc.

Diferencias entre API19 y API23

I) En Octubre del 2013 fue lanzada la versión Android 4.4 (API 19) llamada KitKat.

Esta versión trae consigo muchas novedades, tanto en diseño, como en funcionalidad y rendimiento. Mas allá de estas mejoras, en esta versión, cuando deseamos instalar una aplicación desde Google Play, lo primero que se nos informa previo a la instalación, son los permisos que dicha aplicación requiere. Esta versión no introdujo la administración puntual de los permisos, con lo cual el usuario solo podía aceptar o rechazar la instalación.

II) En Agosto del 2015 fue lanzada la versión Android 6.0 (API 23) llamada Marshmallow.

Una de las grandes novedades de Android 6.0 fue el administrador de permisos. En esta versión podemos conceder permisos a las aplicaciones para que accedan a la información y a las funciones específicas de nuestro dispositivo que nosotros queremos. Fue una medida de seguridad que dio más control y privacidad a los usuarios. Una vez instalada una aplicación Android nos irá pidiendo autorización por cada uno de los permisos que va a necesitar para funcionar correctamente. Nos pueden pedir permiso uno tras otro, o bien, a medida que los vayamos necesitando. También está la posibilidad de rechazar un permiso por un determinado momento o bloquear ese permiso permanentemente. Si lo rechazamos para siempre, cuando la aplicación lo requiera ya nos informará que tal permiso está bloqueado, mostrando un acceso a los ajustes por si queremos activarlo manualmente. También permite administrar los permisos de una aplicación más tarde pudiendo acceder a sus administradores de permisos. Tenemos dos formas de gestionar los permisos en Android: por aplicaciones o por permisos.

Esta tesis basa sus experimentaciones en Android 4.4 API 19, esto es debido a la compatibilidad que tienen las 3 herramientas de testing automático (Droidmate, Monkey y Sapienz) para poder hacer una comparación en las mismas condiciones. Como se comentó antes, nuestra tesis se basa en el artículo Mining Sandboxes [1], el cual utiliza Droidmate para hacer Testing Automático y extraer los permisos encontrados. La compatibilidad de Droidmate varía, funciona con limitaciones con emuladores API 23, puesto que el inlined de apks no es compatible con emuladores rápidos x86, pero si con ARM architecture; por esta razón no se pueden monitorear los llamados al framework para detectar permisos. Por consiguiente y sumado a que Sapienz no funciona en API 23, la tesis utilizó siempre API 19.

2.4. Herramientas de testeo

2.4.1. Droidmate

Conceptualmente Droidmate genera tests mediante la exploración de la Aplicación bajo Test (AuT) , es decir, mediante la interacción en tiempo de ejecución con los elementos de la interfaz de usuario (GUI), llamado en Android Views (o Vistas). Droidmate toma como input un directorio el cual contiene un conjunto de archivos .apk (Aplicación empaquetada de Android) para la exploración.

Previamente a cada una de estas aplicaciones se le corre un proceso de *inlining* para luego ser testeada. El *inlining* se realiza mediante la herramienta Appguard [4] que en grandes rasgos genera una ligera modificación de bytecode dalvik en la aplicación permitiendo el monitoreo de los métodos de las API de SDK de Android sensibles a permisos, estos métodos están especificados en una archivo de configuración. Luego Droidmate instala en el Dispositivo Android cada una de estas .apk que contiene la aplicación que se desea testear y luego lanza el Activity principal de la misma.

Durante el comienzo y luego de cada interacción Droidmate monitorea qué recursos y APIs sensibles fueron accedidos. Mientras la exploración continua, todos los comportamientos monitoreados de la aplicaciones son usados para decidir cuál es el siguiente elemento GUI con el que se interactuara o si la exploración debe terminar. Los datos de salida de cada exploración de una corrida son guardados como objetos serializados de Java. Estos datos son suficientes para replicar un test, tanto manual como automático.

La estrategia de exploración opera en un alto nivel de abstracción tomando como input un estado de GUI y retornando una acción de exploración. Las acciones de exploración pueden ser cualquiera de las siguientes operaciones sobre el dispositivo de Android: *click*, *long-click*, *press home*, *press back*, *reset*, *terminate*. Estas acciones luego son ejecutadas en el dispositivo. Luego se le da lugar al driver de exploración quién es el encargado de traducir una acción en una operación concreta y real sobre el dispositivo, ejecutandolas, leyendo el estado del resultado GUI y las APIs llamadas, retornando así el control a la estrategia de exploración.La estrategia de exploración utilizada por Droidmate está inspirada en DYNODROID [3].

La idea principal es interactuar con las vistas (views) de Android aleatoriamente, pero priorizando las que menos fueron visitadas. Si hay vistas con una misma cantidad de interacciones se elige una en forma aleatoria. Las interacciones sólo se pueden dar con vistas que puedan ejecutar una acción como ser clickables, checkables, long-clickable, estar habilitados y visibles.

Una estrategia mantiene una lista de contextos. Cada contexto esencialmente es una lista de widgets. Cada vez que se ejecuta una acción, se obtiene un snapshot de la pantalla, el cual es un estructura XML del layout actual. Esto lo provee el uiautomator, el cual es el framework oficial GUI de Google. De este XML se obtienen los widgets que son clickeables/long clickeables y están visibles, y con esos se conforma un contexto.

Una vez obtenido el contexto, se compara si es igual a otro previo, dos contextos son iguales si muestran la misma actividad y tienen los mismos widgets en el mismo orden;

si es igual, se usa ese contexto, sino se agrega el contexto actual a la lista. Esto se hace para poder mantener información entre acción y acción, si una acción no produce ningún cambio en la pantalla, el contexto resultante va a ser el mismo y en la siguiente acción probablemente droidmate elija otro widget.

Cada 30 interacciones Droidmate reinicia la aplicacion que se esta testeando. Para evitar bloquearse en una situación anormal como ser que no haya una vista disponible para la interacción, o que la aplicación se bloquee etc. La exploración termina cuando el tiempo límite de ejecución es alcanzado o cuando no hay vistas con las que se pueda interactuar.

AppGuard

AppGuard [4] es un framework de seguridad para Android basado en políticas, que permite tomar una aplicación no confiable y una lista de políticas de seguridad definidas por el usuario como entrada, incorporando un monitoreo de seguridad en dicha aplicación. Las políticas de seguridad pueden especificar restricciones en invocaciones de métodos así como ciertos requisitos.

El enfoque utilizado por AppGuard en primer lugar es, dado los binarios de la aplicación instrumentar las referencias a monitorear (IRM), lo cual implica interceptar las llamadas sensibles para invocar en tiempo de ejecución un monitor de seguridad. Técnicamente, esto se logra mediante la alteración de las referencias a métodos en la VM de Dalvik. Luego en segundo lugar, el monitor de seguridad comprueba dinámicamente si alguna de las políticas de seguridad actualmente aplicadas permite la operación y, a continuación, concede la ejecución o ejecuta un código alternativo (por ejemplo, devolver un valor simulado para evitar la terminación de la aplicación debido a una excepción). Este enfoque no requiere acceso a root o cambios en la arquitectura de Android, de esta manera la seguridad se integra en la aplicación, por otro lado este proceso agrega muy poco sobrecarga en términos de espacio y tiempo de ejecución.

Es importante destacar que AppGuard monitorea tanto llamadas directas a Java como tambien llamadas a código nativo Java. Sin embargo, una limitación, es que no monitorea llamadas a funciones dentro de librerías nativas.

2.4.2. Sapienz

Enfoque

Sapienz [6] comienza por instrumentar la aplicación bajo test. Esta instrumentación puede lograrse en una caja blanca, caja gris o caja negra. Es decir, cuando el código fuente de la aplicación está disponible, Sapienz utiliza instrumentación de grano fino, es decir a nivel de sentencia (caja blanca). Por el contrario, si sólo el archivo binario APK está disponible (como suele ser en el caso de escenarios de pruebas de Android en el mundo real), Sapienz utiliza descompilación (undexing) y empaquetado para instrumentar la aplicación a nivel de método (caja gris). Sin embargo, donde los desarrolladores prohíben el re-empaquetado (como es común en aplicaciones comerciales), este utiliza una cobertura

de piel no invasiva (llamada así porque sólo interactúa con la UI y las acciones del sistema) de nivel de actividad que siempre se puede medir (caja negra).

Sapienz extrae constantes de cadena estáticamente definidas por ingeniería inversa de la APK. Estas cadenas se utilizan como inputs de prueba, para cadenas reales en la aplicación, con esto se logra mejorar el realismo de los inputs que se prueban. Las secuencias de test son generadas y ejecutadas por el componente *MotifCore*, que combina testing automático y exploración sistemática. Los cuales corresponden a dos genes: el de más bajo nivel, atómico y el de más alto nivel, Motif.

El algoritmo de búsqueda de Sapienz inicializa la población a través del generador de pruebas de *MotifCore*. Durante el proceso de evolución genética, los individuos genéticos son asignados al *Test Replayer*, al evaluar las aptitudes individuales. Los scripts de prueba son luego de-codificados en eventos ejecutables en Android por el *Gene Interpreter*, que se comunica con el dispositivo Android a través de *Android Debugging Bridge* (ADB). El State Logger supervisa los estados de ejecución (por ejemplo, actividades que fueron cubiertas, crashes) de la aplicación bajo prueba (AuT) y produce datos de medición para el *Fitness Extractor* para calcular las aptitudes. Luego de todo este proceso, un conjunto de soluciones Pareto óptimas e informes de prueba son generados al final de la búsqueda.

Búsqueda multiobjetivo

Sapienz trata de optimizar tres objetivos simultáneamente:

- maximizar la cobertura del código
- minimizar la longitud de la secuencia
- maximizar el número de fallos encontrados

Todo esto utilizando un enfoque de Pareto-óptimo basado en la ingeniería del software basada en búsqueda (SBSE), esto es, técnicas metaheurísticas que buscan una solución lo suficientemente buena de acuerdo a criterios [15] [16].

Cada conjunto de test ejecutable $\vec{x}$ para la aplicación bajo test se denomina como solución y una solución $\vec{x_a}$ está dominada por la solución $\vec{x_b}$ ($\vec{x_a} < \vec{x_b}$) según una función de aptitud f si y sólo si:

$$\forall i = 1, 2, ..., n, f_i(\vec{x_a}) \leq f_j(\vec{x_b}) \wedge$$
$$\exists j = 1, 2, ..., n, f_j(\vec{x_a}) < f_j(\vec{x_b}) \quad (1)$$

Un conjunto Pareto-óptimo consiste en todas las soluciones Pareto-óptimas (pertenecientes a todas las soluciones X_t), que se definen como:

$$P^* \triangleq \{\vec{x^*} \mid \nexists \vec{x} \in X_t, \vec{x} \prec \vec{x^*}\} \quad (2)$$

Algorithm 1: Overall algorithm of SAPIENZ.

Input: AUT A, crossover probability p, mutation probability q, max generation g_{max}, execution time t
Output: UI model M, Pareto front PF, test reports C

$M \leftarrow K_0$; $PF \leftarrow \emptyset$; $C \leftarrow \emptyset$; ▷ initialisation
generation $g \leftarrow 0$;
boot up devices D; ▷ prepare app exerciser
inject MOTIFCORE into D; ▷ for hybrid exploration (see I)
static analysis on A; ▷ for string seeding (see II)
instrument and install A;
initialise population P; ▷ hybrid of random and motif genes
evaluate P with MOTIFCORE and update (M,PF,C);
while $g < g_{max}$ and $\neg$ *timeout(t)* **do**
  $g \leftarrow$ g+1;
  $Q \leftarrow wholeTestSuiteVariation(P, p, q)$; ▷ see Algorithm 2
  evaluate Q with MOTIFCORE and update (M,PF,C);
  $\mathcal{F} \leftarrow \emptyset$; ▷ non-dominated fronts
  $\mathcal{F} \leftarrow sortNonDominated(P \cup Q, |P|)$;
  $P' \leftarrow \emptyset$; ▷ non-dominated individuals
  for each front F in $\mathcal{F}$ **do**
    if $|P'| \geq |P|$ **then** break;
    calculate crowding distance for F;
    for each individual f in F **do**
      $P' \leftarrow P' \cup f$;
  $P' \leftarrow sorted(P', \prec_c)$; ▷ see equation 3 for operator $\prec_c$
  $P \leftarrow P'[0 : |P|]$; ▷ new population
return (M,PF,C);

Fig. 2.3: Algoritmo 1 - Ejecución de Sapienz [6]

La búsqueda que hace Sapienz se basa en el algoritmo *NSGA-II* para construir conjuntos Pareto-óptimos perfeccionados sucesivamente, buscando nuevos vectores dominantes de prueba. NSGA-II es un algoritmo genético multiobjetivo de búsqueda evolutiva muy usado en SBSE [16], para más detalles, ver [13].

Al final de la búsqueda, los testers pueden elegir el conjunto Pareto-óptimo generado por Sapienz.

Representación SBSE

Sapienz, como dijimos, se basa un algoritmo genético, que realiza una cadena evolutiva entre sucesivas generaciones de individuos, y en donde cada individuo corres-

ponde a un test suite [21] [22]. La representación de un test suite individual generado por Sapienz se ilustra en la figura 2.4.

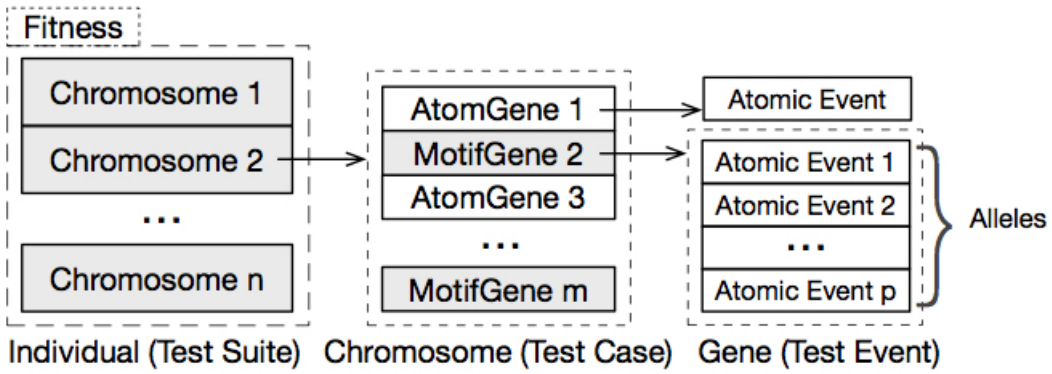

Fig. 2.4: Representación individual genética [6]

Sapienz genera un conjunto de estos test suites, que corresponde a una población de individuos en el algoritmo evolutivo. Cada individuo consta de varios cromosomas (secuencias de test T_1, T_2, ... T_n) y cada cromosoma contiene múltiples genes (eventos de test E_1, E_2 .... E_n), que consisten en una combinación aleatoria de genes atómicos y genes motif. Un gen atómico desencadena un evento atómico e que no puede ser descompuesto, por ejemplo, presionar hacia abajo una tecla, clickear, etc; mientras que un gen motif se interpreta como una serie de eventos atómicos (e_1, e_2, ... e_n).

Operador de variación de SBSE

Definimos un operador de variación de un test suite para manipular a los individuos. El operador se representa en el algoritmo 2:

Este aplica un conjunto de operaciones de grado fino: *crossover*, *mutation* y *reproduction* a cada individuo (a nivel de test suite). La variación entre los individuos de Sapienz se logra mediante el uso de un *crossover* de conjunto de elementos uniforme entre los mimos (los test suites). La variación de cada individuo es manipulada por una operación de *mutation* más compleja. Puesto que cada individuo es un test suite que contiene varios casos de prueba, primero la operación ordena al azar los casos de prueba y luego realiza un cruce de un solo punto en dos casos de test vecinos con probabilidad q, donde la operación aleatoria anterior tiene como objetivo mejorar la diversidad de cruce. Posteriormente, la operación mutation de un caso de prueba de grano más fino baraja los eventos de prueba dentro de cada caso de prueba con probabilidad q, cambiando aleatoriamente las posiciones de los eventos.

La operación *reproduction* deja simplemente un individuo elegido aleatoriamente sin cambios.

Algorithm 2: The whole test suite variation operator.

Input: Population P, crossover probability p, mutation probability q
Output: Offspring Q

$Q \leftarrow \emptyset$;
for i in $range(0, |P|)$ **do**
  generate $r \sim U(0,1)$;
  if $r < p$ **then** ▷ apply crossover
    randomly select parent individuals x_1, x_2;
    $x'_1, x'_2 \leftarrow uniformCrossover(x_1, x_2)$;
    $Q \leftarrow Q \cup x'_1$
  else if $r < p + q$ **then** ▷ apply mutation
    randomly select individual x_1;
    ▷ vary test cases within the test suite x_1
    $x \leftarrow shuffleIndexes(x_1)$;
    for i in $range(1, |x|, step\ 2)$ **do**
      generate $r \sim U(0,1)$;
      if $r < q$ **then**
        $x[i-1], x[i] \leftarrow onePointCrossover(x[i-1], x[i])$;
    ▷ vary test events within the test case $x[i]$
    for i in $range(0, |x|)$ **do**
      generate $r \sim U(0,1)$;
      if $r < q$ **then**
        $x[i] \leftarrow shuffleIndexes(x[i])$;
    $Q \leftarrow Q \cup x$
  else $Q \leftarrow Q \cup$ (randomly selected x_1); ▷ apply reproduction
return Q;

Fig. 2.5: Algoritmo 2 - Obtención de una nueva generación de test suites [6]

Selección de SBSE

Utiliza el operador selecto de NSGA-II [13], que define un operador de comparación de crowding-distance-based $\prec_c$. Para dos secuencias de prueba $\vec{a}$, y $\vec{b}$. Decimos $\vec{a} \prec_c \vec{b}$ si y sólo si

(Ecuación 3, referenciada en Algoritmo 1)

$$\vec{a}_{rank} < \vec{b}_{rank} \vee (\vec{a}_{rank} = \vec{b}_{rank} \wedge \vec{a}_{dist} > \vec{b}_{dist})$$

Esta selección favorece secuencias de test con menor rango de no dominación y, cuando el rango es igual, favorece a la que tiene mayor distancia de aglomeración (región menos densa).

Evaluación de la SBSE

El valor de optimalidad es almacenada como tripla para cada uno de los objetivos: cobertura de código, longitud del test y número de crashes encontrados. La evaluación de optimalidad de SBSE puede consumir mucho tiempo, pero afortunadamente este proceso se puede hacer en paralelo. Por lo tanto, con el fin de lograr una mejora de tiempo en la búsqueda, Sapienz soporta la evaluación en paralelo, asignando individuos a múltiples evaluadores de optimalidad, que pueden funcionar en dispositivos distribuidos.

I. Estrategia de exploración

Las aplicaciones de Android pueden tener interacciones complejas entre los eventos que se pueden realizar desde la interfaz de usuario, los estados alcanzables y la consecuente cobertura lograda. El tester que realiza un test manual, puede explorar interacciones complejas más fácilmente. Sin embargo, para las pruebas automatizadas, es necesario encontrar alguna otra forma de manejar interacciones complejas. Sapienz destaca que los enfoques simples para las pruebas automatizadas de Android utilizan sólo eventos atómicos. Incluso con combinaciones de estos eventos, la falta de conciencia del estado y el contexto, hace que sea difícil descubrir interacciones complejas. Esta puede ser una de las razones por las que se descubrió que muchas herramientas de investigación no tuvieron éxito en comparación con Monkey en el estudio comparativo de Choudhary et al. [12]. Para abordar esta cuestión, Sapienz utiliza patrones Motif, el cual colecta patrones de eventos de bajo nivel con el propósito de obtener una mayor cobertura. Los genes Motif dependen de la información de la interfaz de usuario (UI) disponible en la vista actual, que se basa en widgets para aplicaciones de Android. Estos genes trabajan juntos para hacer patrones de uso de comportamiento en la aplicación, por ejemplo, llenar un conjunto de campos en un formulario de la vista actual y presionar enviar.

Esto se logra mediante la predicción de patrones que capturan la experiencia de los testers con respecto a interacciones complejas con la aplicación. El gen motif se inspira en cómo funciona un ADN motif: Un ADN motif es un patrón de secuencia corto que tiene una función biológica. Los morif se combinan con secuencias atómicas para que, juntos, puedan expresar la función general del ADN. En este caso, los genes motif buscan alcanzar funciones de alto nivel (mediante la definición de patrones) y trabajar junto con genes atómicos para lograr una mayor cobertura de la prueba. En Sapienz [6] se comenta que la experimentación la basaron únicamente en un único gen motif, para evitar cualquier riesgo de sesgo del experimentador. Sin embargo, en trabajos futuros, aseguran que podrían aprender motif de actividades de prueba capturadas por humanos.

Exploración híbrida: Los genes atómicos y genes motif son complementarios (ver Figura 6), por lo que Sapienz los combina para formar secuencias híbridas de eventos de prueba. La exploración aleatoria puede (aleatoriamente) cubrir los estados de la UI que no

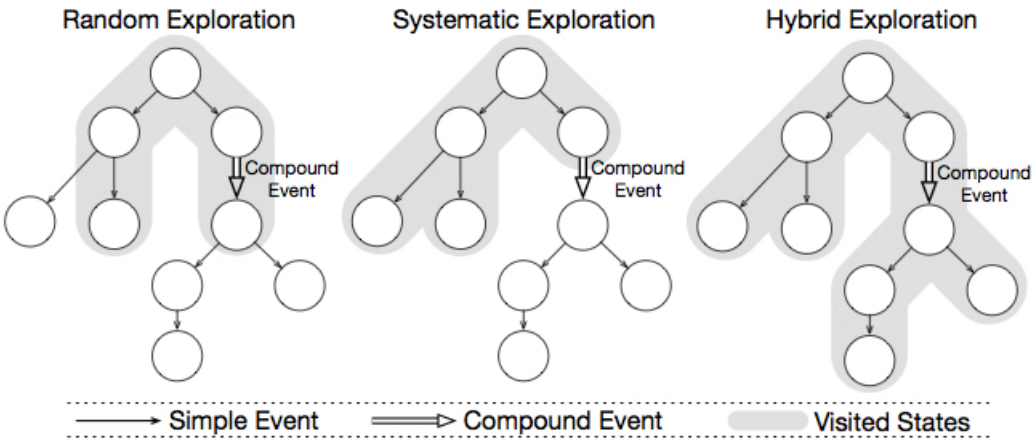

Fig. 2.6: Estrategia de exploración híbrida [6]

estuvieron planificados con eventos compuestos (los cuales consisten en una combinación aleatoria de eventos atómicos), por lo que generalmente puede lograr una cobertura global baja. La exploración sistemática puede lograr una buena cobertura dentro de regiones planificadas de la UI, pero puede ser bloqueada por patrones compuestos no planificados. La estrategia híbrida utilizada por Sapienz se muestra en el Algoritmo 3

II. Análisis Estático y Dinámico

Sapienz realiza dos tipos de análisis: análisis estático para la obtención de de cadenas y análisis dinámico para instrumentación multinivel. Estas dos características proporcionan la información necesaria para que Sapienz genere entradas de prueba realistas y guíe la búsqueda hacia conjuntos de pruebas óptimas con alto cobertura de código.

Obtención de Strings (string seeding): Con el fin de extraer las cadenas estáticas definidas en el código, Sapienz utiliza ingeniería inversa en el archivo APK. Este obtiene una lista de cadenas globales desde el archivo de recursos XML decompilado. Estas cadenas de lenguaje natural son sembradas al azar en los campos de texto por el componente MotifCore, al realizar su exploración híbrida. Este llenado de inputs es particularmente útil cuando se prueban aplicaciones que requieren una gran cantidad de contenido generado por el usuario, como por ejemplo, Facebook, ya que permite publicar y comentar de una manera aparentemente más humana. Cuando al APK no se le puede hacer ingeniería inversa con éxito, que es un caso común para las aplicaciones comerciales, se utilizan strings predefinidos.

Instrumentación de múltiples niveles: Para ser práctico y útil, una técnica de prueba automatizada de Android debe ser aplicable tanto para aplicaciones abiertas como cerradas. Para lograr esto, Sapienz utiliza una instrumentación multinivel de distinta granularidad de instrumentación. La granularidad de instrumentación más gruesa es siempre posible, y se realiza a través de interacciones entre actividades/screen para conseguir pruebas de caja negra o cobertura de piel llamada así ya que sólo interactúa con la UI y

Algorithm 3: The MotifCore exploration strategy.

Input: AUT A, test sequence $T = \langle E_1, E_2, ..., E_n \rangle$, random event list $\mathcal{R}$, motif event list $\mathcal{O}$, static strings $\mathcal{S}$ existing UI Model M and test reports C
Output: Updated (M, C)

```
for each event E in T do
    if E ∈ R then                                   ▷ handle atomic gene
        execute atomic event E and update M;
    if E ∈ O then                                   ▷ handle motif gene
        currentActivity ← extractCurrentActivity(A)
        uiElementSet ← extractUiElement(currentActivity)
        for each element w in uiElementSet do
            if w is EditText widget then
                seed string s ∈ S into w;
            else
                exercise w according to motif patterns in E;
            update M;
    (a, m, s) ← get covered activities, methods, statements;
    C ← C ∪ (a, m, s);                              ▷ update coverage reports
if captured crash c then
    C ← C ∪ c;                                      ▷ update crash reports
return (M, C);
```

Fig. 2.7: Algoritmo 3 - Estrategia de exploración MotifCore [6]

acciones del sistema de la aplicación.

El término cobertura esquelética (skeleton coverage) se utiliza para las coberturas de grado más fino, alcanzado por la instrumentación gris y de caja blanca. En algunos casos, incluso cuando el código fuente no está disponible, es posible una cobertura de grano grueso al nivel de método, que denominamos cobertura esquelética de la columna vertebral. Esta cobertura de la columna vertebral se puede lograr decompliando (undexing) el archivo APK e inlineando y luego re empaquetando el archivo binario. En el caso donde el código fuente está disponible, se usa la cobertura tradicional de statement (que sapienz llama cobertura total del esqueleto).

Implementación

Sapienz está implementado sobre el framework Deap [14] y alcanza la cobertura total del esqueleto (cobertura de statements) usando EMMA [11] y cobertura de la co-

lumna vertebral (cobertura del método) usando ELLA [10]. Calcula la cobertura de piel (cobertura de actividad) llamando a *ActivityManager* de Android para extraer información de la activity/screen. Para los genes atómicos, Sapienz soporta 10 tipos de eventos atómicos que provienen del sistema Android, incluyendo *Touch*, *Motion*, *Rotation*, *Trackball*, *PinchZoom*, *Flip*, *Nav* (tecla de navegación), *MajorNav*, *AppSwitch* y operaciones del sistema como silenciado del volumen finalizar llamada. En cuanto a los genes motif, hay una amplia gama de opciones para estos patrones, y se distinguen entre los que son genéricos (aplicable a todas las aplicaciones) y los que están a medida (aplicable sólo a un pequeño conjunto homogéneo de aplicaciones). Para el propósito del paper Sapienz evita tener distintos genes motif, ya que estos requieren la intuición humana e inteligencia. Sapienz provee sólo un gen motif que sistemáticamente ejecuta campos de texto y widgets de la interfaz de usuario los cuales se pueden hacer click.

Primero rellena todos los campos de texto con strings y luego intenta ejecutar cada widget clickable para transferir a la siguiente vista. Dicho patrón motif podría ser utilizado en escenarios tales como introducir inputs y enviar un formulario. Sapienz hace referencia a que con una selección más inteligente y personalizada de patrones motif para aplicaciones particulares, podría mejorar notablemente el resultado de desempeño.

Para finalizar Sapienz genera un conjunto de artefacts para su reutilización, incluyendo conjuntos de test reutilizables, informes detallados de cobertura e informes de fallos.

2.4.3. Monkey

Monkey es una herramienta de línea de comando que puede ser corrida en una emulador o en un dispositivo y es parte del SDK de Android [5].

Básicamente envía secuencias de eventos aleatorios como ser clicks, rotaciones, gestos, mutear el teléfono y muchos más para poder generar test de stress en una aplicación deseada. Monkey da la posibilidad de configurar opciones como ser la cantidad de eventos que se desean realizar, qué tipos de eventos utilizar y con qué frecuencia. Cuando Monkey se ejecuta genera eventos que son enviados al sistema, el mismo también observa la aplicación bajo test teniendo la habilidad de parar la ejecución y reportando el error si la aplicación crashea, recibe algún tipo de excepción no manejada o se bloquea.

3. EXPERIMENTACIÓN

3.1. Replicación del experimento

Para poder replicar la experimentación nos pusimos en contacto con uno de los autores del paper de Mining Sandboxes (Konrad Jamrozik) [1]. El armó el repositorio que contiene los datos necesarios para replicar el experimento lo más preciso posible. Aqui (https://github.com/konrad-jamrozik/mining-sandboxes-icse2016) podrán encontrar los datos, tanto de inputs y outputs necesarios para replicar ese experimento y a su vez obtener las versiones de apks utilizadas en nuestra experimentación

Benchmarks utilizado

Para hacer la comparación de las 3 herramientas de testing automático, tomamos como punto de referencia a algunas de las aplicaciones (apks) utilizadas en Mining Sandboxes [1]. Esta experimentación fue realizada el 12 de agosto del 2015 con lo cual las aplicaciones datan de fechas previas. Decimos esto, para argumentar que fue imposible correr todas las aplicaciones utilizadas o obtener reproducciones exactas a los resultados obtenidos en el paper.

¿Con qué problemas nos topamos a la hora de hacer la reproducción de dichas aplicaciones?

- La experimentación requiere inlinear cada apk, el cual modifica su bytecode. Algunas aplicaciones, como ser Snapchat, introdujeron chequeos remotos de integridad de bytecode desde que la experimentación fue ejecutada, por esta razón su utilización en la experimentación tuvo distintos resultados. Recordemos que al no poderse inlinear las aplicaciones, no podemos recolectar las llamadas a las APIs durante la exploración de cada herramienta de testing automática.
- En el caso de la aplicación Snapchat, las ejecuciones realizadas por Droidmate en Mining Sandboxes [1] utilizaban credenciales de login. La version de Droidmate que utilizamos no contaba con la posibilidad de loguearse. Dicho diferencia ocasionó que la experimentación de las herramientas que ejecutamos obtenga solo datos previo al login.
- Una gran parte del comportamiento de cada aplicación podría depender en mayor parte de un entorno externo, como ser un servidor remoto. Dicho esto, tanto validaciones como procesos lógicos podrían haber cambiado remotamente o aún peor, que dichos servidores estar caídos o modificados de tal forma que no se le de mas soporte a aplicaciones con versiones antiguas.

¿Por qué utilizamos estas Apks en vez de otras?

Al elegir el mismo set de aplicaciones que se utilizó en Mining Sandboxes [1], y a su vez intentar replicar la experimentación realizada por dicho paper con Droidmate, facilita a la hora de darle sostén a nuestra experimentación con las 3 herramientas. Ya que si podemos obtener resultados similares a la experimentación de Droidmate, nos da una confianza sobre los resultados obtenidos utilizando nuestro entorno para dichas ejecuciones. Y de esta forma, también, valida que todas las herramientas estén corriendo correctamente. Otra razón es que dichas apks, según el paper Mining Sandboxes, fueron obtenidas de Google Play Store, las cuales eran las más descargadas en ese momento.

¿Que tiempo de ejecución se utilizo para cada herramienta?

Para poder reproducir el experimento e imitar lo que se hizo en el paper, cada apk fue ejecución durante 2 hs.
Si bien no está aclarado en el paper, se consultó con el autor y cada apk fue ejecutada varias veces, pero oficialmente fue una vez, según el autor todas las ejecución fueron iguales.
Por otro lado, observamos detalladamente los outputs de las ejecuciones realizadas por el paper, se ve claramente que algunas ejecuciones se cortaron o corrieron mal, y luego seguido a eso el output muestra una ejecución completa para dicha apk.
Por lo cual creemos que las ejecuciones tomadas por el paper para cada apk fueron aquellas que terminan exitosamente.
Para lograr una confianza estadística más grande, decidimos correr cada aplicación 10 veces. Esto evita sobresaltos y resultados sesgados por crasheos, errores humanos de configuración, etc.

Distinción de recursos

Como comentamos, la detección de permisos implica un previo proceso de inlineado, el cual sirve para poder agregar callbacks a la aplicación bajo test en cada recurso que nosotros deseemos, estos recursos son esencialmente una lista de métodos los cuales son gobernados por un permiso.

¿De dónde sale esta lista?

El conjunto de métodos que fue utilizado en nuestra experimentación, es el mismo del paper de Mining Sandboxes con una **leve variación**.
Dicha lista es un conjunto de 97 métodos pertenecientes a distintos servicios que provee Android.
Por lo que pudimos hablar con uno de los autores, la lista de métodos de API utilizados en el paper se basó en el conjunto de métodos API de la versión antigua de AppGuard privacy-control framework [6].
Como dicha lista estaba obsoleta cuando la consiguió (puesto que utilizó una versión más nueva), tuvo que hacer un proceso de sanitización manual basado en la documentación de

Android, para así evitar los métodos deprecados o llamadas que no eran interesantes a su entender (desde el punto de vista de la seguridad).
Nos explico, que hizo esto puesto que varios métodos contaminaban las observaciones hechas en su paper y que no tenían sentido a la hora de mostrar el mensaje de fondo.

¿Que aspecto tienen dichos métodos?

Cada declaración en esta lista, cuenta con el nombre del método que se desea detectar, toda la ruta de su librería, incluyendo la clase, sus parámetros con sus tipos y su objeto de retorno con su tipo.

¿Cual es esta leve variación en la lista?

En el proceso de intentar replicar la experimentación realizada en Mining Sandboxes, observamos que los outputs generados por Droidmate en dicho paper (para algunas apks), incluía dos métodos que no estaban declarados en esa lista. Estos métodos de más, generaban que se obtuvieran menos métodos encontrados que en el paper en el recuento final.
Los métodos olvidados son:

```
- android.content.ContentResolver.query(android.net.Uri,java.lang.String[],
    java.lang.String,java.lang.String[],java.lang.String,
    android.os.CancellationSignal)

- java.net.Socket.<init>()
```

3.2. Herramienta de experimentación ApkTester

Para ejecutar las pruebas creamos la aplicación APKTester que permite automatizar la ejecución de múltiples APKs en las diferentes herramientas a testear. En APKTester definimos un directorio de donde se van a sacar las aplicaciones y elegimos cuál es la herramienta a usar. Además, podemos automatizar el inlineado de las mismas que va a permitir monitorearlas si no lo estaban previamente.

Para correr las aplicaciones en APKTester, tenemos dos componentes importantes, un runner que es el que se encarga de ejecutar las aplicaciones y que depende de la herramienta a testear, y un daemon que es un hilo aparte que durante la ejecución extrae a intervalos regulares los logs que contienen los métodos sensibles obtenidos generados por la aplicación monitor.apk llamada por el proceso de inlining. Todos los runners heredan de una clase abstracta que define los pasos de la ejecución y permite que cada herramienta defina en cada paso cuáles son las acciones a tomar (por ejemplo, antes de la ejecución, con Monkey necesitamos instalar la aplicación, pero en Droidmate y Sapienz no porque de eso ya se encargan las herramientas mismas). A la vez, de esta manera permitimos

que cada runner lance la herramienta y configure el límite de tiempo como corresponde (Droidmate ya tiene un límite de tiempo incorporado por lo que sólo debemos especificarlo en el archivo de configuración correspondiente, en los otros casos APKTester se encarga de terminar con la ejecución de la herramienta una vez superado el límite).

El daemon, como ya dijimos, se encarga de obtener el log que genera la aplicación. Para poder detectar los permisos, el inlining modifica el código para que cuando se llama a un método sensible se intercepte esa llamada. Al interceptarse, además de ejecutar el método normalmente se llama a una aplicación aparte, monitor.apk (generada por Droidmate y copiada antes de cada ejecución por APKTester al directorio /data/local/tmp, que es donde el código inlineado la busca). Esta aplicación, en el paper original se conecta por TCP al servidor que está corriendo Droidmate y le notifica el permiso ejecutado.

Para nuestra experimentación, nosotros modificamos esta APK para que en su lugar escriba en un archivo .log dentro de la SD del dispositivo el método de la API llamada; por lo tanto, en cada instante el archivo log contiene todas los métodos de APIs llamadas desde que se inició la ejecución. Cada diez segundos, el daemon obtiene ese archivo y lo copia al servidor corriendo la herramienta, por lo tanto, luego de la ejecución vamos a obtener un archivo de métodos ejecutados por cada diez segundos de ejecución. Como es muy probable que durante el testeo se haya llamado varias veces al mismo método de la API, lo que hacemos para cada archivo es sólo contar los únicos, y luego generamos un reporte en el cual colocamos cuántos métodos únicos se encontraron para cada instante y cuál fue la lista definitiva de métodos encontrados.

Los métodos escritos en los archivos de log contienen la signatura del mismo (nombre del método, clase a la que pertenece, parámetros de llamada y tipo de retorno) y los valores con los que se llamó. Si bien no estaba especificado en el paper, examinando los resultados obtenidos por Konrad encontramos que en algunos casos se consideraba diferentes permisos a llamadas del mismo método con diferentes parámetros. Esto sucede cuando el parámetro es del tipo *android.net.Uri*, por ejemplo, el método

```
android.content.ContentResolver.registerContentObserver
(android.net.Uri, boolean, android.database.ContentObserver)
```

que se utiliza para suscribirse a los cambios producidos en un contenido dado por la URI. Esta URI puede variar, por ejemplo, *content://sms* es usado para acceder a los mensajes de texto, es muy diferente a *content://com.android.browser/history*, el historial de navegación del navegador de Android, por lo cual deben considerarse dos métodos diferentes ya que muestran que la aplicación desea acceder a diferentes tipos de contenido restringido.

Todas las pruebas fueron ejecutadas en el emulador oficial de Android, simulando un Nexus 5 de 4.95çorriendo Android 4.4 (API 19) con 1GB de RAM y 200MB de tarjeta SD. Luego de cada ejecución se limpiaba la información del emulador para resetearlo al estado de fábrica. El emulador fue ejecutado en todos los casos en una MacBook Pro 2012, con procesador Intel i5 de 2.5GHz, 16GB de RAM y un SSD de 250GB.

4. RESULTADOS

4.1. Replicación del paper Mining Sandboxes

Los aplicaciones que se utilizaron para la comparación de las herramientas elegidas fueron las utilizadas en el paper de Mining Sandboxes como se dijo anteriormente. Estas son 14 aplicaciones, de las cuales 2 de ellas son dos versiones distintas de Snapchat. Si bien comprobamos que las dos versiones de Snapchat funcionan correctamente para las tres herramientas, en nuestras mediciones solo nos focalizaremos en la versión 5.0.34.6.

Cada una de las aplicaciones se ejecutó en diez ocasiones durante dos horas por cada herramienta, dando un total de 260 horas de ejecución por herramienta.

A continuación se detalla la lista de aplicaciones utilizada en la experimentación:

Nombre	Versión	Categoria	Nombre del paquete	¿Corrió exitosamente?
Adobe	11.1.3	Productividad	com.adobe.reader.apk	Si
Antivirus	3.6	Comunicación	com.antivirus.apk	Sí
Ebay	2.5.0.31	Compras	com.ebay.mobile.apk	Sí
ExpenseManager	2.2.3	Finanzas	at.markushi.expensemanager.apk	Sí
FileManager	1.16.7	Negocios	com.rhmsoft.fm.apk	Sí
JobSerach	2.3	Negocios	com.indeed.android.jobsearch.apk	Sí
PicsArt	4.1.1	Fotografia	com.picsart.studio.apk	Sí
ES Task Manager	1.4.2	Negocios	com.estrongs.android.taskmanager.apk	Sí
CleanMaster	5.1.0	Herramienta	com.cleanmaster.security.apk	No
Currency converter	1.02	Finanzas	com.frank.weber.forex2.apk	No
Barcoo	3.6	Compras	de.barcoo.android.apk	No
Firefox	28.0.1	Comunicación	org.mozilla.firefox.apk	No
Snapchat	5.0.34.6	Social	com.snapchat.android.apk	Sí

Como se puede observar en la columna 5 de tabla anterior, hay aplicaciones que no corrieron correctamente.
A continuación daremos el detalle de cada una de estas aplicaciones y cual fue la razón de dicha falla:

- Currency Converter

  El inlineado y la ejecución de la apk fue correcta, se pudo observar que dicha aplicación parecía ejecutarse correctamente, navegando por las diferentes vistas de la misma sin ningún problema.
  Por alguna razón que no entendemos, no se detectaron permisos en toda la ejecución. Como todas las aplicaciones, se corrieron 10 veces para cada apk y para cada

herramienta de testing automático. Y aun así obtuvimos el mismo resultado. Por esta razón fue descartada en la comparación de la experimentación.

- Barcoo

  Lo mismo que ocurrió con Currency Converter paso con esta aplicación, corrió exitosamente pero no se encontró ningún método declarado en la lista.

- Clean Master

  No soporta la arquitectura del emulador que usamos, por lo que no puede ejecutarse

- Firefox

  Se produce un crash cada vez que arranca la aplicación, lo cual hace imposible correrla. Por esta razón fue descartada.

Permisos obtenidos

Para poder saber si nuestra experimentación fue lo bastante cercana a lo presentado en el paper Mining Sandboxes, hicimos una tabla con todos los métodos encontrados por nuestras herramientas (Droidmate, Monkey y Sapienz) y los encontrados por Droidmate del paper.

De esta forma pudimos visualizar e ir puliendo nuestra proceso de obtención de métodos con la herramienta creada por nosotros ApkTester. Hasta que al terminar las 10 iteraciones de cada aplicación, se pudo obtener un resumen de los métodos obtenidos por todas las aplicaciones.

Por motivos de espacio las tablas y la explicación de las mismas se muestran en la sección Apéndice. En dicha sección se muestra qué método fue encontrado por cada herramienta de testing automático al ejecutar una determinada aplicación.

4.2. Cobertura

P1: ¿Qué herramienta logra cubrir la mayor cantidad de comportamiento de las aplicaciones?

Para responder esta pregunta, ejecutamos 10 corridas de dos horas cada una para cada aplicación, tomando cada 10 segundos las llamadas a APIs y extrayendo para cada medición la cantidad de métodos únicos.

Dadas todas las corridas, nos interesa encontrar qué herramienta logra cubrir un mayor comportamiento de las aplicaciones. Para esto, vamos a comparar cuál encuentra más metodos sensibles luego de dos horas de corrida. Una vez obtenidas diez corridas de cada aplicación, vamos a realizar un boxplot de los métodos encontrados para cada aplicación.

4.2.1. Gráficos

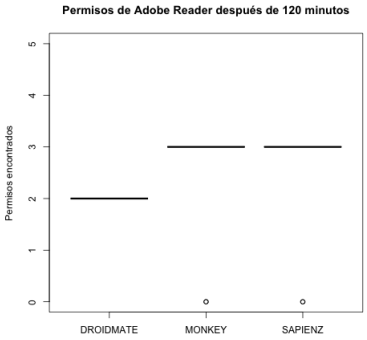

Fig. 4.1: Adobe Reader

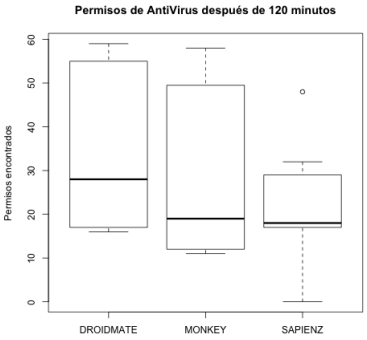

Fig. 4.2: Antivirus

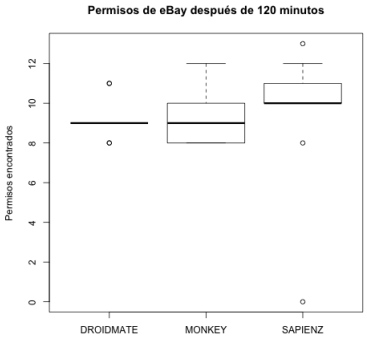

Fig. 4.3: eBay

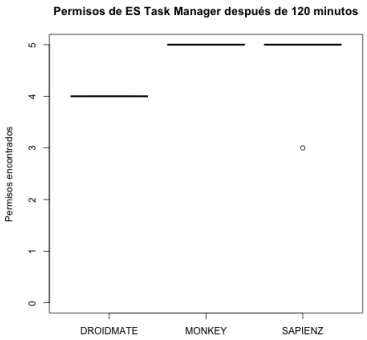

Fig. 4.4: ES Task Manager

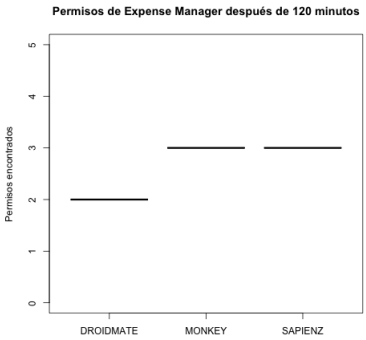

Fig. 4.5: Expense Manager

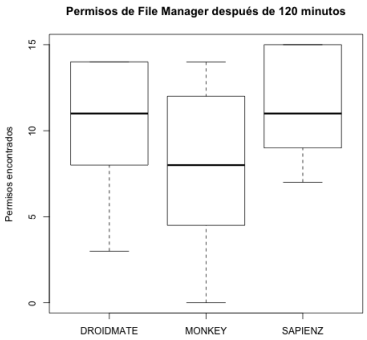

Fig. 4.6: File Manager

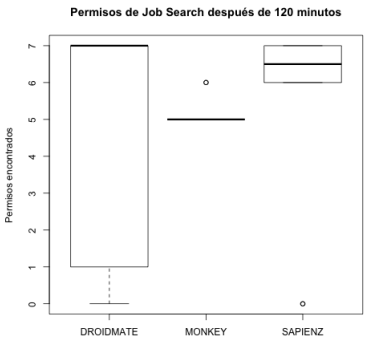

Fig. 4.7: Job Search

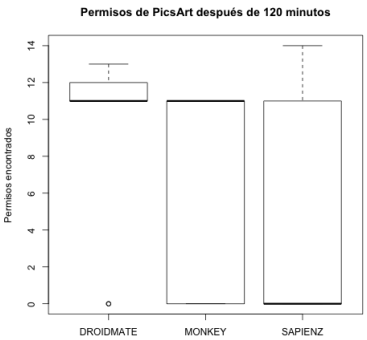

Fig. 4.8: PicsArt

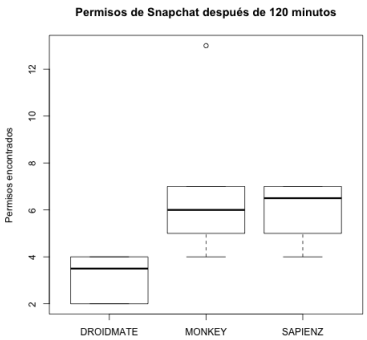

Fig. 4.9: Snapchat

Para entender estos boxplots, primero cabe aclarar el significado de estos datos.

La franja más ancha representa la mediana, es decir, el valor central de todos los datos. La caja que lo rodea tiene en el extremo superior al tercer cuartil (es decir, el valor que representa el 75 % de los datos) y en el inferior al primer cuartil (el 25 %). Hacia arriba y abajo de estos se extienden los bigotes (whiskers), que se definen como los valores que se encuentran hasta el valor máximo (o mínimo) que se encuentre dentro de un rango de 1.5 veces la diferencia intercuartil (la diferencia entre el tercer y el primer cuartil) de los extremos de la caja. Cualquier otro valor se grafica con un círculo, es un outlier

Dicho esto, teniendo en cuenta que para todas las aplicaciones ejecutamos 10 corridas, podemos poner en contexto estos números. La mediana, al ser el número de ejecuciones par, no representa un total obtenido sino el promedio entre quinto y sexto valor, ordenados de forma creciente. De la misma manera, el primer y tercer cuartil representan el tercer y octavo valor respectivamente, mientras que los bigotes se extienden como máximo hasta 1.5 veces la diferencia entre estos dos últimos valores, dejando los outliers afuera

En primer lugar lo que se observa es la gran disparidad en los resultados, 10 de los 27 boxplots (tres para cada aplicación) muestran al menos un outlier. De estos, 2 pertenecen a Monkey, 3 a Droidmate y 5 a Sapienz. Esto nos llama bastante la atención porque nos indica que Sapienz es bastante variable y más propenso a corridas que dan resultados totalmente alejados del resultado promedio. De estos cinco outliers, sólo uno muestra cero permisos encontrados (que podemos atribuir a un error en la corrida), el resto efectivamente encontró una cantidad de permisos muy alejado del promedio de corridas.

Aún cuando no sean outliers, notamos que en cinco de las nueve aplicaciones en al menos una corrida de Sapienz no se encontró ningún permiso (comparado con tres aplicaciones en Monkey y Droidmate con cero permisos en al menos una corrida). Esto evidencia que Sapienz es plausible de tener más corridas interrumpidas que el resto de las herramientas.

Si analizamos la mediana de los gráficos, podemos ver que en tres casos la mediana es igual en Sapienz y Monkey y mayor que Droidmate, en un caso es máxima entre Droidmate y Monkey, en uno entre Droidmate y Sapienz, y luego en dos casos la máxima pertenece exclusivamente Droidmate y otros tantos casos a Sapienz.

En los tres casos en que Droidmate tiene una performance peor que Monkey y Sapienz, estas son aplicaciones que se encontraron muy pocos permisos en general. Por otra parte, en las tres aplicaciones con mediana de permisos más alta (Antivirus, FileManager y PicsArt), Droidmate es uno de los encuentra más permisos en las tres.

Además, generamos un gráfico donde agrupamos las corridas de cada aplicación por número de corrida (es decir, la primera corrida de Antivirus, eBay, Snapchat, etc de cada herramienta, la segunda corrida de cada una, etc) y obtenemos los permisos únicos que se encontraron (sólo contamos una vez cada permiso diferente). De esta manera podemos generar un nuevo boxplot donde podemos observar cuál herramienta encuentra más permisos totales.

En este gráfico podemos observar que Droidmate es el que encontró más permisos en total, tanto teniendo en cuenta el máximo absoluto como la mediana de cada

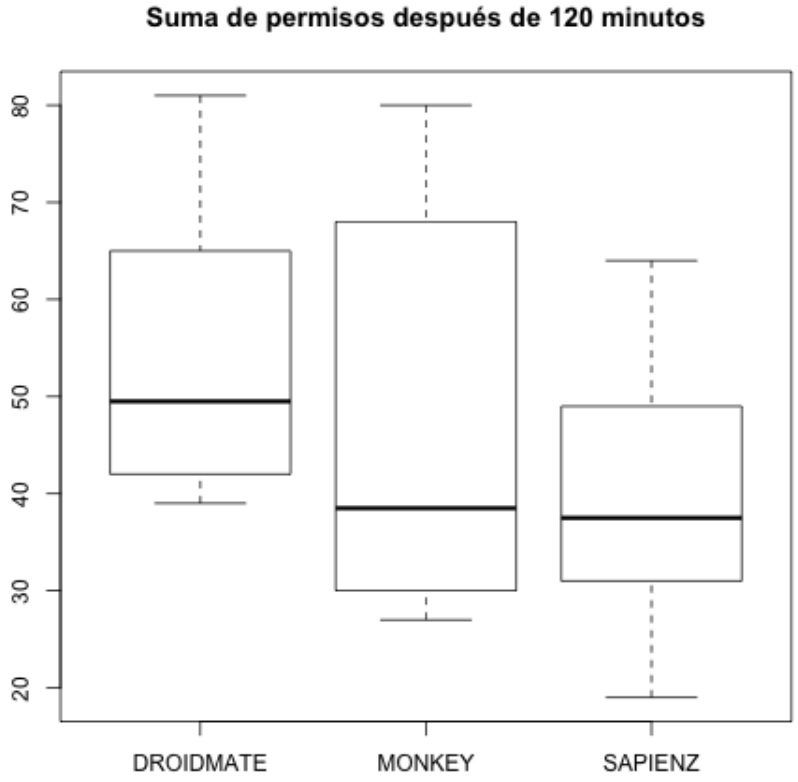

Fig. 4.10: Permisos totales por corrida

medición. Esto tiene mucho sentido considerando que previamente habíamos concluido que Droidmate se comportaba mejor con las aplicaciones que tienen muchos permisos para encontrar; en la corrida con mayor cantidad de permisos encontrados (81) el 72 %, es decir 59 permisos, aparece en AntiVirus

Esta aplicación AntiVirus afecta un poco la comparación, puesto que al tener muchos más permisos encontrados comparado con el resto termina dominando la métrica, en esencia, una corrida con muchos permisos encontrados de esa aplicación influye mucho más que el resto. Para una futura experimentación sería deseable usar un set de aplicaciones donde los permisos a encontrar estén mucho mejor distribuidos a lo largo de las mismas

En resumen, observamos que Droidmate se comporta muy bien con aplicaciones que tienen muchos permisos por encontrar, en Sapienz obtuvimos buenos resultados pero a la vez muy variables con muchas corridas que no encuentran permisos por fallas de la herramienta. En Monkey no obtuvimos resultados demasiado alejados de las otras herramientas, lo cual es un punto a favor considerando su facilidad de uso (no requiere configuración, ya viene incluido en el SDK de Android)

4.2.2. Tamaño del efecto

Para complementar el análisis, vamos a tratar de entender qué tan significativos son estos resultados que obtuvimos. Para esto, vamos a calcular el tamaño del efecto de Vargha-Delaney de los resultados. Para esto, primero vamos a explicar en qué consiste este coeficiente.

El tamaño del efecto de Vargha-Delaney [17], expresado como $\hat{A}_{12}$ es muy usado cuando se trata de medir algoritmos con un componente de azar, y denota la probabilidad que, dada una métrica de performance M, el algoritmo A obtenga mejores valores de M que el algoritmo B. Si los algoritmos son equivalentes, $\hat{A}_{12} = 0{,}5$. La fórmula para calcular el coeficiente de A es:

$$\hat{A}_{12} = (R_1/n_1 - (n_2 + 1)/2)/n_1$$

Donde n_1 y n_2 son la cantidad de mediciones obtenidas para A y B respectivamente, y R_1 es la suma de los rangos de los resultados de B. Es decir, que si A obtuvo los valores $\{4, 10, 8\}$ y B los valores $\{1, 5, 11\}$, los valores de A tienen, en el conjunto total de resultados, ordenados de menor a mayor, las posiciones 2, 4 y 5, por lo que $R_1 = 11$. En este ejemplo, el coeficiente de Vargha-Delaney daría 0,556.

En el paper de Vargha-Delaney [18] se muestran cómo interpretar los valores de este coeficiente. En este paper vemos que un valor de 0,56 indica un tamaño de efecto pequeño, 0,64 uno medio y 0,71 un tamaño grande, lo que nos permite inferir qué tanto es superior (estocásticamente, es decir, que tiene probabilidad de dar mejores resultados) un algoritmo a otro.

Por supuesto, como con cualquier coeficiente de probabilidad, para poder ponerlo en contexto es necesario entender qué poder tiene ese valor. Para eso, para cada coeficiente queremos calcular los intervalos de confianza del mismo, es decir, el intervalo en el que esperamos que el valor obtenido se encuentre con cierta probabilidad (en nuestro caso, 95 %).

El intervalo de confianza α para el coeficiente Vargha-Delaney se calcula de la siguiente manera: [18]

$$C_{1-\alpha} = d \pm x_{crit} s_d$$

con x_{crit} los valores críticos para el nivel de significancia de α

Para calcular los intervalos de confianza del coeficiente de Vargha-Delaney, nos vamos a valer primero de un coeficiente similar, el Delta de Cliff, cuya fórmula es [19]

$$\delta = \frac{\#(x_1 > x_2) - \#(x_1 < x_2)}{n_1 n_2}$$

En el paper de Vargha, vemos que pasar de δ a $\hat{A}_{12}$ se puede hacer de manera trivial:

$$\hat{A}_{12} = (\delta + 1)/2$$

El delta de Cliff también se puede obtener usando una matriz de dominación, de $n_1 \times n_2$, con filas y columnas c_{ij}, siendo a_i el i-ésimo resultado del experimento A (ordenado en forma creciente) e b_j el j-ésimo resultado de B tal que

$$c_{ij} = \begin{cases} 1, & \text{si } a_i > b_j \\ 0, & \text{si } a_i = b_j \\ -1, & \text{si } a_i < b_j \end{cases}$$

Para calcular los intervalos de confianza del delta de Cliff, vamos a basarnos en Hogarthy y Kromney [20].

Este método requiere primero calcular el estimado de la varianza de δ y ofrece tres tipos de estimados posibles para calcular, nosotros usaremos el estimado consistente. La fórmula es

$$S^2_{de} = \frac{(n_2 - 1)S^2_{di} - (n_1 - 1)S^2_{dj} + S^2_{dij}}{n_1 n_2}$$

donde $S^2_{di} = \frac{\sum(d_i - \delta)^2}{n_1 - 1}$ con d_i el valor marginal de la fila i

$S^2_{dj} = \frac{\sum(d_j - \delta)^2}{n_2 - 1}$ con d_j el valor marginal de la columna j

y $S^2_{dij} = \frac{\sum\sum(d_{ij} - \delta)^2}{(n_1 - 1)(n_2 - 1)}$

Con esta varianza consistente se pueden calcular los intervalos de confianza como:

$$CI = \frac{\delta - \delta^3 \pm Z_{\alpha/2} S_{dc}((1 - d^2)^2 + Z^2_{\alpha/2} S^2_{dc})^{1/2}}{1 - d^2 + Z^2_{\alpha/2} S^2_{dc}}$$

con $Z_{\alpha/2}$ la variable normalizada que corresponde al $1 - (\alpha/2)$-ésimo percentil de la distribución normal

Además del intervalo de confianza, se puede calcular el p-valor del coeficiente de la siguiente manera:

$$pv = 2(1 - pn(\left|\frac{\delta}{S_{dc}}\right|))$$

con pn la función de densidad de la normal

Es importante recalcar que, tal como se pueden convertir el Delta de Cliff al tamaño del efecto de Vargha-Delaney con la transformación que enunciamos arriba, se

puede hacer la misma transformación para los intervalos de confianza, por lo que vamos a expresar todos nuestros valores basados en este último coeficiente.

Vamos a comparar entre sí, de a dos, los resultados obtenidos para las herramientas para entender cuál es mejor para cada aplicación. Para esto vamos a tomar la cantidad de métodos encontrados en cada corrida para las diferentes aplicaciones después de 120 minutos y vamos a calcular el tamaño del efecto para cada uno, además de sus intervalos de confianza y p-valores. Además, mostramos el promedio de los permisos encontrados en las diez corridas para cada herramienta, por aplicación.

Droidmate vs Sapienz

Nombre APK	Promedio de permisos Droidmate	Promedio de permisos Sapienz	$\hat{A}_{12}$	Intervalos de confianza	p-valor
Adobe Reader	2	2,4	0,2	[0,053, 0,528]	**0.025**
AntiVirus	33	21,3	0,695	[0,426, 0,875]	0,127
eBay	9,2	9,4	0,3	[0,114, 0,588]	0,075
ES Task Manager	4	4,8	0,1	[0,016, 0,425]	**0.04**
Expense Manager	2	3	0	–	**0**
File Manager	10,4	11,6	0,38	[0,170, 0,647]	0,386
Job Search	5,1	5,3	0,575	[0,345, 0,777]	0,06
PicsArt	9,3	3,8	0,695	[0,424, 0,876]	0,130
Snapchat	3,2	5,9	0,05	[0,012, 0,187]	**0**

Monkey vs Sapienz

Nombre APK	Promedio de permisos Monkey	Promedio de permisos Sapienz	$\hat{A}_{12}$	Intervalos de confianza	p-valor
Adobe Reader	2,7	2,4	0,509	[0,340, 0,676]	0,92
AntiVirus	32,8	21,3	0,563	[0,306, 0,791]	0,08
eBay	9,3	9,4	0,332	[0,144, 0,594]	0,067
ES Task Manager	5	4,8	0,55	[0,451, 0,645]	0,32
Expense Manager	3	3	0,5	–	**0**
File Manager	7,9	11,6	0,254	[0,100, 0,513]	**0.033**
Job Search	5,2	5,3	0,23	[0,072, 0,536]	**0.041**
PicsArt	6,6	3,8	0,59	[0,345, 0,797]	0,067
Snapchat	6,5	5,9	0,5	[0,262, 0,738]	1

Droidmate vs Monkey

Nombre APK	Promedio de permisos Droidmate	Promedio de permisos Monkey	$\hat{A}_{12}$	Intervalos de confianza	p-valor
Adobe Reader	2	2,7	0,182	[0,048, 0,493]	**0.01**
AntiVirus	33	32,8	0,564	[0,334, 0,806]	0,5
eBay	9,2	9,3	0,545	[0,306, 0,766]	0,07
ES Task Manager	4	5	0	–	**0**
Expense Manager	2	3	0	–	**0**
File Manager	10,4	7,9	0,627	[0,375, 0,825]	0,066
Job Search	5,1	5,2	0,7	[0,381, 0,898]	0,192
PicsArt	9,3	6,6	0,69	[0,458, 0,854]	0,084
Snapchat	3,2	6,5	0,025	[0,004, 0,149]	**0**

Recordemos que un tamaño del efecto menor a 0,5 significa que la segunda herramienta obtuvo mejores resultados que la primera, por ejemplo, en Droidmate vs Sapienz, el 0,2 obtenido para Adobe Reader nos indica que, según nuestras mediciones, Droidmate debería obtener mejores resultados que Sapienz un 20 % de las veces (y como son complementarios, Sapienz lo haría un 80 % de las veces).

Si tomamos como 0,05 como es habitual el máximo p-valor que aceptamos, observamos que en la mayoría de los casos el resultado no es significativo. En los casos que es significativo, Sapienz obtuvo mejores resultados que Droidmate en cuatro ocasiones, y Droidmate en ningún caso obtuvo una ventaja significativa. En la comparación Monkey-Sapienz, éste último obtuvos mejores resultados en dos aplicaciones, comparadas con cero de la primera. En la última comparación, Droidmate-Monkey, vemos que hay resultados significativos para cuatro aplicaciones, en dos casos el resultado es que encuentran la misma cantidad de métodos y en otros dos que Monkey es mejor.

De todo esto podemos concluir que es difícil encontrar resultados significativos con el experimento construido como tal. No obstante, existen aplicaciones con resultados muy claros donde este tamaño de muestra es suficiente para concluir que existe una herramienta que es superior a las otras (Sapienz en Expense Manager), o una que es inferior a las otras (Droidmate en Snapchat, Monkey en File Manager). Especialmente en las aplicaciones donde existen muchos métodos para encontrar (Antivirus, PicsArt) la gran varianza que tienen los resultados hace que no sea fácil encontrar resultados significativos con diez corridas.

4.3. Eficiencia

P2: ¿Qué herramienta es más eficiente en encontrar permisos?

No sólo queremos ver qué herramienta encuentra más métodos luego del tiempo asignado, sino también queremos ver cuál lo hace más rápido. Para esto, dadas las diez corridas de cada aplicación por herramienta usadas anteriormente, vamos a analizar en cada minuto cuántas APIs se habían encontrado en cada corrida y quedarnos con el máximo de esos 10 valores, para luego construir un gráfico de progreso para cada una.

4.3.1. Gráficos

Los resultados para las diferentes aplicaciones son los siguientes:

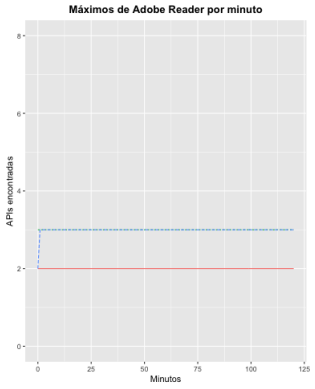

Fig. 4.11: Adobe Reader

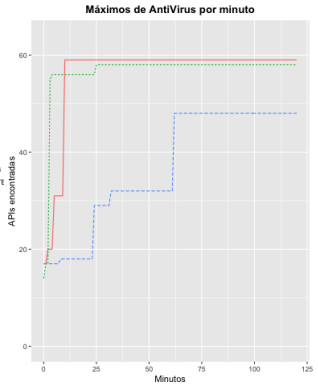

Fig. 4.12: Antivirus

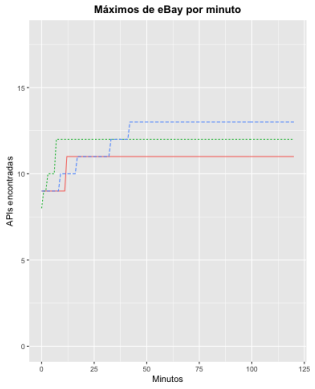

Fig. 4.13: eBay

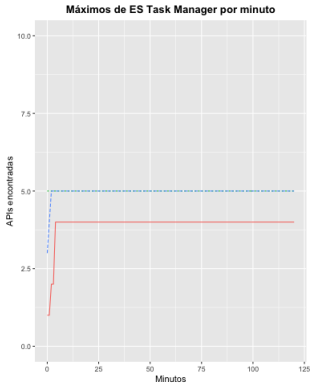

Fig. 4.14: ES Task Manager

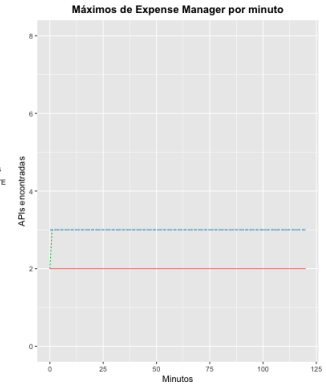

Fig. 4.15: Expense Manager

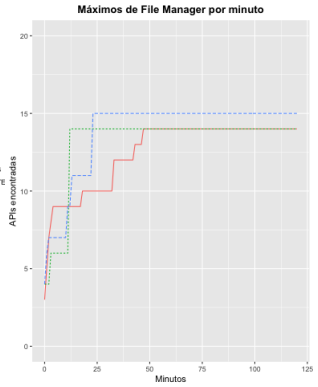

Fig. 4.16: File Manager

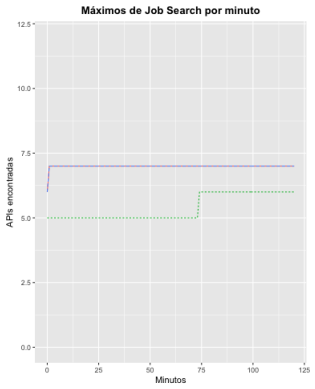

Fig. 4.17: Job Search

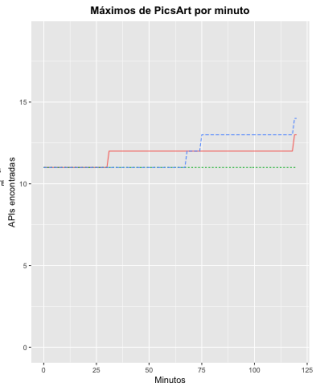

Fig. 4.18: PicsArt

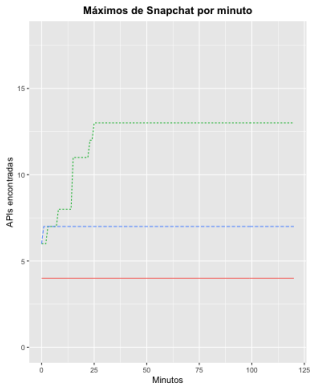

Fig. 4.19: Snapchat

De estos resultados podemos sacar varias conclusiones.

Si bien no es habitualmente el que más métodos encuentra, notamos que Monkey es con frecuencia el que más rápido localiza una cantidad importante de los mismos, por ejemplo, en el Antivirus, si bien Droidmate termina encontrando más métodos al final, Monkey a los pocos minutos ya encontró más de 50 métodos mientras que las otras herramientas no alcanzaron los 30.

La explicación que encontramos para este fenómeno es que Monkey ejecuta muchas interacciones al azar todo el tiempo, con mayor velocidad de lo que lo hacen Sapienz y Droidmate; esto hace que pueda cubrir diferentes pantallas y acciones rápidamente. En aplicaciones con muchas métodos por encontrar, como el caso de Antivirus, Monkey es capaz de encontrar métodos más rápido que las otras herramientas.

Por otra parte, la total aleatoriedad de Monkey hace que en general no sea el que más métodos encuentra, como podemos ver, puesto que esta falta de "inteligencia" genera que se repita mucho en acciones ya ejecutadas.

En Droidmate, consiguientemente, se observa que es el que menos máximos alcanza, y el más lento en alcanzarlos en la mayoría de los casos. Una suposición que hacemos es que dado que la herramienta ejecuta menos acciones por minuto que las otras, es posible que no llegue a cubrir todos los casos que las otras sí logran.

Sapienz, finalmente, tuvo un desempeño peor en Antivirus, pero en el resto de las aplicaciones (salvo en Snapchat) es el que mayores máximos alcanza. Este caso es bastante particular y probablemente amerite un análisis más profundo, sobre si existe algún inconveniente con la instrumentación que se le hace a Sapienz, la cual pueda que le este impidiendo encontrar métodos que sí logran sus contrapartes.

En resumen, como respuesta a esta pregunta, pudimos observar que Monkey encuetra permisos más rápido que las otras herramientas en la mayoría de los casos, lo que lo hace muy eficiente en caso de que exista una limitación de tiempo para la aplicación a testear, en la mayoría de los casos luego de unos pocos minutos Monkey había encontrado más permisos que las otras. Por otro lado, Sapienz termina encontrando la mayor cantidad de máximos al final del tiempo establecido, lo cual lo convierte en una buena alternativa si el factor tiempo no es un problema. De la misma manera, Droidmate es la herramienta más lenta en encontrar permisos, por lo que no parece ser una buena alternativa ante limitaciones de tiempo para evaluar las aplicaciones.

4.3.2. Estado del Arte

P3: ¿Es el estado del arte apropiado para encontrar permisos en las aplicaciones?

Para responder esta pregunta, calcularemos el total de permisos encontrados contra los permisos definidos en el archivo Manifest por cada aplicación (apk).

Recordemos que el desarrollador debe declarar los permisos (constantes) en el archivo Manifest para poder utilizar funciones protegidas por Android. Un ejemplo de permiso es android.permission.ACCESS_FINE_LOCATION, el cual otorga acceso al servicio de Location que brinda Android.

En caso de no declarar en el Manifest dichas constantes, el sistema no le permitirá utilizar los métodos públicos de dicha API, generando una excepción en tiempo de ejecución.

Los permisos que fueron definidos en el Manifest por una aplicación, se pueden obtener directamente del apk del mismo, mediante el comando de consola adb, algunos de los permisos que retorna el comando, como se podrá ver más adelante, no son permisos específicos del sistema de Android (android.permission). Dichos permisos fueron removidos a la hora de calcular el porcentaje total de permisos encontrados vs el total de permisos definidos en el archivo Manifest de cada apk.

La lista de permisos encontrados que se comparará contra el conjunto declarado de permisos de cada apk, está formada con el número total de permisos distintos encontrados por cada herramientas de testing automáticas. Es decir que se tomaron todos los métodos detectados para las 10 corridas para cada herramienta automática de testing (Monkey, Sapienz, Monkey), para un determinado apk.

Un punto importante en esta evaluación fue encontrar qué permiso o grupo de permisos gobierna a los métodos detectados para cada apk bajo test, puesto que lo que se detectaron con **Appguard** son llamados a métodos, dichos métodos pertenecen a una API la cual puede requerir uno o más declaraciones de permisos para su utilización, como así también, varios métodos pueden que pertenezcan al mismo dominio de permiso.

El encontrar qué permiso, es decir, qué constante hay qué definir en el Manifest para poder utilizar un método de una API, implicó buscar en Google e indagar la documentación de Android, decimos esto puesto que no fue sencillo encontrar qué nombre de permiso le pertenece a cada método, en algunos casos la documentación de Android es clara y dice que para utilizar un método determinado debe declarar un permiso particular, pero en otros casos no lo menciona. De todas formas, en el total de métodos (99 en total) que sirvió para instrumentar a cada apk, sólo 3 (detectados en la experimentación con cada herramienta) no se le encontró a qué permiso/constante pertenecía.

La siguiente tabla muestra el porcentaje de permisos encontrados por cada apk. Estas apks son las que corrieron exitosamente en la sección de experimentación. Por otra parte en la columna 4 se muestra el total de los permisos encontrados sobre los permisos del sistema de Android, habiendo restado los permisos propios de negocio definidos por la

apk (columna 3) del total de permisos definidos en el Manifest (columna 2).

Nombre Apk	Permisos declarados por la app	Permisos que no son de android	Permisos encontrados	Nombre permisos encontrados	% permisos encontrados
Adobe	3	0	1 de 3	INTERNET	33.33 %
Antivirus	46	2	5 de 44	INTERNET GET_TASKS CHANGE_WIFI_STATE WAKE_LOCK READ_PHONE_STATE	11.36 %
eBay	12	2	2 de 10	ACCESS_FINE_LOCATION INTERNET	20.00 %
Expense Manager	5	2	1 de 3	INTERNET	33.33 %
File Manager	6	0	2 de 6	WAKE_LOCK INTERNET	33.33 %
Job Search	7	1	2 de 6	WAKE_LOCK INTERNET	33.33 %
PicsArt	13	1	3 de 12	ACCESS_FINE_LOCATION INTERNET WAKE_LOCK	25.00 %
ES Task Manager	19	0	3 de 19	CHANGE_WIFI_STATE READ_PHONE_STATE INTERNET	15.00 %
Snapchat	15	2	3 de 13	CAMERA READ_PHONE_STATE INTERNET	23.07 %

La explicación de este bajo porcentaje de permisos encontrados podría deberse a distintas razones. Algunas de las cuales describiremos a continuación:

1. El desarrollador de cada apk pudo haber definido más permisos en el Manifest de los que utilizó. Este punto es muy común y se estudia en el paper de Felt [9], el cual encontró que el 33 % de las aplicaciones de Android investigadas eran más privilegiadas, es decir, solicitaron más permisos de los que realmente requerirían.

2. El estado del arte no logra descubrir los suficientes métodos como para decir que el problema de detección de permisos está solucionado. La razón de esto pueden ser varios factores y depende intrínsecamente de las herramientas de testing automático que comparamos; puesto que no son lo suficientemente "inteligentes"de detectar más permisos.

3. La lista de métodos que se utilizó para inlinear cada apk está incompleta. El paper de Mining Sandboxes pudo haber omitido/olvidado métodos relacionados con permisos que requieren ser declarados en el Manifest para su utilización. Que por consiguiente repercutió en la obtención de métodos a descubrir por las herramientas de testing automático durante la experimentación, acarreando así un bajo porcentaje de permisos descubiertos.

Para probar este punto, tomamos una apk utilizada en la experimentación, de forma random, observamos los permisos utilizados en el Manifest y luego obtuvimos algunos métodos relacionados a dichos permisos, notando así que estos métodos nunca estuvieron incluidos en la lista de métodos a inlinear, es decir que no fueron tomados en cuenta a la hora de inlinear cada apk.

La aplicación que utilizamos para esta investigación fue **eBay**. La misma declara los siguientes permisos en el Manifest:

- android.permission.INTERNET
- android.permission.ACCESS_FINE_LOCATION
- android.permission.ACCESS_NETWORK_STATE
- android.permission.WAKE_LOCK
- android.permission.WRITE_EXTERNAL_STORAGE
- android.permission.RECEIVE_BOOT_COMPLETED
- android.permission.CAMERA
- android.permission.VIBRATE
- android.permission.NFC
- android.permission.GET_ACCOUNTS

Como se vio en la anterior tabla, sólo se detectaron para la aplicación **eBay**, algún método relacionado a los primeros dos permisos **INTERNET** y **ACCESS_FINE_LOCATION**. Como se mostrara a continuación, para algunos de los restantes permisos declarados en el Manifest por eBay, se obtuvo de la documentación varios de los métodos que están relacionados a cada uno de estos y chequeamos si dichos métodos fueron definidos en la lista para el proceso de inlineado.

Como podemos observar, muchos métodos que se detallan a continuación no fueron utilizados en dicha lista:

Para el permiso: **GET_ACCOUNTS**
Estos métodos no fueron utilizados:

- getAccounts
- getAccountsAsUser
- getAccountsByFeatures
- getAccountsForPackage
- hasFeatures

Para el permiso: **ACCESS_NETWORK_STATE**
Estos métodos no fueron utilizados:

- getNetworkQuotaInfo
- getDataLayerSnapshotForUid
- getActiveNetworkInfoForUid
- getActiveNetworkInfo

Para el permiso: **WRITE_EXTERNAL_STORAGE**
Estos métodos no fueron utilizados:

- getExternalCacheDir
- getExternalCacheDirs
- getExternalFilesDir
- getExternalFilesDirs
- getObbDir
- getObbDirs
- getExternalCacheDir

Para el permiso: **CAMERA**
Estos métodos no fueron utilizados:

- autoFocus
- startFaceDetection
- startPreview

Podemos señalar que en el caso de haber sido definido alguno de estos métodos en la lista de métodos a inlainear, a la vez dichos métodos utilizados en el código de la aplicación eBay y por ultimo las herramientas de testing automáticas ser capaces de alcanzar dichos métodos durante la experimentación, hubieran aparecido en la lista de métodos encontrados, incrementando así el porcentaje de permisos encontrados.

Para contestar esta pregunta, se puedo observar que el porcentaje de detección de permisos para cada apk fue muy baja, no pudiendo superar el 33,33Se dieron 3 posibles explicaciones de por que creemos que este porcentaje fue tan bajo. Con lo cual, se puede decir que el estado de arte no es suficientemente bueno para encontrar permisos en la aplicación.

4.3.3. Categorías de permisos

P4: Hay permisos que son más sensibles en relación a otros. ¿Que herramienta es mejor para encontrarlos?

Para contestar esta pregunta, vamos a analizar los diferentes tipos de permisos. Android caracteriza el riesgo potencial implícito de cada permiso e indica el procedimiento que debe seguir el sistema al determinar si se concede o no el permiso a una solicitud que lo requiera. El nivel de protección se establece de la siguiente forma:

- **Normal**

  Es un valor predeterminado. Son permisos de menor riesgo, que permite a las aplicaciones solicitantes acceder a funciones aisladas de nivel de aplicación, con un riesgo mínimo para otras aplicaciones, el sistema o el usuario. El sistema concede automáticamente este tipo de permiso a una aplicación solicitante en la instalación, sin pedir la aprobación explícita del usuario (aunque el usuario siempre tiene la opción de revisar estos permisos antes de instalarlo)

- **Danger**

  Son permisos de alto riesgo que daría a una aplicación solicitante el acceso a datos privados del usuario o el control sobre el dispositivo que pueda afectar negativamente al usuario. Dado que este tipo de permiso introduce un riesgo potencial, es posible que el sistema no lo conceda automáticamente a la aplicación solicitante. Por ejemplo, los permisos peligrosos solicitados por una aplicación pueden ser mostrados al usuario y requieren confirmación antes de continuar, o puede tomarse algún otro enfoque para evitar que el usuario automáticamente permita el uso de tales instalaciones.

- **Signature**

  Son permisos que el sistema concede sólo si la aplicación solicitante está firmada con el mismo certificado que la aplicación que declaró el permiso. Si los certificados coinciden, el sistema concede automáticamente el permiso sin notificar al usuario o solicitar la aprobación explícita del usuario.

- **SignatureOrSystem**

  Son permisos que el sistema otorga sólo a las aplicaciones que están en la imagen del sistema de Android o que están firmadas con el mismo certificado que la aplicación que declaró el permiso. Este permiso se utiliza para ciertas situaciones especiales en las que varios proveedores tienen aplicaciones integradas en una imagen del sistema y necesitan compartir funciones específicas.

  Como se puede ver en las definiciones anteriores, el permiso más relevante en aspectos de seguridad es el *Danger*.

  Basándonos en lo dicho anteriormente, responderemos la pregunta **P4** que fue planteada en la sección de introducción.

Pero.. ¿Cómo obtenemos que nivel de seguridad tiene cada permiso?

La documentación específica, en la mayoría de los casos, que nivel de seguridad tiene cada permiso. Por lo que fue relativamente sencillo obtener esta información. Para los casos en donde no encontramos esta información, decidimos tomar, como peor caso, a dicho permiso como Danger.

La siguiente tabla muestra la cantidad de permisos Danger que encontró cada herramienta de testing automático, para cada APK.

Nombre APK	Sapienz	Monkey	Droidmate
Adobe	1	1	0
AntiVirus	2	2	2
eBay	2	2	2
Expense Manager	1	1	1
File Manager	1	1	1
Job Search	1	0	1
PicsArt	2	2	2
ES Task Manager	2	2	1
Snapchat	2	2	2

Explicación de los datos obtenidos

Como se ha visto en la sección experimentación, cada aplicación bajo test se ejecutó 10 veces. Dichas ejecuciones alcanzaron un conjunto de métodos relacionados a permisos de Android.
Tomamos el conjunto total de métodos obtenidos para las 10 ejecuciones, para una herramienta de testing automático (Droidmate, Sapienz o Monkey). Luego para el conjunto resultante de métodos descubierto por dichas herramientas, obtuvimos a qué nivel de seguridad pertenece cada uno.

Siendo que la experimentación no mostró grandes diferencia de cantidad de permisos obtenidos por una u otra herramienta, no le vimos sentido mostrar por separado cada una de las 10 corridas. Como primera observación en la tabla anterior, se puede detectar que ninguna de las herramientas logra obtener una gran diferencia con respecto a las otras. Por consiguiente se podría destacar que Sapienz logra obtener una mínima diferencia de uno contra Monkey y dos contra Droidmate (en una sumatoria general).

Respondiendo a la pregunta planteada anteriormente, más allá de la mínima diferencia de Sapienz con el resto de las herramientas, y respondiendo a la pregunta planteada, no se nota que alguna de las herramientas sea superadora o tenga una mayor inteligencia para descubrir más cantidad de permisos *Danger*.

5. CONCLUSIONES

En esta tesis, presentamos un estudio comparativo entre 3 herramientas de testing automático (Droidmate, Sapienz, Monkey), en materia de detección de permisos para la plataforma Android.

Para poder lograr una comparacion fidedigna entre dichas herramientas, evaluamos cada una de estas con un conjunto de apks utilizadas por el paper **Mining Sandboxes**. Pudimos replicar la experimentación hecha en dicho paper, y obtener resultados muy semejantes a los publicados.

A la hora de la comparación, no obtuvimos resultados tan diferentes como creímos que iban a ser desde un principio. Más aún, pensamos que tanto Monkey como Sapienz iban a superar rotundamente a Droidmate por su robustez, y no solo no paso eso, si no que Droidmate saco muy buenos resultados para la ejecución de algunas apks. Luego de presentar los resultados de esta experimentación, respondimos algunas preguntas que nos habíamos planteado desde un principio con relación a dicha comparación. Expusimos cual de ellas logra descubrir más métodos relacionados a permisos; cual es más eficiente en encontrarlos y si alguna de ellas encuentra permisos más sensibles (nivel de protección) que otras.

También discutimos fortalezas y debilidades de cada herramienta e indagamos si dicha efectividad puede concluir que el estado del arte es suficiente para encontrar permisos en aplicaciones.

Creemos que existe un gran universo por explorar en materia de detección automática de permisos. Actualmente consideramos que los resultados no son suficientes para decir que el testing automático puede detectar todo el comportamiento de una aplicación respecto a los permisos que utiliza, sabemos que nuestro experimento está lejos de cubrir por completo este campo. En particular, con el nuevo paradigma de permisos en Android creado por Google a partir de la API 23 del sistema, esto puede provocar un salto hacia adelante en esta materia, al granularizar mucho más el pedido de permisos y cuándo se utilizan. Entendemos que este es un gran punto de partida a futuro para continuar nuestra investigación

Toda nuestra experimentación, infraestructura y datos está publicada en https://github.com/nravasi/apktester

6. APÉNDICE

6.1. Tablas - Métodos encontrados

Las siguientes tablas muestran los métodos encontrados por cada herramienta de testing automático para cada aplicación (apk).

Aclaraciones:

Las herramientas de testing automático se definen con las siguientes letras: M: Monkey D: Droidmate S: Sapienz P: Droidmate del paper.

Por razones de espacio para poder graficar las tablas, se removieron los parámetros de cada método, agregando un "... " en el caso que los tenga.

Lo que se puede observar, a grandes rasgos, es que en la mayoría de los casos en donde había una P, alguna de las herramientas de testing automático ejecutadas por nosotros encontró el mismo método. Las aplicaciones que tuvieron más discrepancia a la hora de matchear con los permisos obtenidos por el paper fueron Picsart y Snapchat. Esto, se pudo haber dado por varias razones, como se comentó en la sección de experimentación.

Cabe destacar que la lista de métodos utilizada para inlinear las aplicaciones es más amplia que los que se detallan en las tablas. La razón por la cuales dichos métodos fueron removidos de estas tablas fue que, ni Droidmate del paper, como las herramientas de testing automático ejecutadas por nuestro ApkTester, pudo ser capaz de encontrar dicho método. Por otro lado, en algunas celdas se puede visualizar una letra que tiene un número entre paréntesis, este significa cuántos métodos distintos de este tipo se encontraron. Puesto que el paper tomaba dicho método como un nuevo método encontrado si era invocado con distintos parámetro.

	Metodo	Antivirus	eBay	Expense Manager	File Manager
1	getRunningTasks(..)	D M S			
2	content.ContentResolver.bulkInsert(..)				
3	content.ContentResolver.delete(..)	D(38) M(39) S(12) P	D(2) M(2) P(2) S(2)		D(4) M(4) P(4) S(4)
4	content.ContentResolver.insert(..)		D M P S(2)		
5	content.ContentResolver.registerContentObserver(..)	D(8) M(8) P S(8)	D M S(2)	M S	D M S(2)
6	content.ContentResolver.update(..)				
7	hardware.Camera.open(..)				
8	hardware.Camera.open(..)				
9	location.LocationManager.getBestProvider(..)				
10	location.LocationManager.getLastKnownLocation(..)		D M P S		
11	location.LocationManager.isProviderEnabled(..)		D M P S		
12	location.LocationManager.requestLocationUpdates(..)				
13	location.LocationManager.requestLocationUpdates(..)		M		
14	location.LocationManager.requestLocationUpdates(..)		D M P S		
15	location.LocationManager.requestLocationUpdates(..)				
16	media.AudioRecord.¡init¿(..)				
17	net.wifi.WifiManager$WifiLock.acquire()				D M P S
18	net.wifi.WifiManager$WifiLock.release()				D M P S
19	net.wifi.WifiManager.setWifiEnabled(..)	D M S			
20	net.wifi.WifiManager.startScan()				
21	os.PowerManager$WakeLock.acquire()	D M S			D M P S
22	os.PowerManager$WakeLock.acquire(..)				
23	os.PowerManager$WakeLock.release(..)	D M S			D M P S
24	telephony.TelephonyManager.getDeviceId()	D M P S			
25	telephony.TelephonyManager.getLine1Number()	D M P S			
26	telephony.TelephonyManager.getSimSerialNumber()	D M P S			
27	telephony.TelephonyManager.listen(..)				
28	java.net.URL.openConnection()	D M P S	D M P S	D M S	D M S
29	java.net.Socket.¡init¿()	D M P S	D M P S	D M S	D M P S
30	content.ContentResolver.query(..)		P		P(2)

Tab. 6.1: Métodos encontrados en Antivirus, Ebay, Expense Manager y File Manager

	Metodo	JobSearch	PicsArt	Task Manager	Snapchat	Adobe
1	getRunningTasks(..)					
2	content.ContentResolver.bulkInsert(..)		P			
3	content.ContentResolver.delete(..)		P			
4	content.ContentResolver.insert(..)				P	
5	content.ContentResolver.registerContentObserver(..)	D S	D M P S	M S	M S	M S
6	content.ContentResolver.update(..)		P			
7	hardware.Camera.open(..)		P			
8	hardware.Camera.open(..)		P		D M P S	
9	location.LocationManager.getBestProvider(..)		D M P S			
10	location.LocationManager.getLastKnownLocation(..)		D M S			
11	location.LocationManager.isProviderEnabled(..)		D M P S		P	
12	location.LocationManager.requestLocationUpdates(..)		P			
13	location.LocationManager.requestLocationUpdates(..)					
14	location.LocationManager.requestLocationUpdates(..)		D M S			
15	location.LocationManager.requestLocationUpdates(..)		D M S			
16	media.AudioRecord.¡init¿(..)				P	
17	net.wifi.WifiManager$WifiLock.acquire()					
18	net.wifi.WifiManager$WifiLock.release()					
19	net.wifi.WifiManager.setWifiEnabled(..)			D M P S		
20	net.wifi.WifiManager.startScan()					
21	os.PowerManager$WakeLock.acquire()	D M S	M			
22	os.PowerManager$WakeLock.acquire(..)				P	
23	os.PowerManager$WakeLock.release(..)	D M S			P	
24	telephony.TelephonyManager.getDeviceId()			D M P S	D M S	
25	telephony.TelephonyManager.getLine1Number()				S P	
26	telephony.TelephonyManager.getSimSerialNumber()					
27	telephony.TelephonyManager.listen(..)			D M P S		
28	java.net.URL.openConnection()	D M S	D M P S	D M P S	S P	D M S
29	java.net.Socket.¡init¿()	D M S	D M P S	D M P S	D M P S	D M P S
30	content.ContentResolver.query(..)	P	P(2)			

Tab. 6.2: Métodos encontrados en Job Search, PicsArt, Task Manager, Snapchat y Adobe

6.2. Código fuente y resultados

La herramienta APKTester que usamos para comparar las tres alternativas se puede encontrar en https://github.com/nravasi/apktester

Para correrla primero se deben configurar los paths a las herramientas y SDK de Android, ya sea modificando el archivo Config.groovy o sobreescribiendo la configuración en apktester.properties. De la misma manera se puede especificar la herramienta a correr, la cantidad de corridas y los minutos a correr por corrida.

En el mismo repositorio se pueden encontrar los resultados obtenidos, en la carpeta results. Allí están los csv usados para generar los gráficos de cobertura, máximos y de sumas, y dentro de la misma en la carpeta summaries los resúmenes de cada corrida, en dos archivos, uno (summary) que contiene los métodos únicos encontrados cada 10 segundos, y el otro, (summaryMethods) que contiene el listado de los métodos únicos encontrados

Bibliografía

[1] Jamrozik, K., von Styp-Rekowsky, P., Zeller, A., *Mining Sandboxes*. Proceedings of the 38th International Conference on Software Engineering, pp 37-48, 2016

[2] Backes, M., Bugiel, S., Hammer, C., Schranz, O., von Styp-Rekowsky, P., *Boxify: Full-fledged app sandboxing for stock android*. 24th USENIX Security Symposium, USENIX Security 15, pp. 691–706, 2015

[3] MacHiry, A., Tahiliani, R., Naik, M., *Dynodroid: An input generation system for Android apps*. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2013, ACM, pp. 224–234, 2013

[4] Backes, M., Gerling, S., Hammer, C., Maffei, M., von Styp-Rekowsky, P., *AppGuard–fine-grained policy enforcement for untrusted Android applications*. Data Privacy Management and Autonomous Spontaneous Security, J. Garcia-Alfaro, G. Lioudakis, N. Cuppens-Boulahia, S. Foley, and W. M. Fitzgerald, Eds., Lecture Notes in Computer Science. Springer Berlin Heidelberg pp. 213–231, 2014

[5] https://developer.android.com/studio/test/monkey.html

[6] Mao, K., Harman, M., Jia, Y., *Sapienz: multi-objective automated testing for Android applications*. Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016). ACM, pp. 94-105, 2016

[7] Lu, L., Li, Z., Wu, Z., Lee, W., Jiamg, G., *Chex: Statically vetting android apps for component hijacking vulnerabilities*. Proceedings of the 2012 ACM Conference on Computer and Communications Security (New York, NY, USA, 2012), CCS '12, ACM, pp. 229–240., 2012

[8] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P., *FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps*. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (New York, NY, USA, 2014), PLDI '14, ACM, pp. 259–269, 2014

[9] Felt, A. P., Chin, E., Hanna, S., Song, D., Wagner, D. *Android permissions demystified*. Proceedings of the 18th ACM Conference on Computer and Communications Security (New York, NY, USA, 2011), CCS '11, ACM, pp. 627–638, 2011

[10] ELLA: A tool for binary instrumentation of Android apps. http://github.com/saswatanand/ella.

[11] EMMA: A free Java code coverage tool. http://emma.sourceforge.net.

[12] Choudhary S.R., Gorla A., Orso, A., *Automated test input generation for Android: Are we there yet?*. Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 429-440, 2015.

[13] Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., *A fast and elitist multiobjective genetic algorithm: NSGA-II.* IEEE Transactions on Evolutionary Computation, 6(2):182-197, 2002.

[14] Fortin, F.A., De Rainville, F.M., Gardner, M.A., Parizeau, M., Gagne, C., *DEAP: Evolutionary algorithms made easy.* Journal of Machine Learning Research, 13:2171-2175, 2012.

[15] Harman, M., *The current state and future of search based software engineering.* Proceedings of FOSE '07 2007 Future of Software Engineering, pp 342,357, 2007.

[16] Harman, M., Mansouri, A., Zhang, Y., *Search based software engineering: Trends, techniques and applications.* ACM Computing Surveys, 45(1):11:1, 2012.

[17] Arcuri A., Briand L., *A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering.* Software Testing, Verification & Reliability, 24:3:219-250, May 2014

[18] Vargha, A., Delaney, H., *A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong.* Journal of Educational and Behavioral Statistics 25:2:101-132, 2000

[19] Macbeth, G., Razumiejczyk, E., Ledesma, R.D., *Cliff's Delta Calculator: A nonparametric effect size program for two groups of observations.* Universitas Psychologica, 10 (2), pp 545-555, 2011

[20] Hogarty, K. Y., Kromrey, J. D., *Using SAS to calculate tests of Cliff's Delta.* Poster presentado en 24th SAS Users Group International Conference. Paper 238, 1999

[21] Alshahwan, N., Harman, M., *Automated Web application testing using search based software engineering* Proceedings of ASE'11, pp. 3–12, 2011.

[22] Fraser, G., Arcuri, A., *Whole test suite generation.* IEEE Transactions on Software Engineering, 39(2):276–291, 2013.

[23] Bartel, A., Klein, J., Le Traoen, Y., Monperrus, M., *Automatically securing permission-based software by reducing the attack surface: An application to Android.* In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (New York, NY, USA, 2012), ASE 2012, ACM, pp. 274–277.

[24] Bläsing, T., Batyuk, L., Schmidt, A. D., Camtepe, S., Albayrak, S., *An Android application sandbox system for suspicious software detection* In Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on (Oct 2010), pp. 55–62.

[25] Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A. N., *TaintDroid: Aninformation-flow tracking system for realtime privacy monitoring on smartphones.* In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6

[26] Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weiss, Y. *Andromaly: a behavioral malware detection framework for android devices.* Journal of Intelligent Information Systems 38, 1 (2012), 161–190