UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

CREACIÓN DE UN SOFTWARE DE ANÁLISIS PARA MALWARE EN APLICACIONES DE ANDROID OS - SAMSARAA

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

NICOLÁS ALEJANDRO MALBRÁN CARNEVALI

PROFESOR GUÍA:
ALEJANDRO HEVIA ANGULO

MIEMBROS DE LA COMISIÓN:
JOSÉ MIGUEL PIQUER GARDNER
JORGE PÉREZ ROJAS

SANTIAGO DE CHILE
NOVIEMBRE 2012

Resumen

Los dispositivos móviles están cada vez más presentes en el día a día de las personas. A medida que evolucionan se utilizan más frecuentemente para revisar mails, gestionar información financiera de la empresa y hasta realizar transacciones bancarias. El aumento de popularidad sumado a la complejidad de los sistemas operativos móviles y a la información que se puede obtener al atacar estos dispositivos, genera un fuerte incentivo para los desarrolladores de malware. Android, en particular, con una cuota de mercado del 22 %, es el sistema que presenta mayor cantidad de aplicaciones maliciosas, con un aumento exponencial durante el 2011.

El objetivo de este proyecto consiste en diseñar e implementar un sistema de análisis estático de aplicaciones para Android. Este sistema debe ser capaz de analizar grandes cantidades de muestras en poco tiempo y generar un ranking con las aplicaciones más sospechosas y peligrosas. Este ranking será utilizado por los investigadores para elegir las muestras más riesgosas y realizar un análisis manual más profundo en busca de malware.

El trabajo fue realizado completamente en las oficinas de McAfee Labs Chile junto al equipo de Mobile Security. En primer lugar se revisaron las técnicas y herramientas para analizar aplicaciones de Android. Luego se detallaron los requisitos del sistema y se eligieron los lenguajes y las tecnologías de uso. A continuación se diseñó, implementó y testeó el sistema de análisis para finalmente implementar un sistema de calificación junto con los mecanismos necesarios para generar el ranking.

Del estudio de las capacidades del malware actual se concluyó que, si bien aún no presentan la misma complejidad que en el mundo de PC, las técnicas de ataque utilizadas han experimentado una significativa evolución. Existen pocos exploits conocidos para Android y la mayor parte del malware usa ingeniería social para entrar a los dispositivos. Por otro lado, aunque se han dado casos de malware en Google Play, la mayoría se ha encontrado en markets alternativos y páginas web maliciosas.

Finalmente, Samsaraa, el sistema desarrollado, tiene la capacidad para analizar y calificar grandes cantidades de aplicaciones. El resultado es un informe detallado de sus características y del riesgo potencial que presenta. Esta información es utilizada para generar el ranking para los investigadores. Además, posee la facilidad para generar estadísticas, comparar aplicaciones y grupos de muestras; entregar una base sólida para realizar estudios de minería de datos y profundizar el estudio del malware. El diseño del sistema es modular y flexible para facilitar su escalabilidad y utilidad en un entorno de producción, analizando miles de aplicaciones diarias.

Dedicado al tiempo,

que aprovecho al máximo.

Agradecimientos

A McAfee por permitirme realizar mi memoria con ellos. En particular al Team Mobile Security que me acogió durante mi estadía. A Carlos, Fernando y Pedro por su conocimiento en seguridad móvil, comentarios sobre el trabajo y la ayuda que me brindaron para que este funcione. A Dorian por las coordinaciones administrativas en la oficina y a Rohit e Ishikawa-san por sus comentarios sobre Samsaraa.

A mi profesor guía Alejandro Hevia quien me orientó, aconsejó y ayudó en la realización de este trabajo. A los profesores José Miguel Piquer y Jorge Pérez por sus acertadas recomendaciones para mejorar la calidad de esta memoria.

A Sandra y Angélica por su paciencia con los tramites de la U.

A mis papas Cecilia y Felipe por apoyarme durante el trayecto, por motivarme a seguir adelante y por la paciencia para leer, corregir y comentar mi memoria.

A mi polola Natalia por el cariño y la motivación que me entrega día a día y a mis amigos del Team Sky por apoyarme y acompañarme durante todos estos años.

Finalmente quiero agradecer a Nexuiz, Trablas, Pedratan, Cypher, Ferruiz, Daniela, Jordude, Ricooldator, Makavelli y SkyRyu por los ratos de distracción y motivación por que sin ellos mi tiempo en McAfee no hubiera sido lo mismo.

Muchas Gracias!

Nicolás Malbrán Carnevali

Santiago, Noviembre 2012

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

Actualmente, los dispositivos móviles tales como celulares “smart-phones” y tablets, están siendo cada vez más utilizados por los usuarios. De acuerdo a NetMarketShare [30], casi un 9 % de los dispositivos que navegan hoy por internet son móviles. Sus usos van desde una simple llamada por teléfono hasta el uso de herramientas de gestión para controlar diversos factores de una empresa, leer y enviar correos electrónicos, leer las noticias y realizar transacciones bancarias. Para poder lograr todas estas funciones es que estos dispositivos usan sistemas operativos cada vez más grandes y complejos.

La combinación de estos tres factores: un aumento en la cantidad de usuarios, dispositivos que utilizan y guardan información crítica (números de cuentas de banco para transacciones, información relevante para programas de gestión de empresas, redes de contactos, etc) y la evolución de los sistemas operativos potencialmente vulnerables, ha generado una numerosa cantidad de virus, malware y códigos maliciosos en circulación para estos nuevos medios. Esto se debe a que los usuarios maliciosos descubren cada vez más beneficios al atacar un dispositivo móvil. Un informe de Lookout Mobile Security [28] muestra que en enero del 2011 había alrededor de 70 aplicaciones Android infectadas por malware, mientras que en julio del mismo año, ese numero subió a casi 400 aplicaciones. Además, casi un 30 % de los usuarios tienen grandes probabilidades de llegar a un link malicioso durante el 2011, aumentando las amenazas a dispositivos móviles independientes del sistema operativo.

En septiembre del 2012 la repartición de sistemas operativos móviles en el mundo fue la siguiente: según NetMarketShare [31], un 63 % utiliza iOS iPhone/iPad/iPod, un 22 % Android, un 9 % JAVA ME, un 2 % BlackBerry y el restante 4 % se divide entre diversos sistemas menos utilizados.

Entre los dos sistemas más empleados, la mayor cantidad de malware se ha detectado para Android. Esto se debe a que Apple usa una política mucho más estricta para publicar sus aplicaciones y no provee acceso al código fuente directamente, dificultando el desarrollo de malware especifico para iOS. [18]

1.1. Motivación

La gran cantidad de usuarios y la importancia de los dispositivos móviles para éstos hace que esta área sea interesante para investigar y estudiar. La seguridad y privacidad de la información es fundamental para aplicaciones críticas como transferencias bancarias o aplicaciones para la gestión de empresas. Si un malware o spyware logra conseguir información confidencial de una empresa, esta podría sufrir graves pérdidas. Por lo tanto, generar soluciones para aumentar la seguridad, confiabilidad y la confidencialidad tienen un gran valor para los usuarios y el medio, e incentiva a los desarrolladores a crear más aplicaciones productivas.

Por otro lado, dado que un malware puede explotar alguna vulnerabilidad del sistema operativo o de una aplicación, suele tener mucho ingenio involucrado. Además que un malware suele "pasar inadvertido" y es un desafío averiguar cómo funciona y qué es lo que realiza mediante su estudio e ingeniería reversa.

Finalmente, dado que la aparición de malware es relativamente reciente, no existen tantas herramientas para acelerar su proceso de análisis. La cantidad de malware aumenta rápidamente día a día y hace necesaria la presencia de una herramienta que pueda analizar automáticamente grandes cantidades de aplicaciones en busca de malware.

1.2. Objetivos

1.2.1. Objetivo general

El objetivo general de este trabajo de título es diseñar, implementar y estudiar un sistema de análisis de malware en aplicaciones para Android. Este software permitirá realizar un análisis estático de las aplicaciones, generando un informe detallado de sus características. Además, creará un ranking de las aplicaciones analizadas según su nivel de riesgo. El objetivo de este ranking es guiar a los investigadores hacia los malware más peligrosos para realizar un análisis en profundidad. El caso de uso principal de este sistema es procesar automáticamente grandes cantidades de aplicaciones provenientes de internet y seleccionar las más sospechosas o riesgosas.

1.2.2. Objetivos específicos

Este objetivo general puede detallarse en los siguientes objetivos más específicos:

- Comprender el funcionamiento del sistema operativo Android y sus componentes y estudiar las metodologías de ingeniería reversa utilizadas para el análisis de sus aplicaciones.
- Analizar e investigar las técnicas de ataques empleadas por malware existentes en dispositivos móviles, buscando patrones de acción, objetivos comunes y parentescos entre los diferentes malware que permitan clasificarlos y/o agruparlos para un mejor muestreo y estudio.

- Estudiar la acción de malware sobre estos dispositivos, de forma de evaluar su impacto y los daños que puedan causar al sistema o su usuario.
- Crear un sistema de análisis automático para las aplicaciones, que genere un informe sobre sus características y permita crear un ranking según su riesgo. Esto guiará a los analistas hacia los malware más peligrosos para acelerar su estudio más detallado, y mejorar la prevención.

1.3. Conceptos básicos

Android OS

Android es un sistema operativo diseñado para dispositivos móviles como celulares o tablets. Android Inc. fue fundada en octubre de 2003 en Estados Unidos por Andy Rubin, Rich Miner, Nick Sears y Chris White, empresa que comenzó con el desarrollo de este sistema. En agosto de 2005 fue comprada por Google, empresa que actualmente se encarga de su desarrollo y mantención [35]. Desde entonces regularmente ha liberado nuevas actualizaciones del sistema operativo. La versión más reciente es la 4.1 Jelly Bean. Actualmente Android es utilizado en un 21 % de los dispositivos móviles [31].

Markets, Android Market y Google Play [12]

Un "market" es un sitio web o tienda desde el cual se pueden descargar aplicaciones para Android fácilmente y adaptados para las interfaces de los dispositivos móviles. Google puso en funcionamiento el Android Market en octubre de 2008 como el lugar oficial para descargar aplicaciones. La mayoría de las aplicaciones disponibles para Android se pueden descargar desde él. En marzo de 2012, Google rebautizó el Android Market como Google Play Store o sólo Google Play.

Sample / Muestra

Un "sample" (o muestra en español) es un archivo (generalmente una aplicación) que puede o no contener malware. Cualquier archivo puede ser una muestra y pueden existir varias muestras de una misma aplicación o malware.

Malware

Malware viene del ingles "MALicious softWARE", que tal como su nombre lo dice, son programas maliciosos creados para atacar sistemas (pc, celulares, tablets, servidores). Los objetivos principales de los diseñadores de malware son obtener dinero y/o información. Existen numerosos y complejos tipos de malware que los hacen más o menos peligrosos según su acción sobre el sistema.

Por ejemplo, borrar datos, recolectar información sobre sitios web visitados, tarjetas bancarias, información personal, enviar spam a otros usuarios, instalar otro tipo de malware, etc. [32]

Troyano

Un programa que aparenta tener una funcionalidad legítima pero que además incluye una función oculta, potencialmente maliciosa, que evade los controles de seguridad normales. [20]

Spyware

Un programa instalado secretamente en un sistema computacional con el fin de recolectar información sensible de los individuos u organizaciones sin su conocimiento. [20]

PUP (Potentially Unwanted Program)

Los PUP son programas potencialmente no deseados. Realizan acciones que no son maliciosas per se, pero que podrían no hacer lo que dicen o realizar acciones no deseadas por el usuario. Por ejemplo, una aplicación que envía las coordenadas del GPS por SMS al recibir un SMS con cierto código, puede usarse para encontrar un celular robado pero también podría usarse para seguir a alguien sin que lo sepa. Este tipo de aplicaciones, entre otras, son consideradas como PUPs.

Vulnerabilidad

Una vulnerabilidad es una debilidad en un sistema computacional, un procedimiento de seguridad o una aplicación que podría ser explotada y aprovechada por un atacante o una fuente de amenaza. [20]

Exploit

Aprovechar una vulnerabilidad para atacar un sistema con el fin de causar un comportamiento indeseado o imprevisto.

Zero Day Exploit

Un Zero Day Exploit es un ataque que aprovecha una vulnerabilidad desconocida hasta ese momento. Estos ataques son particularmente peligrosos pues, dado que la vulnerabilidad no es pública, aun no existe la solución para arreglarla, por lo que el autor del software tiene cero días para prepararse y reparar dicha vulnerabilidad. [32]

Sandbox

Entorno controlado que permite ejecutar código malicioso sin que sea un riesgo para el sistema, impidiendo así que el código ejecutado acceda a los recursos del sistema, salvo los explícitamente permitidos. [20]

Capítulo 2

Antecedentes

2.1. Android OS

Android es un conjunto de elementos formados por un sistema operativo, un middleware y las aplicaciones claves, necesarias para un dispositivo móvil. Google provee el Android SDK y Android NDK para desarrollar aplicaciones para esta plataforma.

El SDK incluye las herramientas y la API necesaria para escribir aplicaciones en Java y exportarlas a Android.

El NDK es un complemento al SDK que permite crear porciones de aplicaciones en código nativo(C o C++), para mejorar su eficiencia en casos críticos.

2.1.1. Arquitectura

En la Figura 2.1 se muestran los principales componentes de Android.

Las aplicaciones: son los componentes de más alto nivel y los que interactúan con el usuario directamente. Android viene por defecto con un cliente de correo, servicio de SMS, calendario, navegador, contactos y mapa, entre otras cosas.

Application Framework: Android provee un framework para facilitar el desarrollo de aplicaciones, dando acceso a todas las características del dispositivo y del sistema operativo.

Libraries: son las librerías incluidas en Android y utilizadas por diversos componentes del sistema. También se encuentran disponibles para los desarrolladores a través del framework.

Android Runtime: posee las librerías principales y la máquina virtual Dalvik, donde se ejecutan todas las aplicaciones, cada una en su propia máquina. Está optimizada para dispositivos pequeños y para correr múltiples instancias a la vez.

Linux Kernel: en su núcleo, Android se basa en el kernel de linux para todas las necesidades de bajo nivel como seguridad, manejo de memoria y procesos, acceso a la red y los drivers para el hardware.

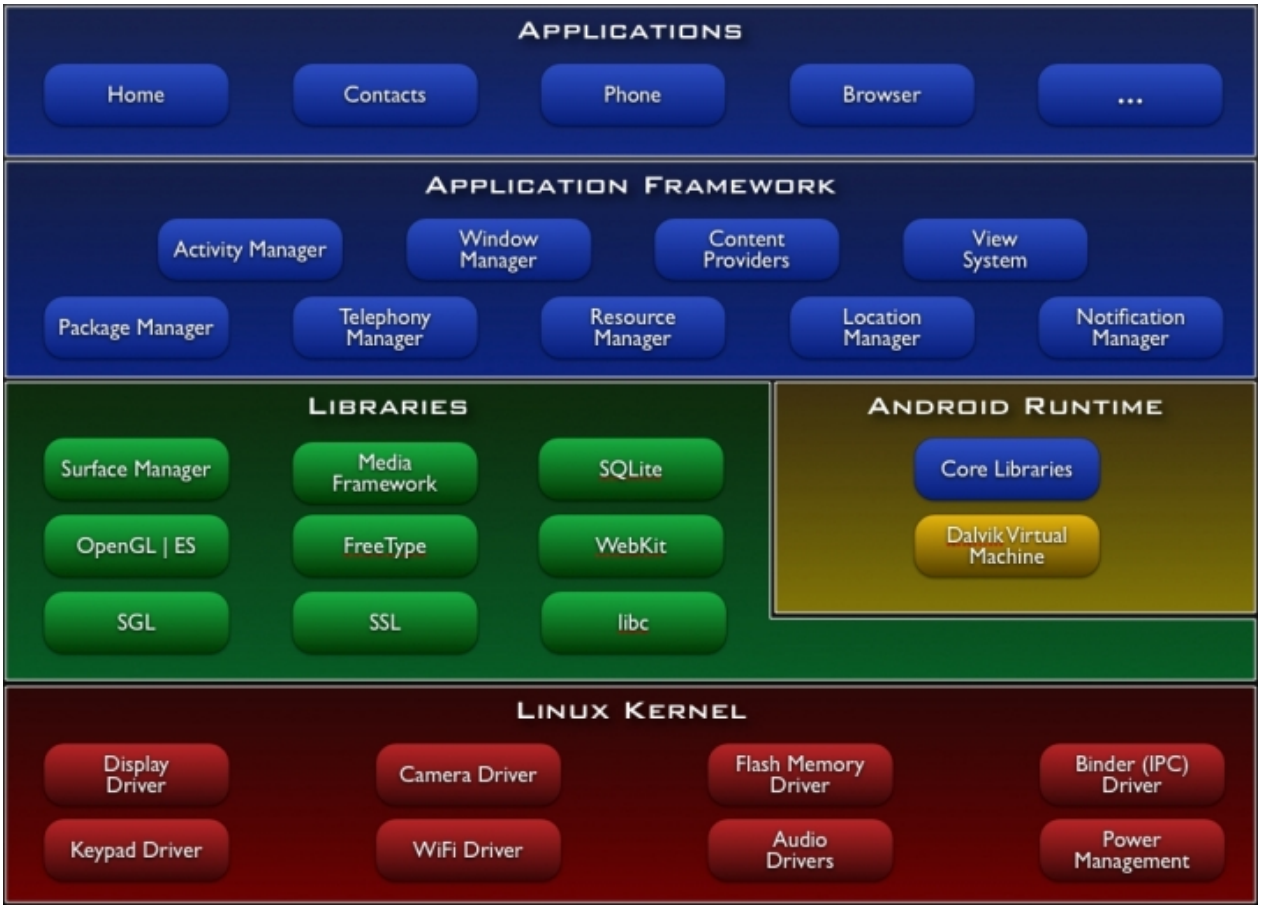


Figura 2.1: Componentes del sistema operativo Android.

2.1.2. Componentes de las aplicaciones

Las aplicaciones son escritas en Java y compiladas al formato de la máquina virtual Dalvik (*dex*). El conjunto del código compilado y los recursos necesarios (como imágenes, vídeos, sonidos, binarios, etc) son empacados y firmados en un *Android Package* (*apk*). Este archivo (el *apk*), se considera una aplicación y es suficiente para ser instalado en un dispositivo con Android.

Las aplicaciones están separadas en distintos componentes:

Activities: una actividad (*Activity*) corresponde a una sóla pantalla con interfaz de usuario. Cada actividad está diseñada para hacer una sóla acción (escribir un correo, editar una imagen, ver la lista de fotos). Una aplicación suele estar compuesta de varias actividades independientes entre sí. Cualquiera puede iniciar otra mientras el programa lo permita.

Services: un servicio corresponde a un proceso que se ejecuta en segundo plano, como reproducir una canción o descargar un archivo. No tiene interfaz gráfica y suele ser usado para operaciones que toman mucho tiempo para no interrumpir al usuario. Generalmente son iniciados por una actividad.

Content providers: un *content provider* es una interfaz común para acceder al contenido de la aplicación y para que otras aplicaciones puedan acceder a este contenido. Por ejemplo, Android provee un *content provider* para acceder a los contactos del usuario.

Broadcast receivers: es un componente que responde a eventos generados por el sistema u otras aplicaciones. Por ejemplo, si queda poca batería, si el sistema terminó de iniciarse o de descargarse un archivo.

Intents: corresponden a la "intención de hacer algo". Son los mensajes de Android para iniciar las *Activities*, *Services* y *Broadcast receivers* (Interprocess Communication). Por ejemplo, un *Intent* con el mensaje VIEW, destinado a un *Broadcast receiver*, puede iniciar una *Activity* en un navegador.

2.1.3. Permisos

Uno de los niveles de seguridad de Android consiste en un sistema de permisos. Ciertos recursos del sistema necesitan permisos para poder usarse, por ejemplo, acceder a internet, enviar/recibir SMS, acceder a los contactos o a la SD card. Estos permisos deben ser declarados en el Android Manifest y son validados por el usuario al momento de instalar la aplicación. En caso de que el usuario no quiera darle estos permisos, la aplicación no podrá instalarse. Por otro lado, aunque una aplicación tenga código para realizar una acción, si no tiene los permisos necesarios, ésta recibirá una *Excepcion* al tratar de ejecutarse.

2.1.4. AndroidManifest.xml

Este archivo es esencial en las aplicaciones de Android. Es un archivo XML que especifica todos los componentes de la aplicación, en particular:

- Las *Activities*, *Services*, *Content providers* y *Broadcast receivers*,
- Los permisos necesarios para ejecutarse,
- La versión mínima de la API requerida,
- Los componentes de hardware que usará, como la cámara, bluetooth o el giroscopio (Features),
- Las librerías necesarias aparte del framework, como la librería de Google maps.

2.1.5. Documentación

Dado que Android es Open Source, todo el código fuente está disponible, al igual que la documentación para el SDK, NDK y los códigos de Dalvik:

Código fuente: `http://source.android.com/source/`

Guía Android SDK: `http://developer.android.com/guide/`

API Android SDK: `http://developer.android.com/reference/packages.html`

Dalvik VM:
- `http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html`
- `http://www.milk.com/kodebase/dalvik-docs-mirror/docs/dalvik-bytecode.html`
- `http://www.milk.com/kodebase/dalvik-docs-mirror/`

2.2. Trabajos relacionados

2.2.1. Android Malware Past, Present, and Future [5]

Carlos Castillo es uno de los miembros del Mobile Security Working Group de McAfee. Este equipo se dedica actualmente a analizar aplicaciones sospechosas para Android con el fin de identificar malware a tiempo. Las aplicaciones son luego clasificadas como malware, PUP o limpias para ser reconocidas por el antivirus de Android.

En noviembre del año 2011, Castillo publicó un white paper [5] sobre el análisis de malware para Android. En él explica cómo analizan regularmente el malware que reciben en McAfee, explica el funcionamiento de Android y detalla las distintas herramientas y metodologías utilizadas. Además, muestra dos ejemplos de malware: uno correspondiente a una aplicación antigua y el otro sobre una aplicación actual y termina finalmente con un pronóstico sobre las características que se esperan para el próximo malware.

Como ejemplo del pasado, Castillo explica el funcionamiento de *Fake Player*. Éste fue el primer trojano descubierto de forma activa ("in the wild") en agosto de 2010. Este malware se asemeja a un reproductor multimedia pero a la vez envía mensajes de texto a servicios premium sin la autorización del usuario. De esta forma, los autores de *Fake Player* obtienen dinero gracias a los mensajes de texto enviados.

Como ejemplo del presente, Castillo explica el funcionamiento de *Plankton*, un Spyware (malware cuyo objetivo es robar información) descubierto por la Universidad de Carolina del Norte. En este caso, el malware aparece como la aplicación "Angry Bird Rio Unlocker". Sin embargo, esta aplicación crea un servicio que escucha comandos remotos que sirven en gran medida para robar información (Numero de teléfono, favoritos, historial del navegador) y enviarla a un servidor remoto. Además, posee la capacidad para descargar nuevas funcionalidades desde internet de forma dinámica, siendo el primero en realizar ésto.

Finalmente, el autor pronostica la aparición de nuevos malware con la capacidad de replicarse a si mismo (una vez infectado un dispositivo, buscar otro vulnerable e infectarlo), malware que tengan rootkit (capacidad para modificar el sistema operativo para permanecer por más tiempo y dificultar su eliminación) y, por último, malware polimorfos o metamórficos (capacidad para cambiar su propio código y dificultar la detección por antivirus).

2.2.2. Android Reverse Engineering Tools [1]

Axelle Apvrille (aka Crypto Girl) es una analista e investigadora especializada en el área de seguridad móvil. Actualmente trabaja para Fortinet[1]. En marzo de 2012 dio una conferencia en Insomni'hack[2]. En esta presentación [1], Apvrille explica cómo hacer ingeniería reversa para las aplicaciones de Android. Analiza su contenido y presenta un ejemplo de ingeniería reversa sobre el malware *Android/Spitmo.C!tr.spy*. Muestra las herramientas que usa para tal efecto y explica para que sirve cada una de ellas. También muestra otras herramientas existentes, con sus ventajas y desventajas, y algunos trucos adicionales que utiliza. Gran parte de las herramientas de esta presentación son explicadas en la página 20.

2.2.3. Reducing the window of opportunity for Android malware [2]

En mayo de 2012, Axelle Apvrille y Tim Strazzere publicaron un trabajo [2] con objetivos similares a Samsaraa. Diseñaron un sistema compuesto de dos grandes partes: un recolector de aplicaciones (crawler) y una heurística para analizar y calificar las muestras según su riesgo potencial.

El crawler consiste en un robot que se conecta al Android Market [12] y descarga la mayor cantidad y variedad de aplicaciones posibles. El documento explica las características que debe tener para descargar aplicaciones de distintos dispositivos y para diferentes versiones de Android. También se muestra cómo usarlo para que no sea detectado y vetado por Google, las cuentas de usuario necesarias para lograrlo y como agregar otros markets al sistema.

Por otro lado, la heurística desarrollada analiza una cantidad de muestras recolectadas, detecta las más sospechosas y les asigna un puntaje para luego hacer un ranking de las más peligrosas.

2.2.4. Dissecting Android Malware: Characterization and Evolution [38]

Yajin Zhou y Xuxian Jiang publicaron "Dissecting Android Malware: Characterization and Evolution" [38] en mayo de 2012. Durante el desarrollo, recolectaron 1200 muestras de malware para Android con el fin de analizarlas, compararlas y clasificarlas para tener un mejor conocimiento del malware actual y observar como ha ido evolucionando desde su aparición. Este trabajo se enmarca en el proyecto *Android Malware Genome Project* [37] al alero del cual también liberaron, bajo ciertas condiciones, la base de datos de muestras.

En este artículo [38], se muestran las diferentes familias de las muestras y su evolución desde el primer malware. Se caracteriza el tipo de instalación de cada una, las técnicas de activación, y el efecto del ataque o el objetivo de cada muestra. Luego se presenta una pequeña comparación con aplicaciones limpias. Y finalmente, se detalla la evolución de dos familias: *DroidKungFu* y *AnserverBot*, mostrando las diferentes técnicas y ataques entre cada una de las nuevas versiones, con énfasis en las estrategias utilizadas para adaptarse y evitar la detección de los antivirus.

[1] **Fortinet:** `http://www.fortiguard.com/`

[2] **Insomni'hack:** `http://www.scrt.ch/en/insomnihack/2012/presentation`

2.2.5. The Honeynet Project [26]

The Honeynet Project [26] es una organización internacional de investigación en seguridad. Fue fundada en 1999, sin fines de lucro y constituida por voluntarios. Se dedica a investigar ataques y problemas de seguridad computacional en general, a educar a la comunidad sobre los peligros existentes y a realizar herramientas de seguridad ya sea para investigación o para prevención. Para lograr esto, despliegan honeynets[3] para capturar ataques activos, analizarlos y finalmente publicar sus descubrimientos. Gracias a esto, ayudan a la comunidad a entender mejor los ataques y a como defenderse.

Además, The Honeynet Project organiza diversos eventos y competencias. Por ejemplo, "desafíos" en donde los participantes tienen que analizar ataques y luego publicar sus descubrimientos. De esta forma los concursantes aprenden a analizar y ver las metodologías utilizadas por otros participantes. También han propuesto proyectos a los "Google Summer of code(GSoc)" concretados por los participantes y han realizado conferencias de seguridad donde exponen presentaciones e invitan a los asistentes a realizar ejercicios prácticos.

Por otro lado, The Honeynet Project también ha publicado dos series de papers. La primera, llamada "Know Your Enemy", con los resultados de las investigaciones y análisis realizados, explicando los distintos ataques y sus soluciones. Y la segunda, llamada "Know Your Tools", donde analizan en profundidad algunas herramientas para el análisis de ataques.[4]

The Honeynet Project también ha creado diversas herramientas de seguridad. De estas, las relevantes para este estudio son:

APKInspector: desarrollado por Cong Zhen en el GSoc 2011. Es una herramienta para análisis estático de las aplicaciones Android. Permite hacer ingeniería reversa, analizar y visualizar los paquetes Android. Sus características son explicadas con más detalle en la página 21.

Droidbox [17]: desarrollado por Patrick Lantz en el GSoc 2011. Es una plataforma para análisis dinámico de las aplicaciones de Android. Permite ejecutarlas en un sandbox y registrar su comportamiento, como las lecturas y escrituras de archivos, conexiones de red, servicios iniciados, SMS enviados, acciones criptográficas y llamadas realizadas.

Android Reverse Engineering virtual machine(ARE) [25]: es una máquina virtual para VirtualBox[5] que viene precargada con diversos programas para analizar aplicaciones de Android y hacer ingeniería reversa sobre ellas. Entre estos programas se encuentran: Androguard, Android SDK, APKInspector, Apktools, dex2jar y Droidbox. Éstos se detallan mas adelante.

[3] Redes designadas para ser atractivas a los atacantes y así analizar sus ataques. [20]

[4] **Lista de papers publicados:** `http://www.honeynet.org/papers`

[5] `https://www.virtualbox.org/`

2.2.6. Un repositorio de muestras: contagio mobile [24]

El blog "contagio mobile" [24] es mantenido por Mila Parkour[6] investigadora de DeepEnd Research[7]. Funciona como una base de datos de malware para dispositivos móviles. El objetivo es proveer muestras de malware para su investigación. Cualquier persona puede enviar una muestra para que sea almacenada. Cada muestra publicada contiene información sobre el archivo de la muestra (nombre del archivo, hash, tamaño, fecha de análisis, etc) y un vínculo al informe de investigación realizado.

Además de éste, Mila Parkour mantiene otros dos blogs: "contagio exchange"[8] y "contagio"[9]. El primero es similar a "contagio mobile" pero para malware de todo tipo (salvo mobile) y el segundo es un blog personal de Mila donde publica análisis de malware realizados por ella.

2.2.7. Un antivirus online: virustotal.com [33]

El sitio web virustotal.com [33] es un antivirus gratuito que analiza archivos y URL en busca de virus, malware, troyanos y otro tipo de contenido malicioso. Al subir un archivo, éste será analizado utilizando diferentes antivirus de escritorio y en linea generando un reporte global sobre el archivo. De igual forma, al enviar una URL esta será escaneada para verificar los resultados obtenidos por distintos antivirus.

Además, este sitio posee estadísticas sobre los archivos analizados en la última semana y una comunidad en la cual los participantes pueden comentar los últimos análisis y hacer reviews sobre los archivos y/o URL escaneados.

2.2.8. Marvin [Mobile Security] [29]

Marvin Safe es una aplicación para Android y un servicio para empresas [29]. Provee la capacidad para subir una aplicación a la nube, analizarla y recibir un reporte sobre sus características. Actualmente, dado que es un proyecto privado, no existe mucha información al respecto. Del sitio web se puede deducir que utiliza técnicas de inteligencia artificial y realiza un análisis estático y de comportamiento. El informe entregado muestra la comunicación realizada a través de internet, las características de la aplicación y un reporte de alarmas sospechosas.

2.2.9. Andrubis [16]

Andrubis [16] [4] es parte del proyecto *Anubis: Analyzing Unknown Binaries* desarrollado por *International Secure Systems Lab* (iseclab). Consiste en un servicio online que permite subir aplicaciones y recibir un informe del análisis realizado. El análisis es estático y dinámico, entrega

[6] **Perfil en Linked In:** `http://www.linkedin.com/in/milaparkour`
[7] `http://www.deependresearch.org/`
[8] `http://contagioexchange.blogspot.com/`
[9] `http://contagiodump.blogspot.com/`

un reporte de las características de la aplicación (como Activities y permisos requeridos) y una lista de las acciones que realiza (como conectarse a internet, escribir archivos o iniciar servicios).

2.2.10. La capa de seguridad de Google: Bouncer [9]

En febrero de 2012, Hiroshi Lockheimer, Vice-Presidente de Ingeniería de Android, publicó un artículo en uno de los blogs de Google[10] presentando *Bouncer* [9], una nueva capa de seguridad para el Android Market [12]. Ésta revisa y analiza las diferentes aplicaciones que están en el Android Market y que llegan a él.

Según el post, este nuevo servicio hace un análisis estático y dinámico de las aplicaciones, las compara con malware conocidos y busca comportamientos o permisos sospechosos. Además, revisa los desarrolladores para evitar que suban más aplicaciones en caso de que ya lleven muchas faltas. A parte de esta publicación en el Blog, no hay más registros oficiales de este servicio por parte de Google.

2.2.11. Mobile Sandbox [19]

Mobile Sandbox [19] es una herramienta en línea para el análisis estático y dinámico de las aplicaciones para Android. Es parte del proyecto MobWorm [14] del grupo de trabajo "Embedded Malware" de la universidad Ruhr-Universität Bochum[11] en Alemania, en el cual analizan tanto ataques como defensas para smartphones.

Actualmente, el servicio está en continuo desarrollo y publicado sólo como herramienta de investigación. Permite subir una aplicación para Android (apk) y generar un informe de su análisis. El análisis dinámico no está disponible aun, sin embargo el resultado del análisis estático permite obtener diferentes características de la aplicación como los permisos requeridos, los servicios presentes, los *Intents* utilizados, las llamadas potencialmente peligrosas y las URL presentes, entre otros detalles. Además, entrega los resultados entregados por diversos antivirus al escanear la aplicación.

Por otro lado, Michael Spreitzenbarth, un estudiante de doctorado de la universidad de Erlangen-Nuremberg[12] [13], mantiene un blog asociado[14] a este sistema. En él se muestra en detalle los análisis realizados a distintos malware para dispositivos móviles.

[10] `http://googlemobile.blogspot.com/`
[11] `http://www.ruhr-uni-bochum.de/`
[12] **Universidad de Erlangen-Nuremberg:** `http://www.uni-erlangen.org/`
[13] **Departamento de computación:** `http://www1.informatik.uni-erlangen.de/`
[14] **Blog:** `http://forensics.spreitzenbarth.de/`

Capítulo 3

Procedimientos

3.1. Metodología

Esta memoria fue realizada junto al equipo *Mobile Security* de McAfee Labs Chile y en sus dependencias. La metodología utilizada consta de varias etapas, la cuales son explicadas a continuación.

1. La primera fase consistió en investigar y aprender las técnicas utilizadas para analizar aplicaciones de Android. Se revisaron diversas técnicas de ingeniería reversa, descompiladores y desensambladores. Se probaron las distintas metodologías y herramientas en muestras de malware, para así, evaluar su desempeño y la información que provee cada una. Además, esta fase involucró aprender cómo funciona el SDK, el sistema operativo Android y todos sus componentes.

   Todo esto permitió conocer el entorno en que se ejecutan las aplicaciones y las herramientas disponibles para analizarlas. Cabe destacar que el objetivo es analizar aplicaciones de las cuales no hay acceso a su código fuente, únicamente el paquete (apk) tal como es distribuido.

2. En la segunda fase se definieron los requisitos que debía tener el sistema y cuáles serían los casos de uso principales.

3. Luego se diseñó la arquitectura, los módulos necesarios para el sistema y todo lo necesario para empezar a implementarlo. Junto con ésto se definieron las herramientas, lenguajes y tecnologías que se utilizarían para desarrollar el software.

4. La cuarta fase, una de las más largas, consistió en implementar el sistema. Éste fue desarrollado en pequeños módulos, testeando cada uno de ellos y certificando que funcionaran bien antes de proceder al siguiente. Para testear que los módulos realicen lo esperado, se utilizaron muestras de malware de Contagio Mobile [24], algunas otorgadas por McAfee y aplicaciones descargadas del Android Market [12].

5. Cuando el sistema estuvo en un estado estable, con los requisitos principales funcionando bien y con una interfaz más usable, se comenzaron las pruebas de uso con una mayor cantidad de aplicaciones. Se utilizó un set de 13.000 aplicaciones provistas por AV-Test [3] para testear el sistema. Ésto permitió probar una mayor cantidad de aplicaciones reales, estudiar casos

particulares, corregir errores puntuales y someter el sistema a un análisis más exhaustivo. También, se usó para recolectar estadísticas sobre los distintos tipos de aplicaciones y el funcionamiento del sistema desarrollado.

6. A continuación, se utilizó el conocimiento del equipo de *Mobile Security* de McAfee, lo aprendido durante la investigación preliminar y las estadísticas recolectadas para definir los indicadores. Estos indicadores (llamados *flags*) son las características que permitirán distinguir una aplicación limpia de un malware. El detalle de su implementación y funcionamiento está explicado mas adelante.

7. La última fase consistió en testear los *flags*. Es decir, usar el sistema para analizar varias muestras de aplicaciones limpias y malware y verificar que pueda puntuarlas correctamente. De esta forma, ver si puede diferenciar las aplicaciones benignas (menor puntaje) y el malware (mayor puntaje).

3.2. Equipos y sofwares relevantes

Durante el desarrollo del software se utilizó un computador prestado por McAfee, con un procesador Intel Core i7-2600 (8 núcleos de 3.40 GHz, 64-bit) y 16GB de RAM con sistema operativo Windows 7 SP1. Dado el entorno donde funcionará el sistema finalmente, gran parte del trabajo fue hecho en una máquina virtual de VMware con Ubuntu 11.10.

Para el desarrollo mismo se empleó Sublime Text 2, un editor de texto enriquecido. El lenguaje de programación principal fue Python con algunos módulos como Androguard y Django y virtualenv para administrar el entorno de Python. También se utilizó HTML, CSS y Javascript para la interfaz web, MySQL para la base de datos, Git para el control de versiones y Truecrypt para el almacenamiento seguro de los datos.

Python es un lenguaje de programación open source, de alto nivel y de propósito general. Es un lenguaje interpretado y soporta múltiples paradigmas de programación como Orientación a Objetos, Imperativo y funcional. Se usaron varios módulos, en particular Androguard, explicado más adelante y Django[1]. Django es un framework MVC (Modelo Vista Controlador) para desarrollo web, que provee muchas herramientas para facilitar el desarrollo, como por ejemplo un ORM (Object-Relational Mapper) para manejar los modelos y la base de datos. Además, se contó con virtualenv[2]. virtualenv, que como su nombre lo sugiere, es un entorno virtual para administrar los módulos de Python. Permite crear un entorno de Python independiente de la instalación disponible en el sistema operativo, lo que facilita la instalación y administración de los módulos necesarios para desarrollar el software.

Para el control de versiones se usó Git[3]. Git es un sistema distribuido de control de versiones, gratis y de código abierto. A diferencia de SVN, permite crear y juntar ramas (branching and merging) mucho más rápido y cómodamente.

[1] **Django:** `https://www.djangoproject.com/`
[2] **virtualenv:** `http://www.virtualenv.org`
[3] **Git:** `http://git-scm.com/`

Finalmente se utilizó Truecrypt[4] para guardar el proyecto a disco. Truecrypt es una herramienta que permite encriptar y desencriptar discos y archivos. Por requerimientos de McAfee se utilizó un disco encriptado para almacenar todo el proyecto y que estuviera seguro en caso de pérdida.

[4] **Truecrypt:** `http://www.truecrypt.org/`

Capítulo 4

Desarrollo y Resultados

4.1. Investigación sobre el análisis de aplicaciones para Android

Existen principalmente dos tipos de análisis, complementarios entre sí, para malware de cualquier tipo: análisis dinámico y estático. Estos se detallan a continuación.

4.1.1. Análisis dinámico

El análisis dinámico consiste en ejecutar el malware en un entorno controlado (sandbox) permitiendo monitorear los cambios que hace en el sistema y los recursos a los que accede. Usualmente se utilizan máquinas virtuales, que emulan el sistema operativo y permiten la modificación de los métodos críticos del sistema, de forma que cuando un malware llame a este método, la máquina virtual sepa que fue llamado.

La ventaja de este tipo de análisis es que se puede automatizar a gran escala y permite tener una visión general de los cambios realizados al sistema operativo y de los recursos a los cuáles accede. Sin embargo, una desventaja es que no se puede saber bien cuál es el objetivo del código ejecutado. En efecto, este puede depender de muchas variables y algunas condiciones pueden no cumplirse en todas las instancias, por lo que al ejecutarlo en un escenario puede ser inofensivo, y en otro ser muy dañino. Además, algunos malware han aprendido a diferenciar cuándo están siendo ejecutados en un sistema normal o en una máquina virtual, por lo que están programados para no hacer nada en un sandbox. De esta manera el análisis se vuelve inútil.

En el caso de Android, existe una herramienta llamada DroidBox [17] que permite realizar análisis dinámico, la cual está explicada en la página 11.

4.1.2. Análisis estático

4.1.2.1. Descripción y objetivo

El análisis estático analiza el código fuente de la aplicación, o el binario en su defecto, con el fin de averiguar qué es lo que hace el programa. Ésto permite saber con mejor precisión qué es lo que realiza y cómo. Sin embargo pocas veces se tiene a mano el código fuente del malware. En estos casos, la ingeniería reversa lo puede extraer gracias a distintas técnicas y herramientas.

Como fue mencionado anteriormente, una ventaja frente al análisis dinámico es que se puede saber exactamente qué es lo que realiza el malware en cuestión, pudiendo determinar con exactitud si es malicioso o no y sus consecuencias.

A pesar de sus beneficios, tiene numerosas complicaciones las cuáles se detallan a continuación:

- De partida, la mayoría de las veces no se dispone del código fuente y lo único a mano es el binario ya compilado.
- Existen descompiladores y desensambladores que permiten obtener el código fuente a partir del binario, ya sea en lenguaje ensamblador (jasmin o smali en el caso de Android) o en lenguaje de alto nivel, como Java. Sin embargo, el lenguaje ensamblador es muy poco intuitivo y requiere gran conocimiento para analizar su código. Y el código obtenido en java no siempre es preciso y puede contener fallas o simplemente no lograr descompilar fragmentos de código, haciendo impreciso su análisis.
- Otra desventaja es la complejidad de los programas. Un programa suficientemente complejo tiene numerosas clases y se vuelve muy costoso revisarlo a mano.
- Además, algunos programadores, con el fin de esconder su código o dificultar la ingeniería reversa, utilizan ofuscadores para enredar y ocultar el código. En estos casos, todos los nombres de variables y de clases son reemplazados por letras (a, b, ab, ad, ed ...) y en ciertos casos el flujo del programa es distorsionado, complicando aun más su lectura.

A pesar de todo esto, existen algunas características de las aplicaciones para Android que simplifican su análisis estático. En particular, el archivo AndroidManifest.xml contiene muchas características de la aplicación, tiene un formato establecido y por lo mismo, no puede ser ofuscado. Por lo tanto, es factible obtener el código de este archivo y analizarlo automáticamente sin mayores complicaciones, lo que entrega una visión general de la aplicación y una específica de los componentes y permisos necesarios para que funcione. Además, hasta ahora, las aplicaciones para Android no son extremadamente grandes, por lo que analizar sus clases a mano no resulta tan costoso. Y finalmente, dado que el SDK de Android está bien documentado y especificado, se pueden numerar las llamadas críticas o peligrosas al sistema y buscarlas en el código automáticamente, aunque esté ofuscado, o buscarlas en el lenguaje de ensamblador.

4.1.2.2. Análisis de un apk

Para analizar una aplicación de Android hay que cumplir varios pasos, éstos se explican a continuación.

Desempacar el apk El formato de archivo Android Package es el mismo que el formato ZIP. El apk puede ser extraído usando cualquier aplicación que lo soporte, como 7zip[1]. Al extraer un apk se encuentra lo siguiente:

- Una carpeta *res*, que contiene varios archivos binarios de xml, que representan las vistas, los strings, y el resto de las configuraciones.
- Una carpeta *assets* donde están las imágenes, sonidos y otros archivos multimedia utilizados por la aplicación.
- Una carpeta *META-INF* con los datos de la firma de la aplicación.
- El archivo *AndroidManifest.xml* que es el mismo archivo explicado en la página 8 pero compilado en su forma binaria.
- El archivo *classes.dex*, que contiene todas las clases de la aplicación, compiladas al bytecode de Dalvik.
- El archivo *resources.arsc* en binario, que corresponde al mapeo entre ids utilizados en la aplicación y los recursos (como strings, imágenes o sonidos).

Leer los xml y resources.arsc Todos los archivos xml dentro de un apk están en binario, por lo que es necesario descompilarlos a xml clásico. Existen varias herramientas para lograr ésto, las cuales se detallan en la página 20. En general, no hay mayores problemas y los archivos se pueden leer sin dificultad. En particular, el archivo AndroidManifest.xml es importante pues contiene la declaración de todos los componentes de la aplicación.

De forma similar, el archivo resources.arsc puede ser transformado a xml clásico con el apktool, descrito en la página 21.

Desensamblar el archivo classes.dex Este archivo corresponde a todas las clases de la aplicación, compiladas en bytecode de Dalvik. Tiene vital importancia pues representa toda la funcionalidad de la aplicación.

Para su análisis existen dos principales métodos:

1. Desensamblarlo a lenguaje de ensamblador (como smali [13]) y luego analizarlo. La ventaja de este método es que el resultado representa tal cual el código compilado y la desventaja es que el lenguaje de ensamblador es mucho más complejo de entender y analizar a mano. Sin embargo, es ideal para una herramienta automatizada.

[1] `http://www.7-zip.org/`

2. Usar la herramienta dex2jar que transforma el dex en un archivo jar, que puede ser descompilado a Java usando diversas herramientas. La ventaja es que se obtiene el código fuente, que es más simple de entender que el lenguaje de ensamblador. Sin embargo, el proceso de descompilación no es perfecto y varias veces se obtiene código con fallas, o simplemente no se logra descompilar entero.

Luego se pueden analizar las clases y el flujo de la aplicación para deducir qué es lo que realiza.

4.1.3. Herramientas

Para realizar los pasos descritos anteriormente existen diversas herramientas, disponibles actualmente, las cuales de detallan a continuación.

4.1.3.1. Herramientas generales

Herramientas del SDK [10]: El Android SDK incluye varias herramientas para desarrollar, corregir y publicar aplicaciones Android. Entre ellas algunas son útiles para realizar ingeniería reversa sobre los apk, las cuales se detallan a continuación:

- **Android Emulator:** el SDK permite crear Android Virtual Devices (ADV's) para emular dispositivos con Android y ejecutar las aplicaciones sin la necesidad de un dispositivo físico. Utilizado con DroidBox puede ejecutar las aplicaciones y hacer el análisis dinámico. Este emulador tiene la opción *tcpdump*, la cual registra toda la actividad de red generada por el emulador y la guarda en un archivo para posterior análisis.
- **dexdump:** permite desensamblar los archivos dex. Su resultado no es muy práctico. Es difícil de leer dado que todas las clases quedan en un solo archivo y no siempre funciona bien. Sin embargo, sirve para verificar si un archivo dex esta corrupto (malformado) o no.
- **Android Debug Bridge(adb):** es una interfaz que se conecta a un dispositivo (ya sea real o emulador) y lo controla desde la línea de comandos.
- **logcat:** es un comando disponible desde adb que permite recolectar y ver los registros de varias aplicaciones y partes del sistema.
- **Dalvik Debug Monitor Server(ddms):** controla y monitorea la ejecución de las aplicaciones.
- **Android Asset Packaging Tool(aapt):** herramienta utilizada en el proceso de creación de los apk. En particular, tiene la opción para leer los archivos xml en binario. Sin embargo el resultado no es muy bueno.

android-apktool [34]: Conjunto de herramientas desarrolladas por Ryszard Wiśniewski para realizar ingeniería reversa sobre las aplicaciones de Android. Está publicado bajo la Apache License 2.0.

Tiene la capacidad de descompilar todos los elementos de un apk y volverlos a compilar. En particular, puede leer los xml y el archivo resources.arsc, desensamblar el archivo classes.dex al lenguaje *smali* y facilitar la corrección de código smali.

Androguard [6]: Conjunto de herramientas desarrolladas en Python que manipulan los archivos dex (Dalvik bytecode), apk (Android Package), xml binarios, class (Java virtual machine) y jar (Java application). En particular permiten traspasar estos archivos a objetos de Python, útiles para su análisis.

Además, Androguard tiene las siguientes herramientas:

- Leer los xml binarios en xml clásicos.
- Ver y calcular la diferencia entre dos aplicaciones. Permite ver la cantidad de métodos idénticos, modificados, nuevos y borrados. También puede ser usado para medir la eficiencia de los ofuscadores y determinar si una aplicación ha sido pirateada.
- Soporte nativo para el formato dex usando una librería de C++.
- Realizar ingeniería reversa de los apk.
- Exportar la aplicación en formato *gexf*[2] para su visualización con gephi[3]. Gephi es un programa que permite dibujar grafos en distintos formatos, en particular en *gexf*, en el cual se visualizan las clases de una aplicación y la relación entre ellas.

APKInspector [36] Parte de The Honeynet Project [26], APKInspector es una herramienta gráfica que realiza análisis estático sobre los apk. Posee una interfaz gráfica para visualizar las aplicaciones. Ésta muestra, entre otras cosas:

- Los permisos utilizados,
- Los componentes de la aplicación,
- El gráfico de las clases generado por Androguard,
- El código smali [13] obtenido,
- El contenido mismo del apk,
- Los xml presentes en el paquete.

[2]**Formato *gexf*:** `http://gexf.net/format/index.html`
[3]`http://gephi.org/`

4.1.3.2. Herramientas para leer xml

Aparte de las herramientas ya mencionadas para transformar los xml binarios en xml clásicos, existe una llamada AXMLPrinter2[4]. Su resultado es similar al obtenido con apktool.

4.1.3.3. Herramientas para leer dex

Desensambladores

dex2jar: Es una herramienta basada en Java que convierte los dex en jar[5]. Su resultado es bastante bueno y puede ser luego utilizado con un descompilador Java para obtener el código fuente de la aplicación.

smali/baksmali [13]: Es un ensamblador y desensamblador para el formato dex. La sintaxis está ligeramente basada en Jasmin[6] y dedexer. Su resultado es el más correcto y fiel al código dex compilado. Gracias a ésto, tiene la capacidad de recompilar el código obtenido, sin la necesidad del código fuente original. Ésto permite hacer ligeras modificaciones al código y luego recompilar la aplicación.

Ida Pro [27]: Es un desensamblador para diversos formatos. En particular, permite desensamblar los archivos de formato ELF, binarios de la arquitectura ARM presente en la mayoría de los dispositivos móviles. Algunas aplicaciones o malware traen binarios ya compilados. Ida Pro permite analizar estos archivos. La versión completa es pagada, sin embargo existe una versión de prueba que acepta el formato ELF.

ded: Es un desensamblador del formato dex. Fue realizado por Damien Octeau, William Enck y Patrick McDaniel y utilizado para analizar más de 1000 aplicaciones Android usando ingeniería reversa [22]. Además, publicaron un segundo paper [8] en donde analizan las vulnerabilidades y medidas de seguridad en las 1100 aplicaciones gratis más populares.

dedexer [23]: Es otro desensamblador del formato dex. Fue realizado y es mantenido actualmente por Gabor Paller. Su sintaxis está basada en la sintaxis de Jasmin, un lenguaje de ensamblador para la Java Virtual Machine.

[4] **AXMLPrinter2:** `http://code.google.com/p/android4me/downloads/list`
[5] **dex2jar:** `http://code.google.com/p/dex2jar/`
[6] `http://jasmin.sourceforge.net/`

Descompiladores java

jd-gui [7]: Es un descompilador java que ofrece una interfaz gráfica para navegar a través de las clases de una aplicación en jar o class. Práctico para usar después de dex2jar y poder analizar las clases de la aplicación.

DJ Java Decompiler [21] Es una aplicación para Windows que permite descompilar y desensamblar aplicaciones java. Una alternativa para jd-gui.

java-decompiler.com Es una colección de descompiladores, desensambladores, ensambladores y ofuscadores para Java.

Jad [15] Otro descompilador de Java, usado por defecto en Androguard.

4.2. Descripción del sistema desarrollado: Samsaraa

4.2.1. Requerimientos

El objetivo principal del software es analizar estáticamente muestras de malware, recolectar estadísticas de su análisis, calificar las muestras en base a su riesgo y seleccionar las más peligrosas. Ésto se puede detallar en los siguientes requerimientos:

- Crear un software que realice un análisis estático de una muestra, que puede ser un archivo apk o dex.
- El sistema debe generar un reporte detallado del análisis.
- El sistema debe ser capaz de vincular las características en común de las muestras analizadas. El objetivo es poder recolectar estadísticas sobre los componentes de cada aplicación y las semejanzas entre ellas.
- El sistema debe dar un valor de riesgo a cada muestra analizada. Este puntaje será utilizado para generar un ranking y poder seleccionar las más sospechosas.
- Dado que uno de los objetivos es poder separar el malware peligroso entre una gran cantidad de aplicaciones, el sistema debe ser capaz de (o fácilmente escalable para) analizar grandes cantidades de malware en poco tiempo.
- Crear una interfaz web para visualizar los reportes y las estadísticas recolectadas.
- El software debe ser lo suficientemente modular y expansible de forma que sea simple agregar una nueva característica para analizar o un nuevo tipo de archivo de entrada (como elf o jpg).

Cabe destacar que en un futuro, este sistema se comunicara con otros servicios de McAfee para formar parte de un ecosistema más grande. En particular, McAfee desarrolló un crawler (recolector de aplicaciones) que está constantemente descargando aplicaciones desde internet, como el Android Market, markets alternativos y diversos sitios que tienen aplicaciones para descargar. Entonces, todas las aplicaciones descargadas por el crawler serán enviadas a Samsaraa para ser analizadas y calificadas. De esta forma, los investigadores podrán seleccionar las muestras más riesgosas y analizarlas manualmente, con más detalle.

En la figura 4.1 se muestran los casos de uso en base a los requerimientos del sistema.

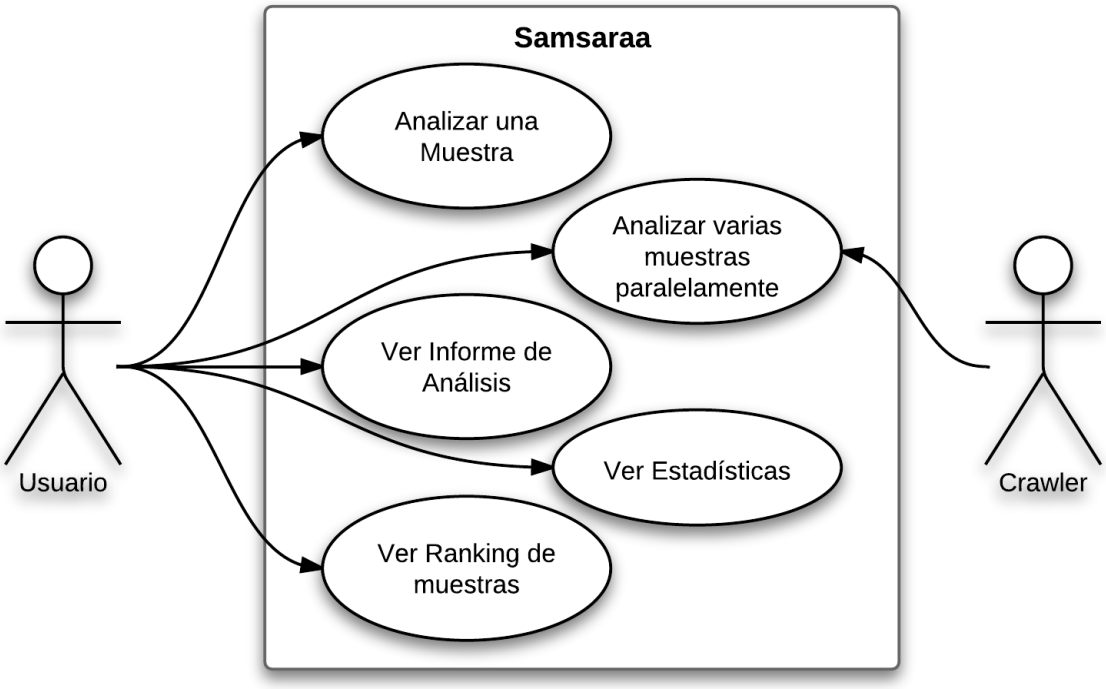


Figura 4.1: Diagrama de casos de uso del sistema.

4.2.2. Elección de las herramientas

Como lenguaje de programación se eligió Python por los siguientes motivos:

- *Alto nivel*: Al ser un lenguaje de alto nivel permite enfocarse en el verdadero problema, sin preocuparse tanto por los detalles del lenguaje.
- *Multiplataforma*: Tiene la ventaja que puede ser ejecutado en la mayor parte de los sistemas operativos.
- *Librerías*: Existe una gran cantidad de librerías que ayudan con las tareas más tediosas. En particular, para la ingeniería reversa de las aplicaciones de Android, como Androguard.
- *Legibilidad*: Dada la filosofía y sintaxis de Python, es fácil leer y entender el código, facilitando su posterior mantenimiento.

- *Multi-propósito*: Se puede usar el mismo lenguaje tanto para el análisis como para la interfaz web y comunicarlos sin problemas.

Para la interfaz web se eligió usar el framework MVC Django pues permite enfocarse en la lógica misma de la aplicación, sin desperdiciar tiempo en tareas más comunes y repetitivas (como la conexión a la base de datos). Además, Django provee un ORM que permite diseñar la capa de datos en Python y automáticamente traspasarlo a una base de datos relacional.

Para la base de datos se eligió MySQL. También se investigaron las bases de datos no relacionales. Sin embargo se eligió una relacional por los siguientes motivos:

- La estructura del problema es suficientemente estándar, por lo tanto no es necesaria una base de datos más flexible. Además, como se requiere obtener estadísticas transversales, los datos almacenados tienen que tener una estructura dada.
- Dado que en el futuro este sistema será integrado con otras herramientas de McAfee, es necesaria una base de datos accesible por el resto de los sistemas.
- El modelo relacional lleva varios años en el mercado y es utilizado por numerosas empresas y proyectos, lo que prueba su validez y utilidad.

Por otro lado, se eligió MySQL frente a otras bases de datos relacionales pues se estima que es suficiente para los requerimientos. Además, como se usará con el ORM de Django, en un futuro, en caso de necesitar una base de datos más potente se puede cambiar por PostgrSQL u Oracle, y el código de la aplicación seguirá siendo el mismo.

4.2.3. Diseño

4.2.3.1. Descripción general

El sistema está divido en tres grandes módulos que interactúan entre si:

- **Analizador**: es el módulo encargado de analizar una muestra, recolectar toda su información y guardarla en la base de datos.
- **Sistema de Rating**: es el módulo encargado de analizar las características de una muestra y definir el nivel de riesgo asociado.
- **Interfaz Web**: es la interfaz que permite interactuar con el usuario y realizar las tareas de forma intuitiva.

4.2.3.2. Arquitectura general

En la figura 4.2 se muestra la arquitectura general del sistema y como interactúan los distintos módulos y sus componentes.

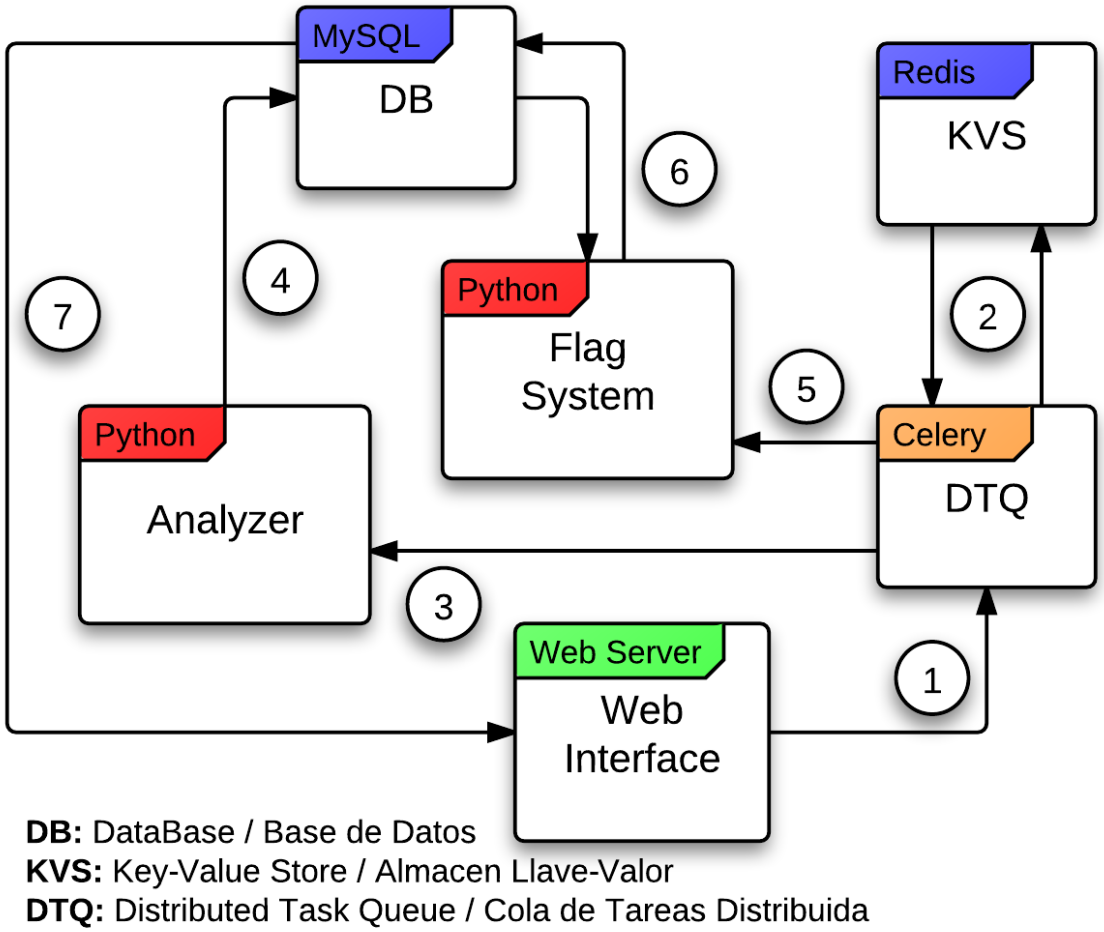


Figura 4.2: Arquitectura general de Samsaraa.

1. A través de la interfaz web, el usuario entrega las muestras para ser analizadas. Esta interfaz envía las peticiones de análisis a Celery, una cola de tareas distribuida que permite ordenar y ejecutar procesos paralelamente.

2. Celery utiliza Redis, un almacén llave-valor, para encolar las tareas y guardar registro de ellas.

3. Celery envía las muestras al modulo Analizador.

4. El modulo Analizador analiza las muestras y guarda el resultado en la Base de Datos.

5. Cuando una muestra fue analizada, Celery envía el hash de la muestra al Sistema de Rating.

6. El Sistema de Rating lee la información de la muestra en la Base de Datos, calcula su puntaje y luego almacena los resultados en ella.

7. Finalmente, la Interfaz Web puede leer de la Base de Datos todo el análisis realizado y mostrar los resultados y estadísticas al usuario.

4.2.4. El modulo analizador: Analyzer

El análisis consiste en aplicar consecutivamente una lista de micro-analizadores llamados *InformationPickers* los cuales reciben la muestra, la analizan y extraen la información requerida.

La función de estos *InformationPickers* es que cada uno se encarga de realizar una pequeña parte del análisis (Por ejemplo, extraer las Activities o recolectar las funciones usadas) y de guardar el resultado en la base de datos. Cada uno de ellos es independiente del resto y deben poder realizarse en cualquier orden. Ésto permite que se puedan agregar nuevos micro-analizadores a medida que se requiera, sin realizar cambios al resto de las clases.

Además, existe un archivo de configuración *samsaraa_config* el cual contiene la lista de *InformationPickers* que se usaran para el análisis. Ésto da la libertad para elegir cuales analizadores usar en un caso específico, sin hacer cambios al código.

Cabe destacar que cada vez que el sistema analiza una muestra, revisa si es que esa muestra ya existe o no (comparando el hash sha1) y en caso de que no exista procede a realizar el análisis. En caso contrario, se utiliza la información existente y finaliza el análisis. El objetivo es optimizar y no tener que analizar varias veces la misma muestra en caso de que existan varias copias en un mismo set o ya haya sido analizada antes.

En la figura 4.3 se muestran las clases principales del modulo Analizador, las cuales se detallan a continuación.

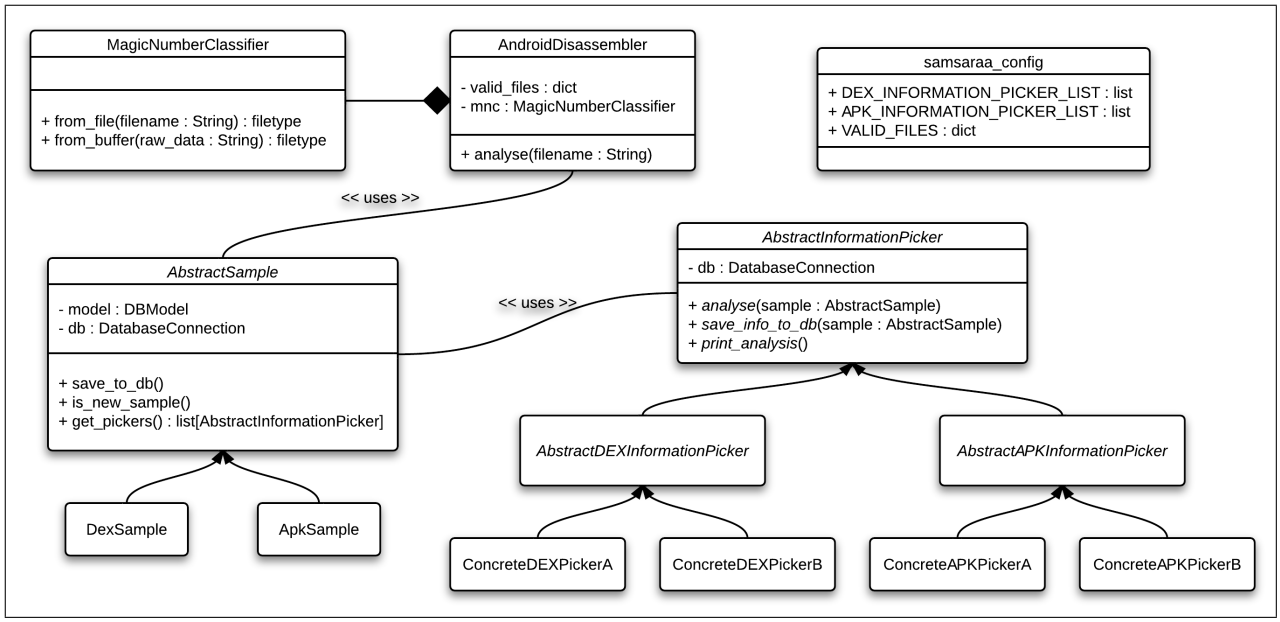


Figura 4.3: Diagrama de Clases simplificado del modulo analizador.

4.2.4.1. MagicNumberClassifier

Esta clase se utiliza para saber el tipo de un archivo. En particular, para averiguar si una muestra es *apk*, *dex* u otro tipo y para conocer todos los tipos de archivos incluidos en un apk

(como imágenes, pdf, sonidos, etc). Para ésto, la clase se basa en el *MagicNumber*, es decir, en los primeros bytes del archivo que indican qué tipo de archivo es.

La clase posee dos funciones, ambas retornan el tipo de archivo, pero se diferencian en que la primera (*from_file(filename)*) recibe el nombre del archivo a analizar y la segunda (*from_buffer(raw_data)*) recibe el contenido del archivo en si.

4.2.4.2. AbstractSample

Representa una muestra y es la estructura que guarda la información y los identificadores de la muestra a lo largo de todo el análisis. Esta estructura es la que reciben los *InformationPickers* para realizar su análisis. Contiene la dirección del archivo, los hashes, y la interfaz para acceder a la base de datos. Por lo mismo, esta clase es la que se encarga de realizar la primera fase del análisis. Ésta consiste en calcular los hashes, crear la interfaz para la base de datos, revisar si existe, obtener la lista de *InformationPickers* que se usarán y realizar tareas comunes para todos los *InformationPickers*.

Esta clase es abstracta y para poder analizar un tipo de archivo en particular es necesario crear la clase concreta correspondiente. Actualmente, se acepta como archivo de entrada un apk o un dex y por lo tanto existen las clases *ApkSample* y *DexSample* respectivamente.

A modo de ejemplo, una de las tareas comunes que realiza *DexSample* es desensamblar el archivo dex y guardar el resultado en memoria. Dado que todos los *InformationPickers* trabajan sobre el código desensamblado, es necesario que esta tarea se realice al principio y el resultado sea accesible por todos ellos. Por lo tanto, *DexSample* es el lugar indicado para hacerlo.

4.2.4.3. AbstractInformationPicker

Es la clase base para definir los *InformationPickers*. Contiene tres métodos que las subclases deben implementar:

- $analyse(sample)$: recibe el modelo *Sample* y realiza el análisis,
- $save\_info\_to\_db(sample)$: recibe el modelo *Sample* y guarda el resultado del análisis en la base de datos,
- $print\_analysis()$: usado principalmente para realizar debugging, sirve para imprimir el resultado del análisis por consola.

Todas las clases que heredan de *AbstractInformationPicker* son los micro-analizadores que se encargan de recolectar toda la información del análisis y guardarla en la base de datos. Como todos tienen la misma clase base y por ende la misma interfaz el sistema se vuelve flexible, dado que es simple agregar un nuevo *InformationPicker*, cambiar otro y/o eliminar un tercero.

Las clases *AbstractAPKInformationPicker* y *AbstractDEXInformationPicker* están para agrupar tareas en común para los apk o dex respectivamente. Finalmente, los *ConcretePickers*, son las

clases que efectivamente realizan el análisis. En la sección 4.2.5 se detallan los distintos *Pickers* que están implementados actualmente tanto para dex como para apk.

4.2.4.4. AndroidDisassembler

Es el módulo principal que se encarga de aplicar cada uno de los *Information Pickers* a la muestra. Posee un único método: *analyse*(*filename*) que recibe el nombre de archivo de la muestra, la analiza y retorna el resultado. En el código 4.1 se muestra una versión simplificada del código de AndroidDisassembler.

Código 4.1: AndroidDisassembler.analyse()

```
1  class AndroidDisassembler(object):
       def analyse(self, filename):
3          """Analyse sample in filename"""

5          filetype = self._filter(filename)
           sample = self.valid_files[filetype](filename)
7          info_pickers = sample.get_pickers()

9          for ip in info_pickers:
               ip.analyse(sample)
11
           sample.save_to_db()
13
           for ip in info_pickers:
15             # ip.print_analysis()
               ip.save_info_to_db(sample)
17
           return sample
```

La función *_filter(filename)* verifica el tipo de archivo y revisa que sea uno de los aceptados, actualmente apk o dex. La linea 6, entrega la clase *Sample* adecuada según el tipo de archivo. Por ejemplo, si la muestra es un apk, *sample* será una instancia de *ApkSample* debidamente inicializada. Luego se obtiene la lista de *InformationPickers* y finalmente se procede al análisis a través de cada uno de estos.

En la etapa del análisis, primero se analiza la muestra (lineas 9-10), luego se guarda la estructura principal (linea 12) y finalmente se almacena toda la información recolectada por cada uno de los *InformationPickers* (lineas 14-16).

4.2.4.5. samsaraa_config

Éste es el archivo de configuración del sistema. En particular, contiene:

- APK_INFORMATION_PICKER_LIST: Lista de *InformationPickers* que se usarán para el análisis de apk y el orden de éstos.
- DEX_INFORMATION_PICKER_LIST: Lista de *InformationPickers* para dex.
- VALID_FILES: Diccionario que contiene los tipos de archivo aceptados y la clase hija de *AbstractSample* asociada.

En el apéndice A se muestra un ejemplo de este archivo. También tiene otros parámetros como la configuración de la base de datos y de la carpeta temporal utilizada para desempacar los apk.

4.2.4.6. Funcionamiento del Analizador

A continuación se detalla la interacción entre las distintas clases del modulo Analizador. En la figura 4.4 se muestra el diagrama de secuencia para el caso de análisis de un apk.

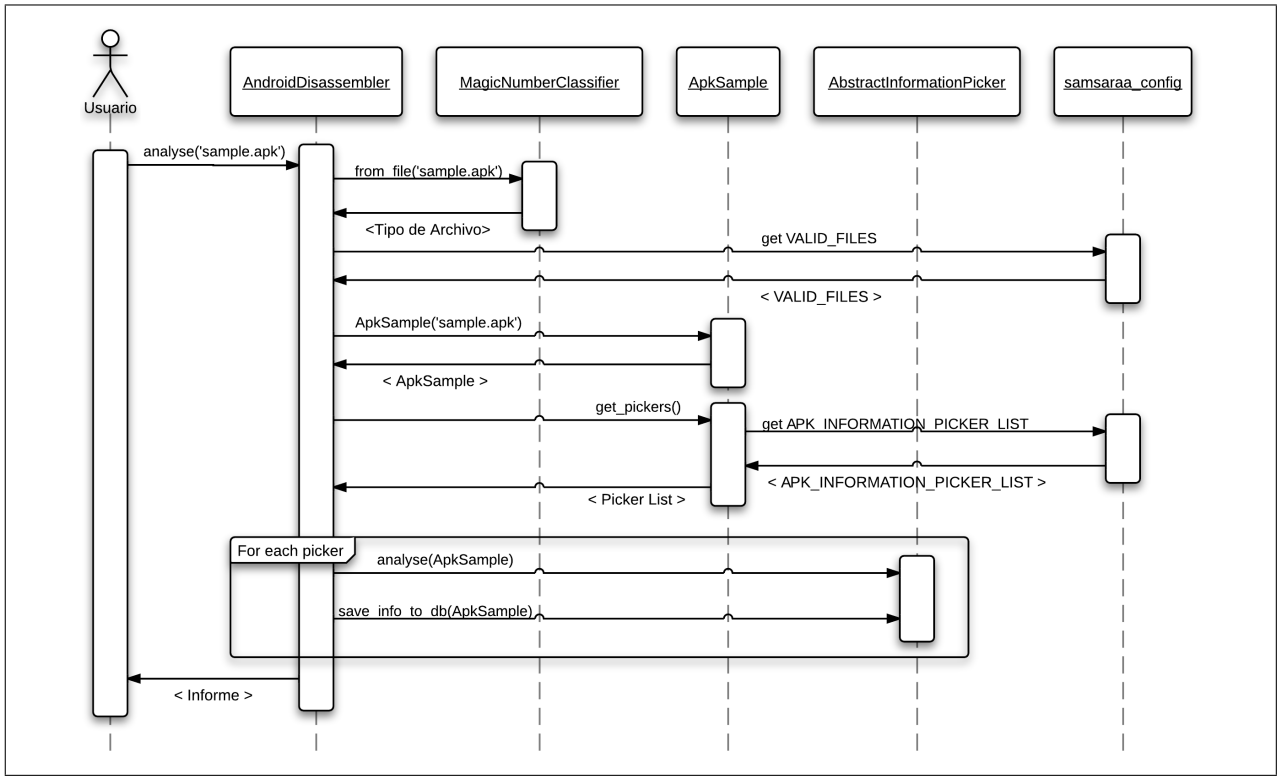


Figura 4.4: Diagrama de Secuencia para el análisis de un apk.

El analizador funciona de igual manera para cualquier tipo de archivo de entrada, sin embargo, actualmente solo procesa apk y dex. Como ejemplo, para el caso de un apk, la secuencia es la siguiente:

1. El usuario, ya sea por línea de comando o por medio de alguna otra interfaz, como la interfaz web, entrega el archivo a analizar.

2. La clase *AndroidDisassembler* pregunta a *MagicNumberClassifier* que tipo de archivo es.

3. Luego lee el archivo de configuración *samsaraa_config* para obtener los tipos de archivos validos y verifica que el archivo recibido sea uno de los aceptados.

4. A continuación, se crea la estructura *ApkSample*, que sirve de contenedor de la muestra durante el análisis y que realiza la primera parte de éste.

5. *AndroidDisassembler* le pregunta a *ApkSample* cuales son los *InformationPickers* que se usarán. Éste, a su vez, los extrae del archivo de configuración y los entrega de forma adecuada para su uso.

6. *AndroidDisassembler* utiliza los *InformationPickers* recibidos para hacer el análisis, recorriendo cada uno de ellos.

7. Finalmente, una vez que la muestra en *ApkSample* pasó por todos los *InformationPickers*, se entrega el resultado del análisis.

4.2.5. Los InformationPickers y el modelo de datos

Los *AbstractSample* y principalmente los *InformationPickers* son los encargados de recolectar la información de una muestra. Esta información es almacenada en la base de datos para luego poder sacar estadísticas, estudiar las muestras en detalle y también a gran escala. En esta sección, se explican los diversos *InformationPickers* y cómo las características que recolectan son guardadas en el modelo de datos.

En la figura 4.5 se muestra el modelo de datos para almacenar las muestras.

La información extraída por los *AbstractSample* es la siguiente: hashes (MD5, SHA1, SHA256), nombre, dirección, tamaño del archivo, fecha y tiempo del análisis. Ésta es almacenada en los modelos *ApkFile* y *DexFile* (para apk y dex respectivamente). Por otro lado, cada *InformationPicker* guarda su información en un modelo propio. Estos modelos están asociados a una clase propia (del mismo nombre) que se encarga de la interacción con la base de datos y de funciones específicas para cada uno. Y luego, estos modelos están vinculados con el *AbstractSample* correspondiente. De esta forma, los *InformationPickers* que recolectan información de un apk guardan su información en modelos asociados a *ApkFile* (ver figura 4.6) y de igual forma los de dex asociados a *DexFile* (ver figura 4.7).

Cada muestra (apk, dex o elf) está asociada a una *Source*. Este modelo sirve para agrupar las aplicaciones, ya sea por la fuente desde donde fueron obtenidas o alguna otra agrupación arbitraria. El objetivo es poder comparar las características entre distintos grupos de aplicaciones teniendo toda la información en la misma base de datos.

Esta estructura en estrella proporciona un modelo flexible y a la vez robusto para almacenar toda la información del análisis. En efecto, al crear un nuevo *InformationPicker* solo hay que agregar su modelo y vincularlo al *Sample* correspondiente, y a la vez, basta con tener el hash de una muestra para obtener toda su información.

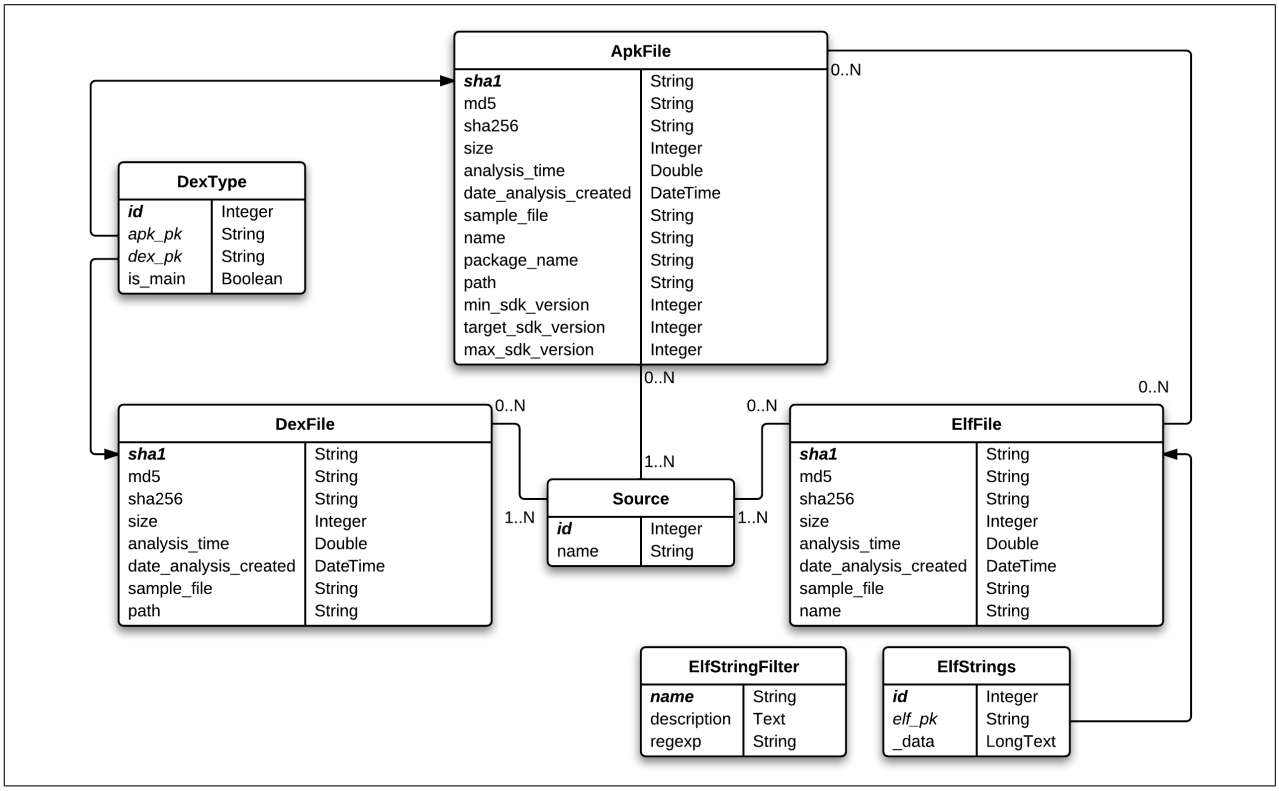


Figura 4.5: Diagrama del Modelo de Datos para las muestras.

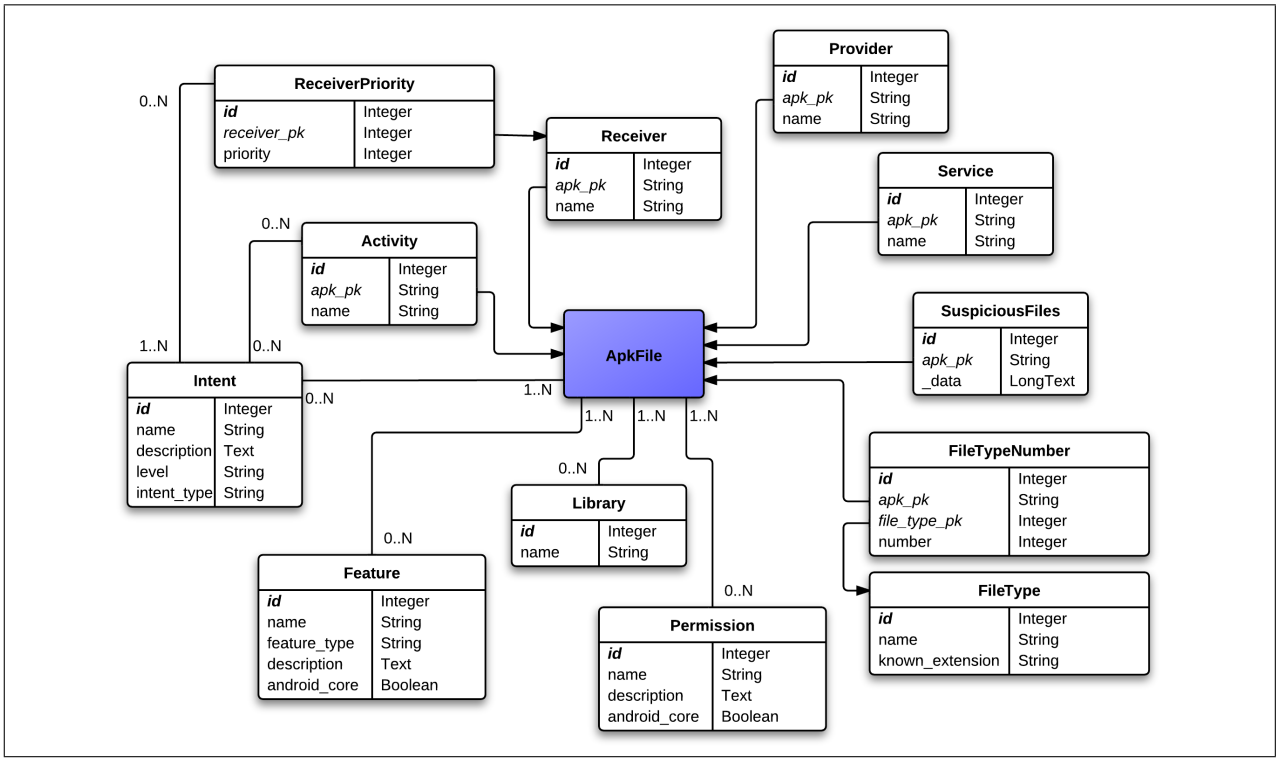


Figura 4.6: Diagrama del Modelo de Datos para la información de un apk.

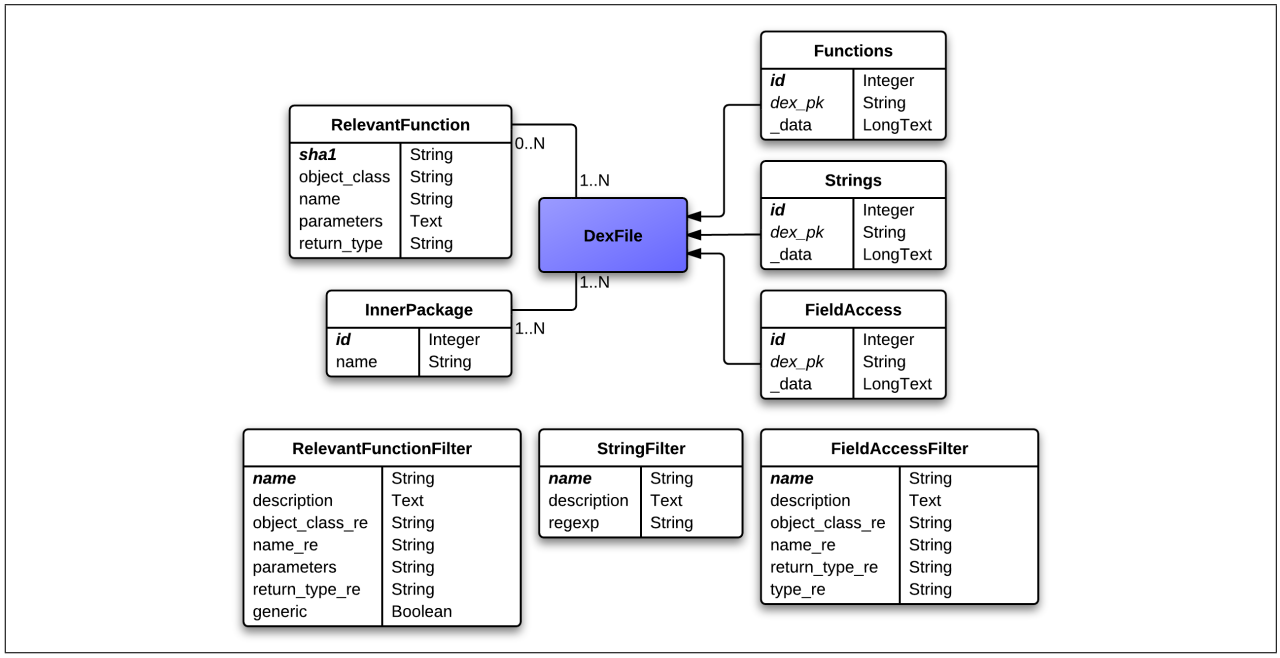


Figura 4.7: Diagrama del Modelo de Datos para la información de un dex.

En la tabla 4.1 se muestra un cuadro resumen de los *InformationPickers* implementados actualmente. Estos están clasificados según 2 criterios:

1. Si sirven para analizar dex o apk
2. Si la información recolectada es transversal o no.

La diferencia entre una característica o información transversal y no transversal es la siguiente:

- *Transversal*: son características de una muestra que suelen aparecer en más de una aplicación y que por lo tanto representan información relevante para obtener estadísticas. Por ejemplo, los Permisos e Intents son elementos transversales pues existen varias aplicaciones que usan un mismo permiso u ocupan un mismo Intent. En este caso, cada característica es guardada como una fila en la base de datos y está asociada a cada una de las muestras que la tiene. Para mantener la coherencia, cada vez que se agrega una nueva característica, se revisa la base de datos para ver si ya está y sólo exista una instancia de cada una. De esta forma, se mantiene el registro de qué aplicaciones ocupan qué permisos o de cuántas aplicaciones usan un permiso en particular.
- *No Transversal*: son características particulares de cada muestra que no proporcionan ninguna relación entre las aplicaciones pues son muy específicas para cada una. Por ejemplo las Activities son no transversales pues es raro que dos aplicaciones tengan una Activity con el mismo nombre, y el hecho de que existan dos que sí lo tengan, no indica nada. Sin embargo, estas características son relevantes a la hora de analizar la muestra en detalle y son listadas en el informe generado. Para este caso, cada característica también es guardada en una sola fila, sin embargo sólo está asociada a una muestra y no hay chequeo de unicidad.

	Transversal	No Transversal
DEX	InnerPackagePicker FunctionsPicker	StringsPicker FieldAccessPicker
APK	PermissionsPicker IntentsPicker LibrariesPicker FeaturesPicker FileTypesPicker InnerAPKAnalyser InnerDEXAnalyser InnerELFAnalyser	PackageInformationPicker ActivitiesPicker ReceiversPicker ServicesPicker ProvidersPicker SuspiciousFilesPicker

Tabla 4.1: InformationPickers

En la tabla 4.2 se muestra una breve descripción de los *InformationPickers* los cuales se detallan a continuación.

4.2.5.1. InnerPackagePicker

Sirve para analizar los dex y provee información transversal. Las aplicaciones se escriben en Java y por lo mismo se organizan en packages. Cuando una aplicación se compila y se genera el apk, el archivo *classes.dex* almacena esta organización. Estos packages internos (InnerPackages) son extraídos por el *InnerPackagePicker* y almacenados en el modelo *InnerPackage*, guardando su nombre.

Generalmente, las aplicaciones legítimas tienen un sólo package, mientras que las troyanizadas suelen contener el package original y el malicioso. Por lo tanto, guardar ésto aporta información relevante para el análisis. De igual forma, se considera información transversal pues si dos dex tienen un mismo package, podría ser el mismo package malicioso.

4.2.5.2. FunctionsPicker

Sirve para analizar los archivos dex y provee información tanto transversal como no transversal. Este *InformationPicker* se encarga de recolectar todas las funciones utilizadas en la aplicación. Estas llamadas son extraídas del archivo classes.dex desensamblado a smali (ver página 22 y [13]).

El problema con las funciones es que una aplicación, por muy simple que sea, tiene muchas llamadas a funciones, con lo cual se vuelve poco óptimo guardarlas como información transversal: una función por fila, con chequeo de unicidad y asociación con cada muestra en la que se encuentra. Sin embargo, las llamadas a funciones son la información más importante para saber qué hace una aplicación (cada acción importante es una función: enviar SMS, conectarse a internet, etc), por lo que sería ideal guardarlas transversalmente y poder comparar entre distintas aplicaciones.

Para resolver este problema se decidió crear dos modelos y usar filtros: *Functions*, *Relevant-*

Nombre	Descripción	Objetivo
InnerPackagePicker	Extrae el nombre de los packages de la aplicación.	Identificar una muestra con mas de un package.
StringsPicker	Extrae los strings del código fuente.	Encontrar URLs, números de teléfonos y parámetros sensibles.
FunctionsPicker	Extrae todas las llamadas a funciones del código fuente.	Detectar funcionalidades clave.
FieldAccessPicker	Extrae los accesos a variables estáticas realizados por la aplicación.	Identificar el acceso a parámetros importantes.
PackageInformationPicker	Extensión al análisis básico de un apk.	Obtener los niveles de la API requeridos y el package principal.
PermissionsPicker	Extrae los permisos requeridos por la aplicación.	Encontrar permisos sospechosos.
IntentsPicker	Extrae los Intents del AndroidManifest.	Identificar comportamientos sospechosos.
LibrariesPicker	Extrae las librerías utilizadas por la muestra.	Detectar funcionalidad adicional.
FeaturesPicker	Extrae los Features del AndroidManifest.	Reconocer el uso de hardware especial.
FileTypesPicker	Extrae el numero y los tipos de archivos del apk.	Detectar tipos de archivos especiales.
ActivitiesPicker	Extrae las Activities del AndroidManifest.	Aportar información sobre el apk.
ReceiversPicker	Extrae los Broadcast Receiver definidos en el AndroidManifest.	Identificar aplicaciones que reaccionan a eventos del sistema.
ServicesPicker	Extrae los Services definidos en el AndroidManifest.	Detectar ejecuciones en segundo plano que posea la aplicación.
ProvidersPicker	Extrae los Content Providers del AndroidManifest.	Aportar información sobre el apk.
SuspiciousFilesPicker	Detecta archivos sospechosos dentro del apk.	Detectar archivos con extensión diferente al Magic Number.
InnerAPKAnalyser	Analiza los posibles apk internos.	Encontrar funcionalidad adicional escondida en otra aplicación.
InnerDEXAnalyser	Analiza los archivos dex en un apk.	Describir la funcionalidad principal.
InnerELFAnalyser	Analiza los posibles archivos elf en un apk.	Identificar exploits.

Tabla 4.2: Descripción de los InformationPickers

Function y *RelevantFunctionFilter* (ver figura 4.7). En resumen, el modelo *Functions* guardará todas las llamadas a funciones de un dex (no transversalmente) y el modelo *RelevantFunction* guardará sólo las funciones importantes (transversalmente), las cuales son definidas por los filtros *RelevantFunctionFilter*. Este diseño permite calcular eficientemente estadísticas transversales de las funciones importantes y a la vez, tener todas las funciones a mano para un análisis detallado, pero más lento.

Las llamadas a funciones se caracterizan e identifican por la firma, la cual se puede separar en 4 partes:

- *object_class*: representa el tipo de objeto al cual pertenece la función.
- *name*: es el nombre de la función.
- *parameters*: es una lista separada por comas del tipo de los parámetros.
- *return_type*: es el tipo del valor de retorno.

Cabe destacar que el análisis es estático, por lo que no es posible saber cuántas veces se llama una función, cuál es el valor de los parámetros o cuál es el valor de retorno.

Por ejemplo, en la función *indexOf* (código 4.2): *object_class* sería "String", *name* sería "indexOf", *parameters* sería "String, int" y *return_type* sería "int". Sin embargo, los valores "test", "t" y "1" son desconocidos estáticamente.

Código 4.2: Función indexOf

```
1        String s = "test";
2        int i = s.indexOf("t", 1)
```

Ésta es la estructura utilizada para guardar las funciones en la base de datos sin perder información. El modelo *Functions* almacena todas las funciones en una sola fila, como arreglo de Python comprimido (campo *_data* de tipo *longtext*) y asociada al *DexFile* correspondiente. Ésto ahorra espacio en la base de datos al no tener que guardar un registro por función. La desventaja es que no se pueden hacer consultas específicas a la base de datos(usando SQL), pero es posible extraer todo el arreglo y trabajarlo en Python. Por otro lado, el modelo *RelevantFunction* guarda sólo las funciones importantes, una por fila, asociadas a todos los *DexFile* donde se encontraron y usa como llave primaria un hash de las 4 partes. El hash permite verificar la unicidad de manera eficaz y eficiente, y el hecho que esté una por fila facilita el cálculo de estadísticas.

Finalmente, una función se considera importante si es que calza con alguno de los filtros definidos. Durante el análisis de *FunctionsPicker*, una vez que se obtienen todas las funciones de una muestra, se usan todos los filtros "generic" para obtener las funciones importantes y guardarlas transversalmente. La diferencia entre un filtro "generic" y uno "no generic" se explica mas adelante, en la sección: Calificación de aplicaciones, página 52.

Para definir un *RelevantFunctionFilter* se utilizan cuatro expresiones regulares, una para cada una de las partes de una función y de las cuales al menos una tiene que estar definida. Para que una

función calce con un filtro, todas las expresiones regulares definidas deben calzar con la función. Por último, para mejor usabilidad se agrega un nombre (usado también como llave primaria) y una descripción.

4.2.5.3. StringsPicker

Sirve para analizar los dex y provee información no transversal. Este *InformationPicker* se encarga de recolectar todos los strings utilizados en la aplicación. Éstos son extraídas del archivo classes.dex desensamblado a smali (ver página 22 y [13]). Es importante analizar los strings pues en ellos se puede encontrar las URLs visitadas, correos y números de teléfono utilizados, nombres de archivos especiales y diversos parámetros que ayudan a comprender que realiza una aplicación.

Al igual que el modelo *Functions*, *Strings* guarda todos los strings de un dex como un arreglo de Python en el campo *_data* y asociado al *DexFile* correspondiente. Dado que una aplicación contiene muchos strings, no tiene sentido guardarlos como una característica transversal y de esta forma se optimiza el espacio en la base de datos.

También existen filtros para strings: *StringFilter*, con un nombre, descripción y expresión regular. Estos filtros encuentran rápidamente strings relevantes en una aplicación, los cuales sí pueden ser analizados transversalmente. Por ejemplo, las URL generalmente están expresadas en strings y puede que varios malware utilicen la misma URL para conectarse a un servidor malicioso. Más adelante se explica el uso detallado de estos filtros.

4.2.5.4. FieldAccessPicker

Este último *InformationPicker* para dex provee información no transversal. Su objetivo es recolectar los accesores (setters y getters de variables estáticas o de instancia de las clases) encontrados en el archivo classes.dex desensamblado a smali (ver página 22 y [13]). Todos estos accesos son guardados en el modelo *FieldAccess* como arreglo de python (al igual que los strings y las funciones).

Es importante ver estos accesos, pues muchas opciones de configuración de Android son variables de instancia o estáticas de una clase en particular. Además, también se puede acceder a algunos *Content Provider* a través de estos campos. Por ejemplo, la lista de contactos se puede acceder a través de un acceso a la clase *android.provider.ContactsContract* y varias características del sistema están en *android.os.Build*, como *android.os.Build.HARDWARE* (el nombre del hardware) o *android.os.Build.MANUFACTURER* (el nombre del fabricante).

El problema, al igual que con las funciones, es que una aplicación tiene muchos de estos accesos y la mayoría son accesos a variables de la misma aplicación (y no a clases de Android o del sistema), los cuales a priori, no proveen información útil. Es por ésto que se decidió guardarlos como información no transversal y utilizar los filtros *FieldAccessFilter* para encontrar los importantes.

Estos accesos están caracterizados por cuatro parámetros:

1. *object_class*: el objeto al cual pertenece la variable.
2. *name*: el nombre de la variable.
3. *return_type*: el tipo de la variable (int o String por ejemplo).
4. *type*: el tipo de acceso. Es decir, si es una lectura o escritura (get o put) y si corresponde a una variable de instancia o a una variable estática (iget/iput y sget/sput respectivamente).

Los filtros *FieldAccessFilter* se definen de forma similar a los filtros de funciones. Utilizan cuatro expresiones regulares, una para cada parte del acceso, de las cuales tiene que haber al menos una definida. Para calzar un acceso, éste tiene que coincidir con todos los patrones definidos.

4.2.5.5. PackageInformationPicker

Sirve para analizar los apk y provee información no transversal. Este *InformationPicker* es una extensión al análisis básico realizado por *ApkSample* y se encarga de recolectar varios parámetros del AndroidManifest que caracterizan la aplicación. Por ejemplo, el nivel de la API utilizada y el nombre del Package son extraídos por este Picker. La información es guardada en *ApkFile*, junto con el resto de las características especificas del apk.

4.2.5.6. PermissionsPicker

Sirve para analizar los apk y provee información transversal. Este *InformationPicker* se encarga de recolectar los permisos (Permissions) requeridos por la aplicación. El modelo *Permission* almacena el nombre, la descripción especificada en el Developer's Guide [11] y un flag para indicar si corresponde o no a un permiso oficial (*android_core*). Durante el análisis puede que se encuentren nuevos permisos, que no están descritos en el Developer's Guide. En este caso también se guardan, pero sin descripción y como "no oficial".

Los permisos de una aplicación son fundamentales pues otorgan una visión general de que es lo que puede hacer o no una aplicación. Por lo mismo, también se consideran información transversal dado que es muy útil saber cuantas aplicaciones usan tal o cual permiso, obteniendo un parámetro para clasificarlas.

4.2.5.7. IntentsPicker

Sirve para analizar los apk y provee información transversal. Este *InformationPicker* se encarga de recolectar los Intents especificados en el AndroidManifest (ver página 8). Éstos pueden estar asociados ya sea a una Activity o a un Broadcast Receiver y son guardados en el modelo *Intent*. La información que se almacena de cada uno es: el nombre y la descripción del Intent, tal como está especificado en el Developer's Guide [11] y dos parámetros más que sirven para clasificar el Intent:

- *level* puede tomar un valor entre: *standard*, *core* u *other*. *standard* son los Intents definidos como Standard en [11]; *core* son el resto de los Intents especificados en [11] y *other* son aquellos encontrados en las muestras analizadas y que corresponden a Intents de alguna aplicación específica.
- *intent_type* puede tomar un valor entre *activity*, *broadcast* u *other* y representa el elemento donde fue encontrado el Intent. *other* es utilizado cuando el Intent encontrado no corresponde ni a una Activity ni a un Broadcast Receiver.

Es importante conocer los Intents de una aplicación pues representan la forma de comunicación con otras aplicaciones y los eventos de los cuales está pendiente. Por ejemplo, para escuchar los eventos: "SMS Recibido", "Usuario Presente" o "Reinicio Completo" hay que definir un Intent específico.

4.2.5.8. LibrariesPicker

Sirve para analizar los apk y provee información transversal. Este *InformationPicker* busca y guarda en el modelo *Library*, el nombre de las librerías especificadas en el Android Manifest.

Actualmente, las librerías no son un indicio muy claro de malware, pero dan una idea de las capacidades que tiene según las librerías que use.

4.2.5.9. FeaturesPicker

Sirve para analizar los apk y provee información transversal. Los Features requeridos en el Android Manifest son recolectados por este *InformationPicker* y guardados en el modelo *Feature*. Un Feature es generalmente una característica del hardware que usa una aplicación (como una cámara o un sensor). El modelo permite guardar el nombre, una descripción, el tipo de Feature (Audio, Location, Camera, etc) y un flag(*android_core*) para indicar si corresponde o no a una de las características oficiales, tal como se indica en el Developer's Guide.

Conocer los Features requeridos por una aplicación permite controlar el hardware que usa la aplicación. Por ejemplo, el uso del micrófono o la cámara pueden ser indicios de un software espía.

4.2.5.10. FileTypesPicker

Sirve para analizar los apk y provee información transversal. Una aplicación viene en forma de apk, la cual es un tipo de archivo comprimido zip que contiene todos los archivos necesarios para que funcione. Este *InformationPicker* se encarga de analizar todos los tipos de archivo (como zip, xml, pdf, doc, apk, elf, etc) que tiene y su cantidad. Esta información es guardada en los modelos *FileType* y *FileTypeNumber*. Están en modelos separados, de forma que los tipos de archivos sean transversales y se pueda obtener fácilmente estadísticas del tipo: "¿Cuántas muestras contienen un archivo elf?" y "¿Cuantas imágenes tiene este apk?". Estos modelos guardan únicamente el nombre del tipo de archivo y el numero de veces que aparece en una muestra.

El tipo de archivo es definido gracias al “Magic Number” del archivo. Es importante conocer que archivos trae un apk dado que, por ejemplo, puede traer un archivo binario (elf) que podría ser un Exploit.

4.2.5.11. SuspiciousFilesPicker

Sirve para analizar los apk y provee información no transversal. Este *InformationPicker* está ligeramente asociado al anterior y se encarga de buscar “Archivos Sospechosos” (SuspiciousFiles). Se define un *SuspiciousFile* como un archivo que tiene un Magic Number diferente de la extensión. Esta información es guardada en el modelo *SuspiciousFiles*.

Existen varios malware que ocultan archivos binarios elf disfrazándolos de imagen (con un nombre y extensión de imagen como logo.jpg o banner.png) de forma que al ver los archivos incluidos, éstos parezcan inofensivas imágenes. Si bien esta técnica no es tan eficiente y el Picker no es tan preciso (pues necesita una extensa lista de asociaciones entre MagicNumber y extensión) siempre es bueno saber cuando un desarrollador está intentando esconder algo, ya que no hay razones benignas para hacerlo.

4.2.5.12. InnerDEXAnalyser e InnerAPKAnalyser

Estos dos *InformationPicker* son un tanto especiales. Ambos utilizan el módulo principal *AndroidDisassembler* para su análisis y proporcionan cierto nivel de recursividad y modularidad. Proveen información transversal de su análisis de apk.

Como los archivos dex se aceptan como archivo de entrada, *InnerDEXAnalyser* se encarga de buscar todos los archivos dex dentro de un apk (generalmente sólo uno) y llamar a *AndroidDisassembler* para su análisis. De esta forma, todos los dex que lleguen al sistema, ya sea como archivo de entrada o en un apk, son analizados de la misma forma. Además, cada apk queda asociado con sus dex y cada dex con todos los apk en donde se encontró. Ésto permite calcular estadísticas de los apk que comparten el mismo dex. Un archivo dex está asociado a un archivo apk vía el modelo intermedio *DexType*, el que indica cuál es el dex principal del apk, es decir, el que corresponde al “classes.dex”.

Por otro lado, *InnerAPKAnalyser* se encarga de analizar los apk que puedan estar adentro del original. Este *InformationPicker* busca los archivos apk dentro de la aplicación y llama recursivamente a *AndroidDisassembler* para su análisis. Es así como se pueden analizar recursivamente todos los apk y guardar una estructura de como fueron encontrados (quienes son “padre” y quienes son “hijo”).

Estos dos análisis (dex y apk) son muy importantes. Todos los apk traen un archivo dex, el código java compilado para Android, y por lo tanto el núcleo de éste. Además, existe mucho malware que trae otras aplicaciones adentro para instalar después y que suelen ser la parte importante y más maliciosa de éste. Esta forma de realizar el análisis genera un sistema modular y extensible, fácil para adaptarse a la evolución del malware y proporcionar la recursividad necesaria para analizar aplicaciones dentro de aplicaciones.

4.2.5.13. InnerELFAnalyser

Sirve para analizar apk y provee información transversal. Este *InformationPicker* se encarga de analizar los archivos elf encontrados en un apk. El análisis consiste en extraer todos los strings en ASCII encontrados en el binario. La información es almacenada en los modelos *ElfFile* y *ElfStrings* (ver figura 4.5). Al igual que con los strings de dex, éstos también son guardados como arreglo de Python por la misma razón. Por lo que, los strings son información no transversal con respecto a los archivos elf. Sin embargo, los archivos elf sí son transversales con respecto a los apk. Ésto da la opción de ver qué apks comparten el mismo archivo elf.

Los archivos elf son importantes de analizar pues, tal como se mencionó anteriormente, son los archivos binarios para Android. Ésto implica que pueden ser Exploit o agregar mayor funcionalidad a la aplicación o malware en cuestión.

Al igual que con los strings encontrados en los dex, se usan los filtros *ElfStringFilter* (idénticos a los *StringFilter*) para buscar string relevantes. Los modelos *ElfStringFilter* y *StringFilter* usan los mismos campos, pero se diferencian en la interacción con archivos dex o elf.

Cabe destacar que los archivos elf no son aceptados como archivo de entrada actualmente, pero están a medio camino para serlo. De hecho, el modelo de datos está diseñado para tal efecto. Ésto se hizo así, dado que puede ser útil en el futuro, poder alimentar el sistema con archivos elf y de esta forma agregar los exploits conocidos, los cuales fueron encontrados simplemente como binario y no dentro de una aplicación. El objetivo es marcar estos elf como maliciosos y anticiparse a su aparición en un malware. Más adelante se detalla este proceso y cómo hacer que sean aceptados como archivo de entrada.

4.2.5.14. ReceiversPicker

Sirve para analizar apk y provee información no transversal. Este *InformationPicker* se encarga de recolectar los Broadcast Receivers definidos en el Android Manifest. Luego almacena su nombre en el modelo *Receiver*, su prioridad en el modelo *ReceiverPriority* y se vincula con los Intents asociados. Estos dos parámetros se guardan en modelos distintos debido a la estructura que puede tener el Android Manifest (ejemplo en el código 4.3). Esta prioridad corresponde al orden en el que serán llamados los distintos Broadcast Receivers, que responden a un mismo Intent, en caso de que éste sea enviado en un Broadcast. Se representa por un número entero, generalmente en $[-1000, 1000]$.

Los Receivers son los módulos encargados de reaccionar a los Broadcast pudiendo activarse, por ejemplo, cuando llega un SMS o cuando el sistema terminó de prenderse. Además, como el sistema tiene prioridad 0 cualquier Receiver con mayor prioridad se activará primero, interceptando los eventos. Por ejemplo, en el caso de la recepción de un SMS, el sistema envía un Broadcast avisando que llegó. Luego, se obtienen todos los Receiver que responden a este Intent (SMS_RECEIVED) y se ejecutan por orden de prioridad. El sistema tiene prioridad 0, por lo que si un malware instala uno con prioridad mayor, éste sera ejecutado primero y puede cancelar el Broadcast. De esta forma el usuario no será avisado de la recepción del SMS. Es por eso que es importante para el análisis conocer los Receivers que tiene una aplicación.

Código 4.3: Ejemplo de AndroidManifest.xml

```
<manifest>
    ...
  <application>
    <receiver android:name="receiver_name1">
      <intent-filter android:priority="50">
        <action android:name="intent_name1"></action>
        <action android:name="intent_name2"></action>
      </intent-filter>

      <intent-filter android:priority="5">
        <action android:name="intent_name1"></action>
        <action android:name="intent_name2"></action>
      </intent-filter>
    </receiver>
    ...
  </application>
</manifest>
```

4.2.5.15. ServicesPicker

Sirve para analizar apk y provee información no transversal. *ServicesPicker* se encarga de extraer los Services definidos en el Android Manifest y guardar su nombre en el modelo *Service*. Es importante pues los Services son procesos que se ejecutan en segundo plano, sin que el usuario se entere.

4.2.5.16. ActivitiesPicker

Sirve para analizar apk y provee información no transversal. *ActivitiesPicker* se encarga de extraer las Activities definidas en el Android Manifest y guardar su nombre en el modelo *Activity*. Además, se asocia a los Intents que pueden iniciar esta Activity. A priori, las Activities no proveen mucha información que ayude a clasificar malware, pero dan una visión general de la aplicación para un análisis manual. Además, sus Intents sí pueden proveer pistas interesantes.

4.2.5.17. ProvidersPicker

Sirve para analizar apk y provee información no transversal. *ProvidersPicker* se encarga de extraer los Content Providers definidos en el Android Manifest y guardar su nombre en el modelo *Provider*. Los Content Provider sirven para compartir información con el resto de las aplicaciones. Dado que aun no se ha encontrado malware que abuse de ellos, no son tan útiles para clasificar. Sin embargo, al igual que las Activities, proveen una visión general de la aplicación.

4.2.6. El sistema de rating: Flag System

El sistema de rating es el módulo encargado de calificar las aplicaciones y darles un puntaje según su riesgo. Luego de que las aplicaciones son procesadas con el módulo Analizador, éstas pasan por el módulo de rating. El puntaje otorgado se basa en las características de la aplicación según sean más o menos sospechosas de ser características de malware. Es así como se definen los "Flags". Los flags son una asociación entre una o más propiedades que puede tener una aplicación y el nivel de riesgo de que pueda ser malware. Por ejemplo, un flag puede ser del tipo: "Si tiene permisos para enviar SMS sumar 4 puntos." o "Si se llama a la función para instalar otro apk sumar 3 puntos.". El puntaje es un numero entero en una escala arbitraria, mientras mayor es el puntaje, mayor es la probabilidad de que esa aplicación sea malware.

Los flags se definen en base a la información recolectada por el Analizador. Luego se elige un "Set de Flags", que puede ser uno o más flags, que será usado para obtener el puntaje. Finalmente, se verifican todos los flags en la aplicación y se obtiene el puntaje final. Gracias a que la información recolectada por los *InformationPickers* queda almacenada en una forma estándar y práctica, se puede definir un "Framework" para definir flags fácilmente.

Existen tres tipos de flag genéricos que abarcan la mayor parte del universo de posibles flags. La ventaja de ellos es que se pueden definir y administrar fácilmente mediante la interfaz web, sin la necesidad de escribir nuevo código o conocer completamente el sistema.

A continuación se explica el framework de creación de flags y luego se detalla como éstos se usan para calificar aplicaciones.

4.2.6.1. Framework de creación de flags

Se diseñó una estructura fácil para crear nuevos flags y flexible frente a nuevas características que puedan aparecer. Tan fácil como elegir una característica y asignarle un puntaje parcial. Ésto es posible y funciona bien gracias a que los *InformationPickers* se diseñaron para guardar cada característica como un objeto distinto (una fila en la base de datos).

En la figura 4.8 se muestra el diagrama de clases del Framework de creación de flags. En la 4.9 el modelo de datos asociado y a continuación se detalla su funcionamiento.

Todos los flags están diseñados para que se puedan guardar en la base de datos (modelo y tabla del mismo nombre que la clase) y extraer cuando sean necesarios. De esta manera, a la hora de usarlos para calificar, basta con extraer los flags necesarios, cargarlos en las clases correspondientes y utilizarlos. Independiente de la clase a la que pertenecen, cada clase sabe como cargar los flags desde la base de datos y usarlos.

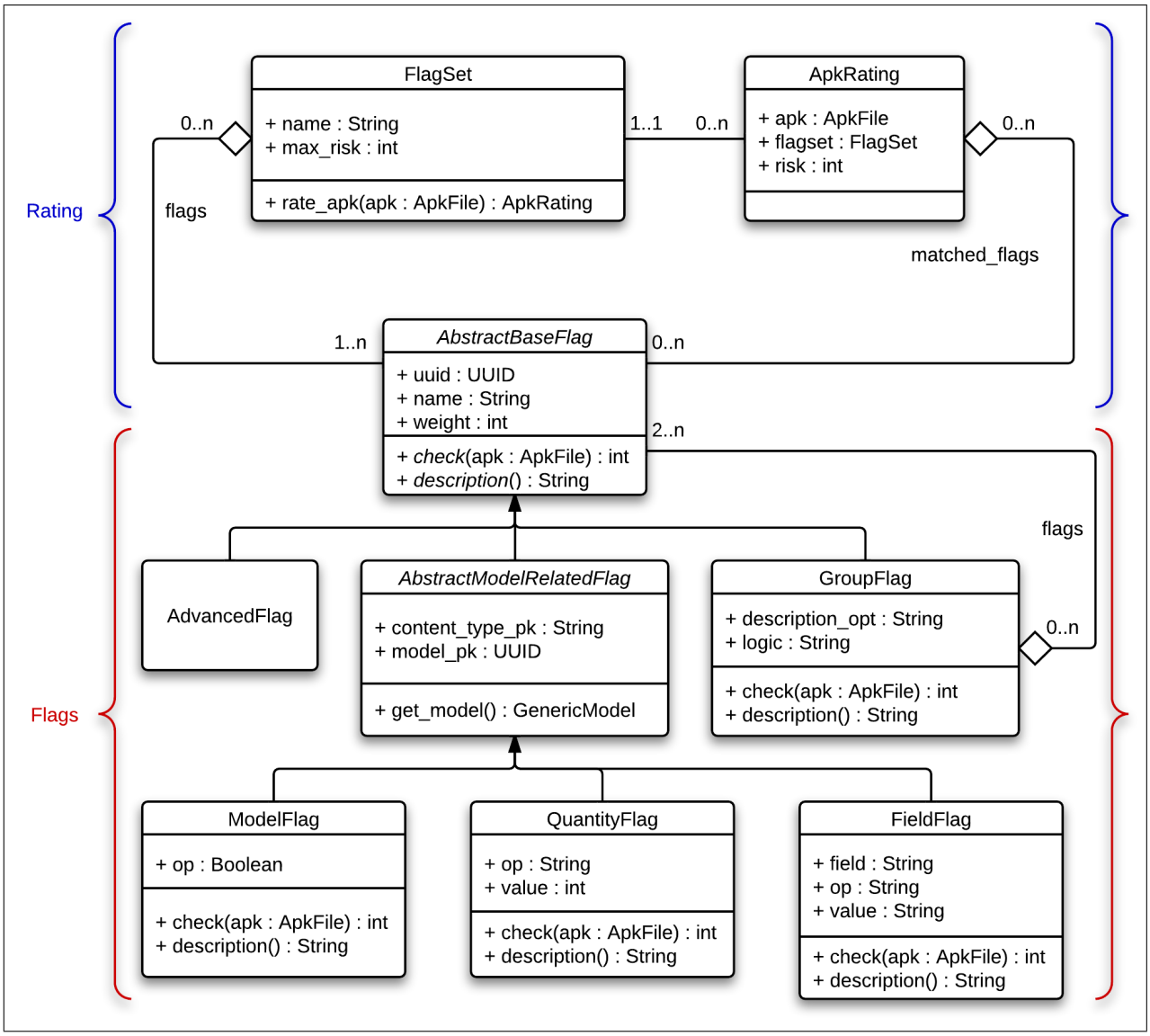


Figura 4.8: Diagrama de Clases simplificado del Framework de creación de flags.

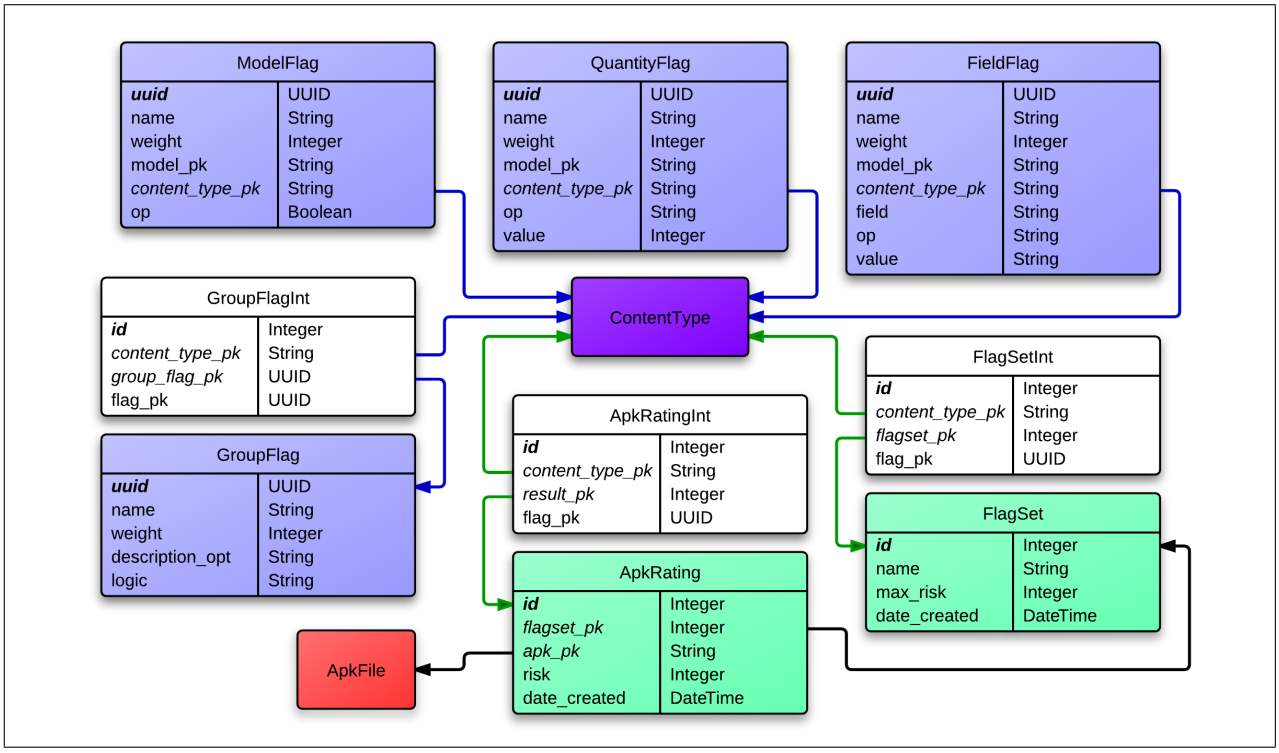


Figura 4.9: Diagrama del Modelo de Datos del Framework de creación de flags.

AbstractBaseFlag Ésta es la clase abstracta base para definir los flags y tiene la interfaz común a todos ellos. Posee tres campos: *uuid*, *name* y *weight*; y dos funciones abstractas, que las subclases deben implementar: *check* y *description*.

1. *uuid*: es un identificador único para cada flag (**U**niversally **U**nique **Id**entifier). De esta forma todos los flags pueden identificarse en la base de datos, aunque estén en tablas distintas por ser de clases distintas.

2. *name*: es un nombre dado al flag. Su única función es poder diferenciarlos fácilmente en la interfaz gráfica.

3. *weight*: es el peso del flag. Éste es el valor que se agrega al puntaje en caso de que la característica definida por el flag se encuentre en la aplicación. Es un número entero positivo, en una escala arbitraria.

1. *check(apk)*: es la función encargada de verificar el flag en el apk. Recibe la muestra(*ApkFile*) y retorna un número positivo (usualmente *weight*) si es que está la característica y cero en caso contrario. Éste es el lugar donde se realiza todo el trabajo. Las subclases deben implementar esta función para ser usadas para calificar.

2. *description()*: es una función que describe el flag. Similar a la función *toString()* de Java, sirve para mejorar la usabilidad en la interfaz gráfica, complementando el nombre del flag.

GroupFlag Esta clase sirve para crear un flag que sea combinación de otros. Usando el patrón de diseño Composite, con esta clase se puede hacer casi cualquier combinación lógica entre dos o más flags, por ejemplo: “Flag1 y Flag2”, “Flag1 o Flag2 o Flag3” o “Flag1 y Flag2, o Flag3”. Cabe destacar que debido a su estructura se pueden anidar innumerables flags, pues un *GroupFlag* puede ser unión de más *GroupFlags*.

Además de los parámetros heredados de *AbstractBaseFlag*, la clase *GroupFlag* tiene tres campos extra:

1. *description_opt*: es una descripción opcional del flag.
2. *logic*: es un string que indica la relación lógica entre los flags agrupados. Actualmente puede ser “or” o “and”, que representan la relación lógica correspondiente, pero es flexible para agregar más relaciones (como “xor”).
3. *flags*: es la lista de flags agrupados. Todos son instancias de *AbstractBaseFlag* y tienen la misma interfaz, por lo que pueden ser tratados indistintamente.

Tal como indica el patrón Composite, la función *check(apk)* llama recursivamente a la función *check(apk)* de cada flag agrupado y aplica la relación lógica sobre el resultado. En caso de que se cumpla, retorna el valor de *weight* del *GroupFlag*, sin tomar en cuenta los pesos de los flags agrupados.

La función *description()* retorna el valor de *description_opt* en caso de estar definida y en caso contrario construye una descripción según el resto de los parámetros. Por ejemplo: “Any of name_flag_1, name_flag_2” o “All of name_flag_1, name_flag_2”.

AbstractModelRelatedFlag Esta clase también es abstracta y sirve de modelo intermedio para agregar una funcionalidad común a los distintos tipos de flags. Esta funcionalidad es la interfaz necesaria para comunicarlos con las clases asociadas a los *InformationPickers* y así verificar los flags.

Esta clase tiene dos campos: *content_type_pk*, el tipo de modelo de *InformationPicker* y *model_pk*, la llave primaria de este modelo. El framework Django tiene un módulo llamado *The ContentTypes Framework* que permite (gracias a *content_type_pk* y *model_pk*) trabajar con distintos modelos indistintamente a través de una interfaz genérica. Ésto da la opción de asociar un flag a un modelo de *InformationPicker* sin saber necesariamente cuál es, lo que genera una flexibilidad increíble.

Además, una parte de este framework consta de una tabla en la base de datos, *ContentType*, que guarda identificadores para todos los modelos del sistema. De esta forma, se pueden guardar las relaciones entre una subclase de *AbstractModelRelatedFlag* y cualquier modelo de *InformationPicker* o entre un *GroupFlag* y otros flags, independiente del modelo que sean. Cabe destacar que para el caso de relaciones de 1 a N, basta con crear una llave foránea hacia el *ContentType* correspondiente, mientras que en el caso de las relaciones de N a N, hay que crear una tabla intermedia para asociar los dos modelos (*ContentType* y *GroupFlag*). Gracias a esta estructura, todos los parámetros necesarios para definir un flag se pueden almacenar en la base de datos y administrarlos fácilmente.

ModelFlag Esta clase es la primera de los tres modelos de flag genéricos asociados a un *InformationPicker*. Sirve para definir flags del tipo: "El apk tiene la característica A" o "El apk no tiene la característica B". Donde la "característica" puede ser cualquiera de las recolectadas por los *InformationPickers*. Por ejemplo, el *PermissionsPicker* guarda cada Permiso requerido por la aplicación, como *SEND_SMS* y el *IntentsPicker* guarda los Intents, como *ACTION_BOOT_COMPLETED*. Luego, se pueden definir los flags: "El apk tiene el permiso SEND_SMS" o "El apk tiene el Intent ACTION_BOOT_COMPLETED".

El campo *op* es un booleano que sirve para definir el flag como "tener" o "no tener" la característica definida en *AbstractModelRelatedFlag*. La función *description()* construye un string en base a los parámetros de *ModelFlag* que lo definen.

La función *check(apk)* se encarga de verificar que el *ApkFile* tenga o no tenga la característica en cuestión. Para ésto, todos los modelos de *InformationPicker* para los cuales tenga sentido la pregunta "¿Un apk puede tener esta característica?" deben implementar la función *belong_to(apk:ApkFile)*, una interfaz común que recibe el apk y retorna True o False según la característica esté o no. Esta función, como está implementada en los modelos de *InformationPicker*, sabe como verificar que un apk tenga esa característica. En la figura 4.10 se ejemplifica ésto. InformationPickerModel puede ser cualquier modelo asociado a un *InformationPicker*, como *Permission* o *Intent* (ver modelo de datos en las figuras 4.6 y 4.7).

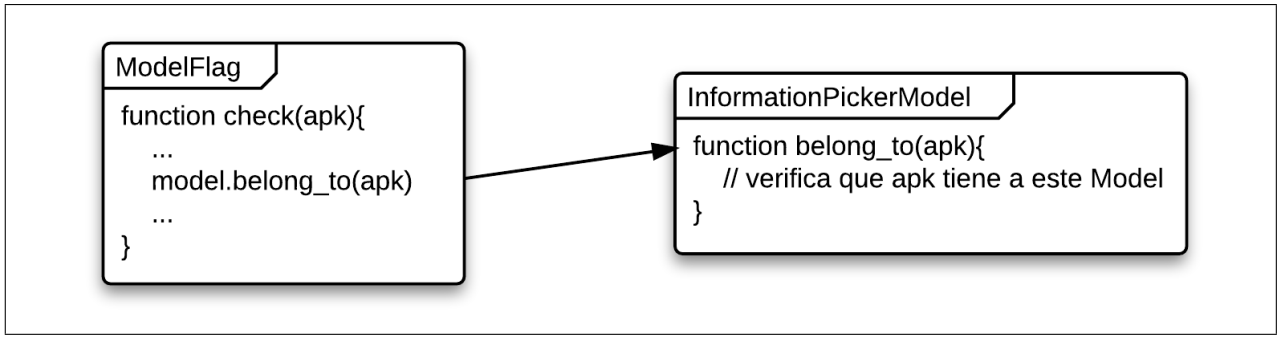


Figura 4.10: Ejemplo de llamada a *belong_to(apk)* en *ModelFlag*.

Por otro lado, la pregunta "¿Un apk puede tener esta característica?" tiene sentido para todos los *InformationPicker* salvo para los que guardan su información como arreglo de Python (*StringsPicker*, *FunctionsPicker*, *FieldAccessPicker* y los strings de elf); dado que no guardan una característica por fila/objeto. Para estos casos se usan los filtros: *StringFilter*, *RelevantFunctionFilter*, *FieldAccessFilter* y *ElfStringFilter*. Luego, el flag pasa a ser: "El apk tiene (al menos) una característica que calza con el filtro X" y el flag se asocia con el filtro en vez de la característica en sí.

QuantityFlag Esta clase también corresponde a un flag genérico y sirve para definir flags del tipo: "El apk tiene X cantidad de la característica A" o "El apk tiene más de 8 características B". Es decir, sirve para evaluar la cantidad de una cierta característica, recolectada por los *InformationPicker*, que tiene un apk. Por ejemplo, "El apk tiene 8 o más permisos", "El apk tiene más de 0 *SuspiciousFiles*" o "El apk tiene más de 1 package".

Además de los parámetros heredados, *QuantityFlag* posee dos campos para definir el flag:

1. *op*: es un string que identifica la operación a realizar con la cantidad de características. Actualmente se aceptan los strings: eq(igual), neq(distinto), gt(más grande que), lt(más chico que), gte(más grande o igual a) y lte(más chico o igual a).
2. *value*: es un entero positivo con el cual se compara la cantidad evaluada. Por ejemplo, "más de 5", "igual o menor que 3" o "exactamente 6".

Similar a *ModelFlag*, la función *check(apk)* de *QuantityFlag* revisa cuántas características de cierto tipo tiene la muestra. Para ésto, todos los modelos de *InformationPicker* deben implementar la función *quantity_in(apk:ApkFile)*, una interfaz común que recibe el apk y retorna la cantidad de esa característica que tiene el apk. En la figura 4.11 se gráfica ésto.

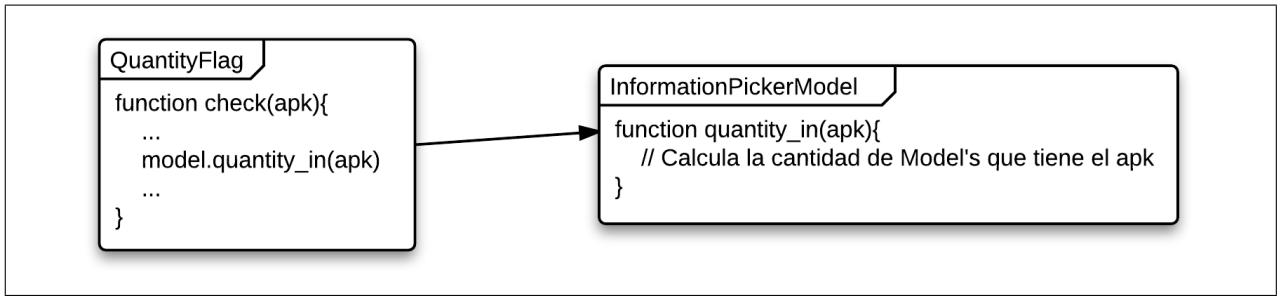


Figura 4.11: Ejemplo de llamada a *quantity_in(apk)* en *QuantityFlag*.

QuantityFlag tiene una variación adicional. Si en los campos de *AbstractModelRelatedFlag*, sólo se define *content_type_pk*, entonces el flag verifica la cantidad de ese tipo de modelo. Por ejemplo, si *content_type_pk* corresponde a *Permission*, entonces el flag verificará la cantidad de permisos del apk. Por otro lado, si además se define *model_pk* entonces el flag verifica la cantidad de ese modelo en particular. Por ejemplo, si *content_type_pk=Intent* y *model_pk=ACTION_BOOT_COMPLETED*, el flag verifica la cantidad de ese intent (ACTION_BOOT_COMPLETED) que tenga el apk. La cantidad de un modelo específico usualmente será 1 o 0 y no tiene mucha utilidad, salvo en el caso de usar un filtro, el cual puede abarcar más de una característica (como todas las funciones de un package) y en ese caso si tiene sentido preguntar por la cantidad.

FieldFlag Esta clase es el último flag genérico y sirve para definir flags del tipo: "El apk tiene una característica del tipo A con un parámetro B". Es decir, sirve para evaluar un parámetro de alguna de las características recolectadas por los *InformationPicker*, que tiene un apk. Por ejemplo, "El apk tiene un Receiver con prioridad mayor que 0", "El apk pesa menos de 70kb" o "El apk tiene un package con nombre que empieza con *SMS*".

En este caso, *FieldFlag* posee tres parámetros a definir:

1. *field*: representa el parámetro o campo del modelo a verificar. Por ejemplo: *priority* de *ReceiverPriority*, *size* de *ApkFile* o *name* de *InnerPackage*.

2. *op*: es un String que representa la operación a realizar con el *field*. Por ejemplo: seq (igual), sneq (diferente), gt (mas grande que), start_ci (Empieza con) o has_ci (Contiene). Fue diseñado para poder agregar nuevas operaciones fácilmente.

3. *value*: es un String que representa el valor con que comparar a *field* usando *op*.

Usando estos parámetros se pueden definir flags eligiendo el campo, la operación y el valor con qué comparar. En este caso, los modelos de *InformationPicker* deben implementar la función *get_field_values_for_apk(apk, field)*, la cual recibe el apk, obtiene todas las características de ese tipo de modelo y retorna una lista de todos los valores que tiene para ese campo. Por ejemplo, la llamada a la función *InnerPackage.get_field_values_for_apk(apk, "name")* retorna una lista con el nombre de todos los packages de ese apk. Luego, la función *check(apk)* llama a esta función, utiliza el resultado para comparar con el valor definido y así determinar si se cumple o no la condición del flag.

Como los parámetros de los modelos y las operaciones disponibles para *FieldFlag* no necesariamente son conocidas por el usuario, la interfaz web se encarga de facilitar las opciones disponibles. De esta forma, se pueden crear *FieldFlags* vía la interfaz web sin conocer completamente el sistema, facilitando la creación y administración de éstas.

Cabe destacar que este tipo de flag no tiene sentido para varios modelos. Por ejemplo, preguntar por un parámetro de los Strings, pues es sólo un String. Lo mismo sucede con las funciones y los accesos a campos. Para estos casos se usan los filtros y los otros tipos de flag. Por otra parte, algunas combinaciones de flag dan el mismo resultado. Por ejemplo, un *ModelFlag* que verifica un cierto package(de nombre X), es lo mismo que un *FieldFlag* que verifica un package con nombre exactamente X. Si bien este flag no agrega tantos casos al universo de posibles flags, agrega algunos interesantes como para ser un tipo de flag genérico.

AdvancedFlag Si bien, usando los 4 tipos de flag: *ModelFlag*, *QuantityFlag*, *FieldFlag* y *GroupFlag*, se abarca gran parte del universo de posibles flags, existen algunos casos que no están cubiertos. Un caso importante son las excepciones, por ejemplo, "Si el apk tiene URLs, pero no son de una lista de URLs confiables(google, facebook) sumar 3 puntos". Se podría crear un *ModelFlag* y hacer una expresión regular, pero la lista de URL's confiables no sería manejable fácilmente. Es para estos casos que se diseñó un cuarto tipo de flag, los *AdvancedFlag*. La idea consiste en dejar la opción de poder definir un flag con cualquier condición. La forma de realizar ésto es realizar un framework para que el usuario pueda implementar una parte del flag en código Python y luego usarlo como un flag común. Este flag está diseñado, pero no está actualmente implementado.

Para crear este tipo de flag, el usuario tiene que diseñar un objeto Python que implemente la interfaz *AdvancedFlagInterface*, la cual tiene una función: *check(apk)*. Ésta es similar a las anteriores: recibe un apk y retorna un booleano que dice si se cumple la condición o no. En la figura 4.12 se muestra el diagrama de clases para la implementación de *AdvancedFlag*.

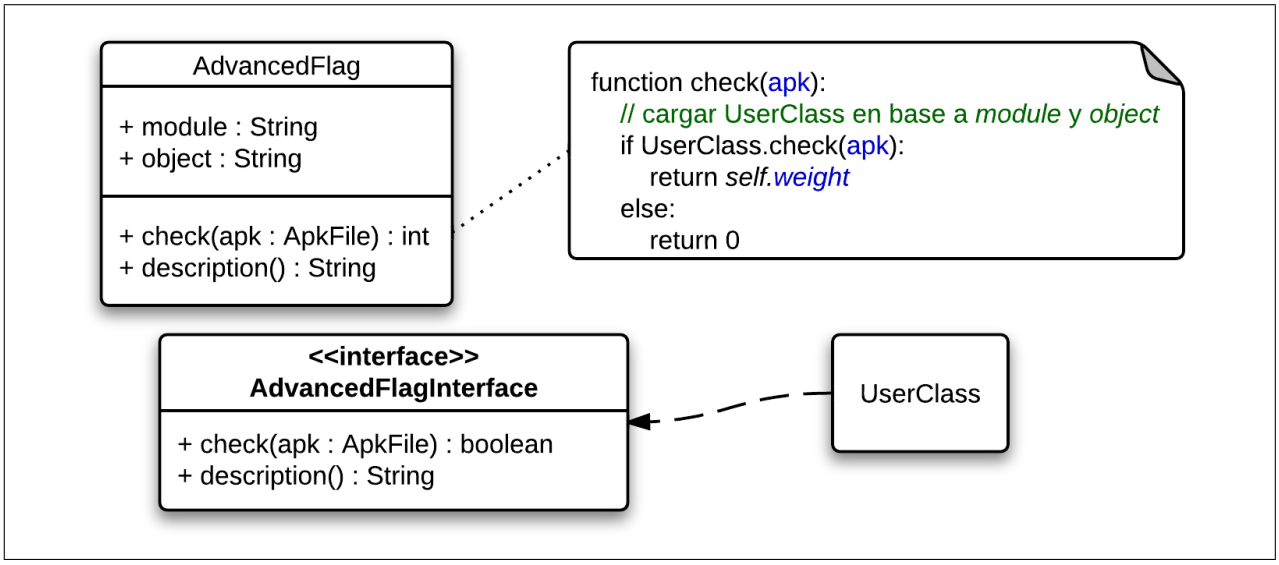


Figura 4.12: Diagrama de Clases para la implementación de *AdvancedFlag*.

A diferencia de los flags anteriores, que se pueden crear fácilmente con la interfaz web, para este caso, el usuario tendría que programar la condición del *AdvancedFlag*. En el diagrama, *UserClass* es la clase que tiene que implementar el usuario. La ventaja es que, a cambio de conocer la interfaz de acceso a la información del apk, el programador accede a toda su información y puede establecer la condición que desee. Luego, sólo tiene que configurar los parámetros de *AdvancedFlag* para que acceda a su código y funcione en conjunto con los otros tipos de flags.

Los parámetros necesarios para definir este tipo de flag, y que se pueden guardar en la base de datos, serían dos:

1. *module*: el módulo donde se encuentra la clase que implementa la interfaz *AdvancedFlagInterface*.
2. *object*: el nombre de la clase (o objeto) que implementa esta interfaz.

De esta forma, *AdvancedFlag.check(apk)* puede cargar la clase implementada por el usuario y verificar la condición. Por otra parte, la función *description()* solo llama a la función *description()* implementada por *UserClass*.

Gracias a este tipo de flag, el universo de posibles flag está completo y se puede definir cualquier condición para verificar en una muestra, siempre y cuando algún *InformationPicker* obtenga la información necesaria.

FlagSet Es la clase que se encarga de realizar el chequeo de los flags sobre una aplicación. Está asociada con un conjunto de flags utilizados para realizar el rating. Al igual que los flags, los *FlagSets* están parametrizados y guardados en la base de datos. Ésto permite modificar rápidamente el conjunto de flag para calificar y tener varios conjuntos de flags para distintos tipos o versiones de calificaciones.

La clase tiene tres parámetros y una función:

1. *flags*: la lista de flags utilizados para calificar.
2. *name*: un String para identificar el *FlagSet* en la interfaz gráfica.
3. *max_risk*: un entero que representa el valor máximo que puede obtener una muestra si es que calzan todos los flags. El valor es redundante (pues se puede obtener en base a la lista de flags) y se almacena para evitar calcularlo reiteradamente.

La única función, *rate_apk(apk)*, recibe un *ApkFile* y recorre todos los flags verificando si se cumple o no su condición. Luego calcula el riesgo en base a los flags que calzaron y retorna el resultado como *ApkRating*, una estructura ad-hoc. Gracias a que todos los flags tienen la misma interfaz y al "ContentTypes Framework" se puede trabajar con ellos indistintamente y el sistema se mantiene flexible.

ApkRating Es la estructura ad-hoc para guardar el puntaje del rating de un apk. Tiene cuatro campos:

1. *apk*: la asociación al apk que se evaluó.
2. *flagset*: la asociación al *FlagSet* que se uso para evaluar.
3. *risk*: el puntaje final obtenido.
4. *matched_flags*: una lista de los flags que calzaron.

Este objeto guarda la información necesaria y dado que se encuentra en una sola tabla, se puede generar el ranking rápidamente. Al igual que *GroupFlag* y *FlagSet*, *ApkRating* también usa el modelo *ContentType* para guardar la relación de N a N con los distintos tipos de flags.

4.2.6.2. Calificación de aplicaciones

Los flags son el mecanismo que proporciona la "inteligencia" del sistema. Es en base a ellos que se determina si una muestra es potencialmente malware o no. Es por ésto que uno de los objetivos, es que los flags se puedan parametrizar y modificar fácilmente (un panel en la interfaz web). De esta forma, se puede regular y adaptar rápidamente "el cerebro" de Samsaraa y ajustarlo para aumentar la precisión de detección. Ésto es factible dado que cuatro de los cinco tipos de flags (todos menos *AdvancedFlag*) se administran completamente vía la interfaz web, sin tocar el código. Incluso, también se puede ajustar el peso de los *AdvancedFlags*.

Como se ha mencionado antes, la interfaz web permite crear, modificar y eliminar los distintos tipos de flags. Debido a que el usuario no necesariamente sabe como está organizada la información, la interfaz facilita las opciones para cada parámetro. Luego de definir todos los flags necesarios,

se procede a crear un *FlagSet* y se eligen y califican las muestras. Este proceso se realiza paralelamente, un *ApkFile* por proceso, de forma de aumentar la velocidad de calificación.

El sistema de flags tiene algunas observaciones que son importantes de remarcar. En particular, dado que los flags están asociados a una característica recolectada por los *InformationPickers*, ésta debe estar previamente almacenada en la base de datos para poder ser utilizada en un flag. Ésto implica que, o se encontró en alguna muestra anterior o se puede agregar manualmente a la base de datos. Por ejemplo, si el usuario quiere definir un flag que evalúe si la aplicación tiene o no un permiso especial, el sistema tiene que haber encontrado una muestra que lo tenga. En caso contrario, el usuario puede agregar la entrada en la base de datos (vía la interfaz web) para luego crear el flag. Un segundo ejemplo es si el usuario quiere crear un flag que identifique un Exploit (archivo elf). En este caso, el sistema tiene que haber analizado este archivo de antemano. Ésta es una de las razones por lo cual es interesante aceptar los archivos elf (u otros archivos relevantes) como archivos de entrada, de manera que no sea necesario encontrar una muestra que lo tenga y sea suficiente con alimentar el sistema con estos archivos.

Otro punto que hay que destacar es el uso de los filtros. Un filtro, gracias a su definición, puede abarcar más de una característica, por ejemplo: "Todas las funciones del package android.net" o "Todos los strings que empiecen con http://". De hecho, sólo se puede crear un flag que abarque más de una característica usando un filtro, y solo para funciones, strings o accesos de campo; o un *AdvancedFlag*. Por otra parte, los filtros de funciones también se crearon para poder calcular y visualizar las estadísticas de su uso. En efecto, dado que una aplicación contiene muchas llamadas a funciones, es imposible mostrar el uso que hacen de todas ellas. La gracia de los *RelevantFunctionFilters* es poder definir algunas funciones más importantes o relevantes para el análisis (como las funciones de telefonía o de internet) y luego comparar su uso entre todas las aplicaciones. Ésto reduce notablemente la cantidad de funciones a comparar y la visualización se vuelve manejable. De esta forma, se puede ver fácilmente cuáles son las funciones más usadas o comparar su uso entre distintos grupos de aplicaciones (usando el *Source* como agrupación).

Por esta razón, se inventaron los filtros de funciones "generic" y "no generic". Ambos tipos de filtros se pueden usar en un flag, sin embargo, los filtros "generic" están destinados para recolectar estadísticas del uso de funciones. La idea es que sean filtros que abarquen más de una función (como todas las funciones de un mismo package), pues éstos son los que se usarán para clasificar y almacenar las *RelevantFunction* (ver página 34). Y por otro lado, los filtros "no generic" son principalmente para definir flags, y especifican una función en particular o un conjunto más acotado de funciones.

4.2.6.3. Definición de flags

Tal como se menciono anteriormente, el sistema de flags es la inteligencia de Samsaraa y tiene el objetivo de poder distinguir una aplicación limpia de un malware, o de al menos encontrar las aplicaciones más riesgosas. Es por ésto que elegir los flags adecuados, es una parte importante para que funcione el sistema. Los flags definidos actualmente fueron creados en base a la publicación "Dissecting Android Malware: Characterization and Evolution" (ver sección 2.2.4 y [38]), la publicación "Reducing the window of opportunity for Android malware" (ver sección 2.2.3 y [2]), en la experiencia adquirida de los investigadores de McAfee y en las estadísticas recolectadas por

Samsaraa. Algunos ejemplos se pueden ver en la tabla 4.3.

Nombre	Descripción	Tipo
API Call: Send SMS/MMS	Posee llamadas a las funciones para enviar SMS o MMS.	GroupFlag
Known Exploits	Posee archivos elf maliciosos conocidos (exploit).	GroupFlag
Activation: USB	Se activa frente a los eventos de conexión USB.	GroupFlag
Activation: Boot Completed	Se activa frente al evento “Reinicio completo”.	ModelFlag
API Call: get installed packages	Obtiene la lista de Aplicaciones instaladas.	ModelFlag
API Call: abortBroadcast	Llama a la función: abortBroadcast().	ModelFlag
Small file size	El apk pesa menos de 70KB.	FieldFlag
To many Permissions	Pide más de 8 permisos.	QuantityFlag
Presence of Receivers	El apk posee Broadcast Receivers.	QuantityFlag

Tabla 4.3: Algunos ejemplos de los flags definidos en el sistema.

Los pesos de cada flag se definieron usando la estrategia explicada por Apvrille y Strazzere [2]. La idea consiste en, una vez definidos los flags, analizar y calificar dos conjuntos de aplicaciones: uno de aplicaciones limpias y el otro de malware. Luego, para cada flag, se compara la cantidad de aplicaciones limpias que calzaron versus la cantidad de malware y se usa esta diferencia (en porcentaje) para elegir el peso. La idea es buscar grandes diferencias entre los dos grupos para decir que un flag caracteriza más a uno que al otro. Los pesos se definieron usando los valores de la tabla 4.4.

Diferencia entre aplicaciones limpias y malware	Peso asignado
50 % o superior	5
40 % a 49,9 %	4
30 % a 39,9 %	3
20 % a 29,9 %	2
10 % a 19,9 %	1
menor a 10 %	0

Tabla 4.4: Reglas para definir el peso de un flag.

En la tabla 4.5 se pueden ver los pesos asociados a algunos flags. Por ejemplo, el flag: “API Call: Send SMS/MMS” calzo en un 8,1 % de aplicaciones limpias y en un 46,6 % de malware, con una diferencia de 38,5 %, por lo que su peso es de 3.

Nombre	Cantidad en aplicaciones limpias	Cantidad en malware	Peso
API Call: Send SMS/MMS	8,1 %	46,6 %	3
Known Exploits	0 %	8,3 %	3
API Call: get installed packages	9,8 %	39,8 %	3
Activation: Boot Completed	28,3 %	51,3 %	2
Activation: SMS/MMS Received	4,0 %	28,3 %	2
High Priority Receiver	4,0 %	21,5 %	1
API Call: abortBroadcast	8,1 %	22,7 %	1

Tabla 4.5: Pesos asociados a los flags.

4.2.7. La interfaz de usuario

Para interactuar con el sistema existen dos interfaces: por consola y por una interfaz web. La consola solo sirve para analizar muestras y desplegar el resultado del análisis. El resultado no es muy amigable pero es más práctico para hacer debugging. Por otro lado, la interfaz web permite utilizar todas las funcionalidades del sistema y acceder a la base de datos más fácilmente.

Esta interfaz es una aplicación web que utiliza Django, un framework MVC de python (ver página 15). Corre sobre un servidor web y se accede a través de un navegador. Permite realizar las siguientes funciones:

- Analizar muestras,
- Ver los resultados del análisis,
- Visualizar y comparar las estadísticas calculadas por el sistema,
- Ver el ranking de las aplicaciones,
- Administrar el sistema.

4.2.7.1. Análisis

Actualmente el sistema tiene tres modalidades para analizar una muestra: vía consola, vía interfaz web (un sólo archivo) y vía interfaz web (muchos archivos).

La primera opción consiste en ejecutar un comando por consola que realiza el análisis. Recibe como parámetro el archivo para analizar y el resultado lo almacena en la base de datos y lo muestra en pantalla. Se utiliza principalmente para hacer debugging.

La segunda alternativa es utilizar la interfaz web. En este caso, el usuario sube el archivo al servidor a través de un formulario; el sistema guarda y analiza la muestra y luego, si no hubo errores en el análisis, redirige al usuario al informe de la muestra.

Finalmente, la tercera alternativa también es a través de la interfaz web, pero permite analizar varias muestras simultáneamente. Dado que uno de los requerimientos es analizar grandes cantidades de aplicaciones en poco tiempo, se desarrolló un módulo que trabaja en procesos paralelos. La interfaz para utilizar este módulo es bastante simple: recibe una lista de las muestras y retorna el resultado del análisis para cada una de ellas. Actualmente, la interfaz web permite especificar una carpeta en el servidor. Luego el sistema recorre esa carpeta buscando muestras y las analiza. El sistema utiliza el módulo de python Celery (ver figura 4.2 página 26), que administra una cola y un pool de procesos. De esta forma, primero se encolan todas las muestras encontradas y luego varios procesos van desencolando y analizando las muestras una por una, paralelamente. Finalmente, a medida que las muestras se terminan de analizar, va apareciendo el resultado en la interfaz web. Este resultado indica si el análisis terminó correctamente o no y despliega un link al informe de la muestra.

4.2.7.2. Informes de análisis

La interfaz web permite ver los informes del análisis de apk, dex y elf. En ellos se muestra toda la información recolectada durante el análisis y las relaciones entre las distintas muestras.

En el apéndice B se muestra un ejemplo de informe para apk. El informe es bastante intuitivo y muestra todas las características antes mencionadas. En particular "Inner Apks (1)" muestra un link al reporte de los apk que están adentro de la muestra, "Parent Apks (0)" muestra los apk que contienen a éste y "FileTypes" muestra los tipos de archivos presentes en el apk y su cantidad. Además, tiene vínculos a los informes de archivos elf presentes en la muestra ("ELF Files (2)") y al informe de los archivos dex: "More Details" para el dex principal (*classes.dex*) y "Other Dex Files (0)" para el resto. Los números entre paréntesis al lado de varios de los títulos, corresponden a la cantidad presente de esos elementos. Por ejemplo: "Permissions (12)" indica que el apk requiere 12 permisos para funcionar.

Por otro lado, el informe de análisis de un archivo dex está separado en cuatro partes, de manera de poder visualizar fácilmente la gran cantidad de datos que son recolectados. Estas cuatro partes corresponden a cada uno de los *InformationPickers* de dex: Packages, Strings, Funciones y Accesos a campos. En el apéndice C (figura C.1) se muestra un ejemplo del informe de Packages. En éste también se pueden ver los distintos apk que comparten este mismo dex (sección "Parent Apks (1)"). Para los otros tres casos existe un segundo menú que permite elegir un filtro y así restringir la visualización. Por ejemplo, al elegir el filtro de funciones "Telephony", sólo se muestran las funciones de telefonía. En el apéndice C se muestra un ejemplo de informe para los strings (figura C.3), funciones (figura C.4), funciones de telefonía (figura C.5) y accesos de campo (figura C.2).

Finalmente, el informe de elf muestra todos los strings ASCII encontrados en el binario y todos los apk en los cuales se encontró este elf. Al igual que para los strings de dex, en este informe hay un menú de filtros para visualizar sólo una selección de strings. En el apéndice D se muestra un ejemplo de informe de elf.

Cabe destacar que los tres informes muestran también las características del archivo, como el tamaño, nombre, fecha de análisis y hashes calculados.

4.2.7.3. Estadísticas

A través de la interfaz web, también se pueden observar las estadísticas recolectadas gracias al análisis de información transversal. En particular, se pueden ver los permisos, Intents, Features, RelevantFunctions, Packages, librerías y tipos de archivo (FileTypes). Sobre cada una de ellas, el sistema identifica qué características son usadas, en cuántas y qué muestras, facilitando el porcentaje sobre el total y la lista de muestras que la usan. Además, se agrega la jerarquía entre los distintos apk, es decir, cuántas muestras tienen apk internos y cuántas y cuáles contienen el mismo apk. También se muestra el uso común de dex y elf, es decir, cuáles y cuántos apk comparten el mismo dex y/o elf. Sobre todas estas estadísticas, la interfaz permite descargar la información en formato CSV. De esta manera se pueden utilizar softwares externos para realizar estudios estadísticos más detallados.

Por otro lado, el sistema también guarda estadísticas sobre sí mismo. Es decir, cuántas muestras hay almacenadas, cuánto se demora en analizar un apk y un dex y cuál es el tamaño promedio (en kb) entre apk y dex. Sobre estas estadísticas, la interfaz web también despliega gráficos para facilitar su visualización.

Finalmente, gracias a que cada muestra está asociada a un *Source*, se pueden crear grupos de muestras y comparar las características entre los distintos grupos. Por ejemplo, el uso de permisos entre aplicaciones limpias y malware. En efecto, esta funcionalidad fue una de las utilizadas para definir los flags, comparando las características entre aplicaciones limpias y malware y detectando cuáles son más utilizadas por malware. De esta forma, se definieron propiedades particulares de las aplicaciones que permiten calificarlas con mayor seguridad como maliciosas

4.2.7.4. Ranking

El objetivo principal del sistema es poder calificar las aplicaciones y hacer un ranking de las más riesgosas. Este ranking está disponible en la interfaz web. En él se puede ver la lista de todas las aplicaciones analizadas, ordenadas por nivel de riesgo. Ésto es lo que permite a los investigadores seleccionar las más sospechosas y luego analizarlas con mayor detalle. Este ranking solo muestra el puntaje final. Sin embargo, permite acceder al detalle de cada una de las aplicaciones, en el que se muestran todos los flags que calzaron y que justifican su riesgo.

También, la interfaz web entrega la opción de ver la eficacia de los flags. En particular, hay un panel para elegir un *FlagSet* y ver cuántas aplicaciones calzan con cada uno de ellos. Además, se pueden comparar los distintos grupos de aplicaciones usando los *Sources*. Ésta fue la funcionalidad utilizada para elegir el peso de cada uno de los flags. En efecto, comparando entre muestras de malware y muestras de aplicaciones limpias, se definió qué flags eran los que más caracterizaban un malware.

Al igual que con el resto de las funcionalidades, es factible observar sólo un conjunto de las aplicaciones seleccionado un *Source* en particular. De esta forma, se puede ver el ranking de un grupo en particular o comparar el ranking entre dos grupos de aplicaciones.

4.2.7.5. Administración

Finalmente, la interfaz web tiene un panel de administración en el cual se pueden crear y modificar todas las variables del sistema. Por administrar se entiende: ver, crear, modificar y eliminar (CRUD) las características. Actualmente el sistema permite modificar las siguientes variables:

- **Los flags**: definir nuevos flags, ver los existentes y eliminar los que no sirven. Aquí también se agrupan en *FlagSets*, pudiendo agregar o quitar flags a un *FlagSet*.
- **Los filtros**: especificar los distintos filtros para funciones, strings y accesos de campo.
- **Características de las muestras**: este panel también da la opción de agregar, modificar y eliminar las distintas características de las muestras. Por ejemplo, agregar un nuevo permiso o un nuevo Intent. Esta funcionalidad sirve sobre todo para poder crear flags con características especiales que aún no se han encontrado en una muestra.

Por otro lado, algunos parámetros del módulo *Analyzer* se pueden modificar en el archivo *samsaraa_config* (ver página 29). En particular, se pueden cambiar los *InformationPickers* que se usarán durante el análisis, y los archivos de entrada aceptados por el sistema.

4.2.8. Extensibilidad

Samsaraa fue diseñado e implementado modular y flexiblemente; de esta forma será más simple de mantener, corregir y mejorar en el tiempo. También se puso especial cuidado en el formato del código y en la elección de los nombres, de manera que sea de fácil comprensión y legibilidad. En particular, el sistema tiene una gran facilidad para extenderse en dos direcciones. La primera consiste en agregar el análisis de una característica o un tipo de análisis, y la segunda consiste en aceptar un nuevo tipo de archivo de entrada.

4.2.8.1. Agregar un *InformationPicker*

Un *InformationPicker* es un componente que se encarga de una pequeña parte del análisis. Una de las formas de extensión consiste en agregar más detalles que analizar con un nuevo *InformationPicker*. Para ésto, hay que realizar los siguientes pasos:

1. Desarrollar el *InformationPicker*, el cual debe heredar de *AbstractInformationPicker*, *AbstractAPKInformationPicker* (si es de apk) o *AbstractDEXInformationPicker* (si es de dex) e implementar las funciones requeridas (ver página 31).
2. Crear las tablas en la base de datos y los modelos correspondientes que se encargarán de almacenar la información recolectada.

3. Para integrar esta información al framework de flags, el *InformationPicker* debe implementar algunas de las interfaces de *ModelFlag*, *QuantityFlag* y/o *FieldFlag* (ver sección 4.2.6.1).

4. Finalmente, debe agregar el *InformationPicker* al archivo de configuración *samsaraa_config* (ver página 29) para que sea utilizado en el análisis.

El objetivo de este tipo de expansión es aumentar la información extraída de un archivo. Por ejemplo, actualmente no se analizan las imágenes contenidas en un apk. Sin embargo, en un futuro podría existir una vulnerabilidad que se aproveche usando imágenes especialmente construidas. En ese caso, lo ideal es agregar un *InformationPicker* para apk que revise las imágenes que contiene y verifique si cumplen o no las condiciones para explotar esta vulnerabilidad. Un caso parecido sería para analizar pdf que exploten alguna vulnerabilidad.

Otra razón para agregar un *InformationPicker* es incorporar un nuevo tipo de análisis. Por ejemplo, se podría crear un mini-analizador que estudie y construya un árbol de ejecución del código y así poder comparar, entre varias aplicaciones, partes del comportamiento, aunque no estén implementadas de la misma forma.

4.2.8.2. Agregar un tipo de archivo de entrada

Un tipo de archivo de entrada es un archivo aceptado por el sistema como muestra para analizar (input del sistema). Actualmente sólo se aceptan los tipos apk y dex. Dado que las aplicaciones de Android están empacadas en archivos apk, es el principal objetivo de análisis. Y por otro lado, dada la experiencia de los investigadores en McAfee, muchas veces reciben muestras que sólo consisten en archivos dex, los cuales generalmente vienen dentro de un apk. De esta forma, el sistema rechazará analizar cualquier otro tipo de archivo (como pdf o jpg).

La segunda forma de extender el sistema es aceptar otro tipo de archivo de entrada. Para realizar ésto, hay que agregar los siguientes componentes:

1. En primer lugar, el desarrollador debe implementar una subclase de *AbstractSample* (ver página 28) que se encargue de la primera parte del análisis, de guardar esa información en la base de datos y de entregar la lista de *InformationPickers*.

2. Debe agregar una tabla a la base de datos para almacenar las características de la muestra: nombre de archivo, tamaño, hashes, fecha y tiempo de análisis, así como cualquier otro dato que se estime conveniente. Esta tabla, al igual que *ApkFile* y *DexFile* debe estar vinculada a *Source* para que las funcionalidades que permiten comparar por grupos funcionen correctamente.

3. Luego debe agregar al archivo *samsaraa_config* (ver página 29) el nuevo tipo de archivo aceptado y la subclase de *AbstractSample* correspondiente.

4. Finalmente, se deben implementar los *InformationPickers* encargados del análisis. Cabe destacar que la subclase de *AbstractSample* debe tener acceso a la lista de *InformationPickers* que se deseen usar en el análisis. Para flexibilizar el sistema, esta lista se puede definir en el

archivo de configuración e indicar a esta subclase que la lea. Actualmente, *ApkSample* y *DexSample* funcionan de esta manera.

El objetivo de este tipo de expansión es mejorar y aumentar la cobertura del análisis del sistema. Por ejemplo, tal como se menciono antes, los archivos elf son binarios para la arquitectura ARM (utilizada en los celulares con Android) que agregan mayor funcionalidad a una aplicación. Sin embargo, un archivo elf también puede ser un Exploit. La ventaja de agregarlo como tipo de archivo de entrada es poder alimentar el sistema con archivos elf encontrados fuera de aplicaciones. De esta forma, se pueden crear flags que los identifiquen sin necesidad de encontrarlos dentro de aplicaciones. Ésto crea un sistema mas preventivo y menos reactivo.

Actualmente, los archivos elf son analizados pero no son aceptados como archivo de entrada. Están en un estado intermedio pues se guardan como muestra (tabla *ElfFile*, asociado a *Source* y los strings de elf asociados a *ElfFile*). Sin embargo, sólo son analizados cuando se encuentran en un apk. Para terminar la transición, hay que implementar un *ElfSample*, mover el análisis de strings de *InnerELFAnalyzer* a un nuevo *ElfStringsPicker* asociado a elf y configurar *InnerELFAnalyzer* para que llame a *AndroidDisassembler* cada vez que encuentra un elf en un apk.

Al igual que con el caso de los apk (*InnerAPKAnalyzer*) y dex (*InnerDEXAnalyzer*), si se acepta un nuevo tipo de archivo de entrada A, y éste puede ser encontrado dentro de otro tipo B (por ejemplo un jpg dentro de un apk), hay que agregar un *InformationPicker* para el tipo B que busque los archivos A y llame a *AndroidDisassembler* para su análisis. De esta forma el sistema se mantiene modular, flexible y completo.

Capítulo 5

Discusión

5.1. Tipos de Malware y sus comportamientos

Con la popularidad del sistema operativo Android y su facilidad para publicar y compartir aplicaciones, la cantidad de malware para Android ha aumentado considerablemente durante el 2011 [38]. A continuación se muestra una caracterización del malware actual.

Hasta ahora, el malware presente en Android no es muy complejo. En efecto, sólo existen pocos Exploits (6: Asroot, Exploid, RATC, KillingInTheNameOf, GingerBreak y zergRush) [38] y la mayor parte del malware utiliza ingeniería social para instalarse en el dispositivo. Las tres formas más comunes de engañar al usuario para instalar un malware son:

1. **Repackaging:** Troyanizar o re-empaquetar una aplicación conocida y limpia con el comportamiento malicioso. Ésto es particularmente simple dado que sólo hay que agregar el código malicioso, sin necesidad de alterar el funcionamiento original. En efecto, existe mucho malware que funciona como aplicación conocida y por detrás ejecuta el ataque. Según [38], un 86 % del malware utiliza esta técnica.
2. **Update Attack:** Aparentar ser una aplicación inofensiva, la cual a su vez, descarga sin el consentimiento del usuario, una aplicación maliciosa que efectúa el ataque.
3. **Drive-by Download:** Pretender ser una aplicación interesante para el usuario (mediante falsa publicidad y falsas promesas) para lograr que éste instale la aplicación en el dispositivo.

Por otro lado, las funcionalidades o tipos de ataque que realiza el malware hasta ahora son principalmente las que se muestran a continuación. Si bien, existen varias muestras que realizan sólo un tipo de ataque, también existen muchas familias de malware que utilizan varias combinaciones de ellas, aumentando su efecto.

1. **Premium SMS:** Ésta es la forma más común de generar ganancias monetarias para un desarrollador de malware. Consiste en que, una vez instalado el malware, se envía, sin el consentimiento del usuario, mensajes de texto (SMS) a servicios premium. Estos servicios tienen un costo extra asociado al mensaje enviado y es por este medio que los desarrolladores reciben su pago. Cabe destacar que, usualmente, estos servicios envían un mensaje de

confirmación al usuario. Sin embargo, el malware se encarga de interceptar este mensaje y evitar que el usuario se entere.

2. **Obtener root:** Otro tipo de ataque consiste en utilizar algún Exploit con el fin de obtener privilegios de administrador (root). Ésto permite tener un control completo sobre el dispositivo sin que el usuario lo note y facilita los siguientes ataques. Según [38], un 36,7 % del malware utiliza exploits para obtener privilegios de root.

3. **Espiar al usuario:** Una funcionalidad común a numerosos malware (93 % según [38]) es espiar al usuario. Consiste en obtener distinta información del usuario y subirla a un servidor remoto. La información recolectada suele ser: los identificadores del teléfono (IMEI, IMSI, numero de teléfono, IP), las cuentas del usuario (mail, twitter, facebook), contenido del usuario (favoritos, SMS, registro de llamadas, contactos, historial del navegador) y en la segunda mitad del 2011 aparecieron dos malware (Zitmo (Julio) y Spitmo (Septiembre)) que buscan obtener las credenciales de seguridad para la cuenta del banco del usuario (segundo factor de autenticación para transacciones). Estos dos últimos, trabajan en conjunto con malware de PC para obtener el resto de los factores de autenticación y poder realizar un fraude.

4. **Botnet:** Otra funcionalidad común es la de control remoto o Botnet. Ésta consiste en poder enviar comandos al malware vía internet o SMS para que realice futuras acciones, como actualizarse, espiar al usuario o instalar otra aplicación.

5. **Dropper:** También existen malware que traen consigo una segunda aplicación maliciosa, la cual instalan en el sistema. Esta segunda aplicación es la que suele realizar el ataque principal, mientras que el objetivo de la primera es engañar al usuario para poder instalarla.

Hasta ahora, sólo se ha encontrado un malware que tenga comportamiento destructivo (que dañe directamente la información del usuario). Este malware (Moghaba/Stamper), inserta una imagen predefinida en todas las fotos del usuario, sin poder recuperarlas. Además, posiblemente debido a un error de programación, inserta muchas veces la imagen en las fotos, aumentando su tamaño y llenando la memoria del dispositivo.

Si bien han habido varios casos de malware en Google Play (Android Market), la mayor parte del malware se encuentra en markets alternativos o páginas web maliciosas. Por otro lado, aunque la complejidad encontrada es muy inferior a del malware de PC, se ha observado una rápida evolución en los tipos de ataques, los objetivos y las metodologías utilizadas para lograrlos.

5.2. Efectividad del sistema

5.2.1. Ventajas y desventajas

A continuación se explican las ventajas y desventajas de Samsaraa.

En primer lugar, el sistema creado permite analizar rápida y masivamente grandes cantidades de aplicaciones. En efecto, un apk se demora un promedio de 2 segundos (2.04s) en analizar y

medio segundo (0.46s) en calificar. Ésto demuestra la velocidad de Samsaraa, que analiza y califica casi 1500 muestras por hora, con sólo 4 procesadores. Además, debido a su diseño modular y al análisis en paralelo se puede extender fácilmente para utilizar más procesadores o varios servidores simultáneamente. De esta manera, se cumple uno de los objetivos principales: poder analizar grandes cantidades de aplicaciones en poco tiempo.

Sin embargo, aunque el análisis sea rápido, el resultado y la información recolectada es notablemente detallada. En efecto, en sólo 2 segundos se obtiene una visión general de todas las características de la muestra (información del apk) y la posibilidad de observar con más detalle las capacidades de ésta (información del dex). Además, la modularidad del análisis, permite extender aún más la información recolectada o por el contrario, evitar analizar algunas partes para acelerar el análisis.

Otro punto a destacar, es que gracias a que toda la información es almacenada en una base de datos adecuada, es factible realizar futuros análisis sobre los resultados. En particular, es posible calcular estadísticas o realizar minería de datos sobre las muestras y así caracterizar de mejor forma tanto las aplicaciones como el malware. También es posible, utilizar toda la información para generar informes globales e ir evaluando cómo evolucionan las aplicaciones y el malware.

Por último, gracias a los dos puntos anteriores, Samsaraa permite analizar, calificar y comparar distintos grupos de aplicaciones. Es posible generar informes grupales sobre las muestras y obtener velozmente una caracterización del conjunto.

Por otra parte, el sistema tiene principalmente tres desventajas. La primera es un problema de escalabilidad: dada la gran cantidad de información que recolecta (sobre todo strings y funciones) el tamaño de la base de datos crece rápidamente, lo cual va a ser un problema cuando existan millones de aplicaciones analizadas. Sin embargo, como es un problema de escalabilidad se puede resolver, por ejemplo, comprimiendo y sacando de la base de datos principal los datos de aplicaciones más viejas y dejando sólo un informe de las antiguas y toda la información de las más recientes. Ésto permitiría poder consultar rápidamente las aplicaciones nuevas sin perder la información antigua a cambio de que el acceso sea un poco más lento.

La segunda desventaja, es que, si bien se analiza gran parte de la muestra, no se analiza completamente. Es decir que si existe algún comportamiento malicioso en una fracción que no se analiza (por ejemplo, las imágenes) no será posible detectar este comportamiento. A causa de ésto, el desarrollador debe ser preventivo, estar al tanto de los posibles ataques que puedan existir e implementar los *InformationPickers* necesarios para analizar las fuentes de esos ataques.

Y la tercera desventaja es que el análisis realizado es solamente estático. En efecto, dado que no se ejecuta el código, no se efectúa un análisis dinámico. Esto implica que no es posible saber si las funciones relevantes se ejecutan o no, por ejemplo, si la aplicación realmente se conecta a un servidor remoto o si efectivamente envía mensajes de texto sin el consentimiento del usuario. Tampoco es posible saber cuantas veces se ejecuta una rutina en particular. Por otro lado, si el código tiene la funcionalidad para descargar remotamente otra aplicación o ejecutable, no es factible analizar los nuevos componentes descargados pues no están disponibles para el análisis estático.

5.2.2. Estadísticas del sistema

A continuación se entregan algunas estadísticas de funcionamiento de Samsaraa.

Para testear el funcionamiento del sistema, se utilizó un conjunto de 13.000 muestras entregadas por AV-Test [3]. Estas pruebas se realizaron en una maquina virtual de Ubuntu (utilizando VMware) con 4 procesadores (Intel Core i7, 3.4GHz) y 6GB de RAM. El análisis fue realizado en grupos de 1.000 archivos para ver el avance progresivamente. Cada uno de estos grupos se demoró un promedio de 12 minutos en ser analizados (0,7s/muestra) y se detectó que no todos los archivos eran apk. También existían muchos dex, varios elf y otro tipo de archivos que no se analizaron.

Conjunto	**Cantidad**	**Fuente**	**Comportamiento**
A	105	Google Play [12] (desarrolladores destacados y más descargas).	Limpio
B	68	Celular Samsung Galaxy SII (disponibles por defecto desde fabrica).	Limpio
C	339	Base de datos de malware de McAfee.	Malware
D	6	Contagio Mobile [24].	Malware
E	60	Al azar entre las fuentes A, B, C y markets alternativos.	Aleatorio

Tabla 5.1: Conjunto de muestras utilizadas para testear la eficiencia y eficacia del sistema.

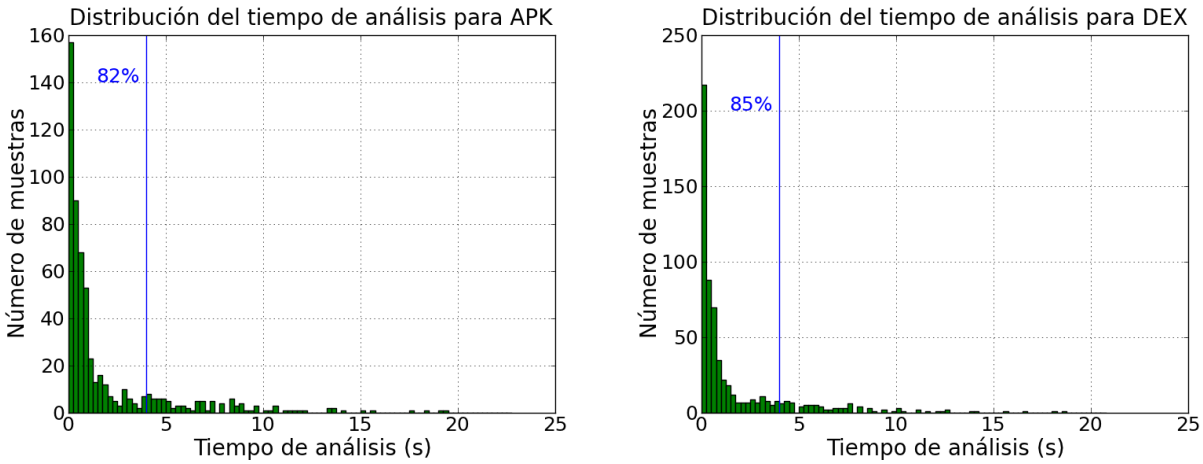
Sin embargo, luego de analizar un segundo conjunto de muestras (ver tabla 5.1), se llegó a la conclusión de que el conjunto de AV-Test no era representativo de aplicaciones comunes y/o populares que suelen utilizar los usuarios. En efecto, la mayoría de las aplicaciones de AV-Test eran más livianas, menos complejas, más rápidas de analizar y principalmente archivos maliciosos. Si bien el objetivo del sistema es detectar malware, la mayor parte de aplicaciones que analiza se espera que sean limpias, y por lo tanto, para el buen funcionamiento del software es necesario saber cómo reacciona frente a éstas. En la tabla 5.2 se muestran estadísticas de los pesos y tiempos de análisis para apk y dex para este conjunto de muestras.

	APK		**DEX**	
	Tiempo de análisis	Tamaño	Tiempo de análisis	Tamaño
Máximo	23.33 s	29.8 MB	21.48 s	6.4 MB
Promedio	2.04 s	1.8 MB	1.74 s	489.9 KB
Mínimo	0.04 s	7.5 KB	0.01 s	1.6 KB
Desviación estándar	3.28 s	3.2 MB	2.91 s	819.9 KB

Tabla 5.2: Tamaños y tiempos de análisis para apk y dex. Conjuntos A-E.

De la tabla 5.2 y los gráficos de distribución del tiempo de análisis en la figura 5.1 se puede observar que, para apk, aunque el mayor tiempo de análisis fue 23s, un 82 % se demora menos de 4s, con un promedio de 2s. Y por otro lado, para dex, el máximo son 21s, un 85 % se demora menos

de 4s y con un promedio de 1.74s. Además, se puede destacar que dado que todos los apk contienen un dex en su interior, la mayor parte del tiempo de análisis de un apk es por analizar ese dex (el tiempo de apk incluye el tiempo de su dex interior).



(a) Apk

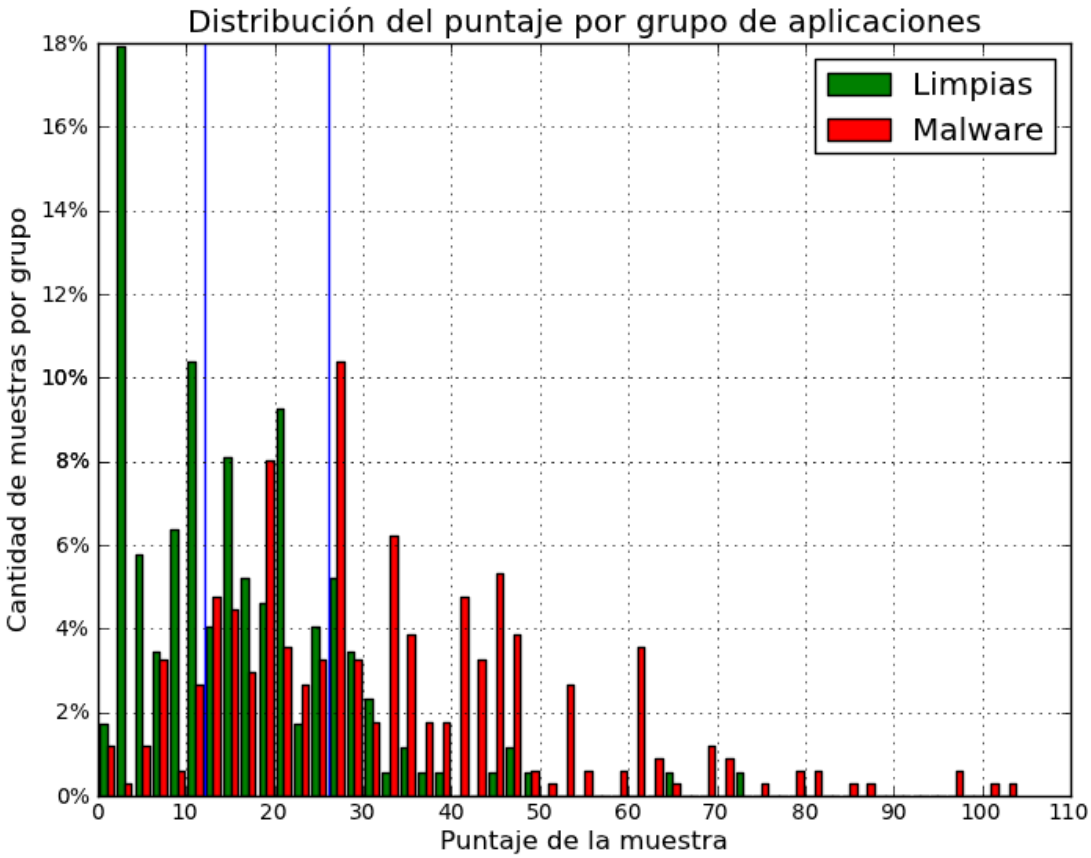
(b) Dex

Figura 5.1: Distribución del tiempo de análisis para apk y dex.

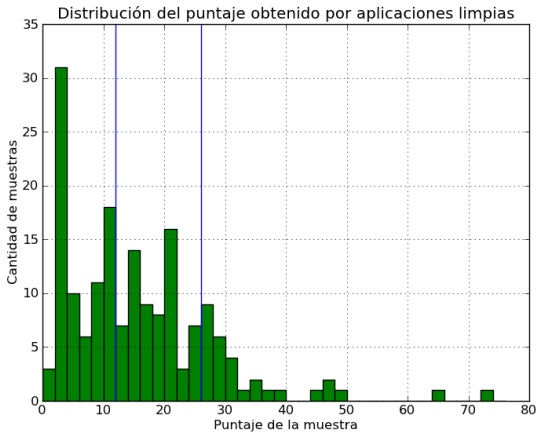
5.2.3. Eficacia del sistema de flags

El objetivo principal del sistema es poder hacer un ranking de todas las aplicaciones analizadas y resaltar las que tengan mayor probabilidad de ser malware. Estas muestras serán seleccionadas por los investigadores de McAfee y analizadas manualmente una por una para confirmar su peligro para el usuario. El objetivo de filtrar las muestras es que, dado la gran cantidad de aplicaciones existentes y la velocidad con que surgen nuevas, es imposible analizarlas todas manualmente, por lo que es imprescindible filtrar las más importantes y peligrosas. Un objetivo derivado es poder clasificar correctamente todas las aplicaciones entre malware y limpio. Si bien es un objetivo mayor, sería ideal poder decidir ésto automáticamente.

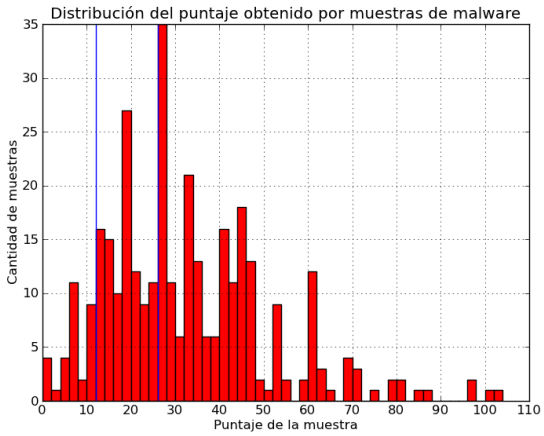
Tal como se mencionó anteriormente, el sistema de flags es la inteligencia del sistema y cumple el objetivo de encontrar las aplicaciones más riesgosas. Para testear su eficacia para calificar, se realizó lo siguiente: se eligieron dos grupos de aplicaciones, limpias y malware (ver tabla 5.1), se obtuvo su puntaje y se verificó si puede diferenciar correctamente los dos grupos. Tal como se explicó anteriormente, el puntaje es un entero positivo en una escala arbitraria, suma del peso de los flags que calzaron. Por lo tanto, mientras mayor sea el puntaje, mayor es la probabilidad de que la muestra sea malware. En la figura 5.2 se muestra este resultado. En ella se puede observar la distribución del puntaje obtenido por los dos grupos: aplicaciones limpias (en verde) y malware (en rojo). De la figura 5.2 deriva la tabla 5.3 en donde se observa la efectividad del sistema al clasificar aplicaciones.



(a) Comparación entre aplicaciones limpias y malware. El número de aplicaciones está normalizado para poder comparar ambos grupos.



(b) Aplicaciones limpias



(c) Muestras de malware

Figura 5.2: Distribución del puntaje para aplicaciones limpias y malware. Una aplicación con 26 puntos o más se puede considerar malware y con menos de 12 puntos, limpia.

	Correctamente clasificados	Falsos Positivos	Falsos Negativos	Indefinidos	Total
Malware	206 (61.1 %)	0	31 (9.2 %)	100 (29.7 %)	337 (66 %)
Limpias	79 (45.7 %)	30 (17.3 %)	0	64 (37.0 %)	173 (34 %)
Total	285 (55.9 %)	30 (5.9 %)	31 (6.1 %)	164 (32.2 %)	510 (100 %)

Tabla 5.3: Resultados de la efectividad del sistema.

	Cantidad de Malware		Cantidad de Limpias		Probabilidad de ser malware
	Absoluta	Normalizada	Absoluta	Normalizada	
Muestras sobre 26	206	0.611	30	0.173	77.9 %
Muestras bajo 12	31	0.092	79	0.457	16.8 %

Tabla 5.4: Probabilidades de que una muestra sea malware.

De la figura 5.2 se puede destacar que, casi todas las aplicaciones que tienen 26 puntos o más son malware (con 77.9 % de probabilidad, ver tabla 5.4), mientras que una aplicación con menos de 12 puntos es casi seguramente limpia (con 83.2 % de probabilidad, ver tabla 5.4). En total, un 55.9 % de las muestras fueron clasificadas correctamente.

Existe un 5.9 % de falsos positivos, es decir aplicaciones limpias que tienen 26 puntos o más y podrían considerarse malware. Por ejemplo, dos de las aplicaciones limpias tienen cerca de 70 puntos: el antivirus de Lookout y el antivirus de McAfee. Si bien son aplicaciones sin comportamiento malicioso, al ser antivirus, utilizan muchas características sospechosas para una aplicación normal, como por ejemplo:

- Obtener la lista de aplicaciones instaladas para poder analizarlas,
- Acceder a los datos del teléfono (IMEI, IMSI, numero de teléfono) para poder identificarlo correctamente,
- Escuchar los Broadcast de “Reinicio Completo” para iniciar el antivirus apenas se prenda el celular,
- Pedir permisos de SMS, para verificar el numero de teléfono (se pide el numero de teléfono y se confirma mediante un SMS),
- Se capturan los SMS, para confirmar el número telefónico,
- Poseen URLs para poder actualizarse y comunicarse con el servidor respectivo,
- Requieren muchos permisos pues necesitan acceso a varias características del sistema para funcionar bien.

Por otro lado, existe un 6.1 % de falsos negativos, es decir malware que fue clasificado como limpio. En efecto, hay malware que tiene menos de 12 puntos. Ésto se debe a que son aplicaciones

maliciosas muy simples, que no utilizan muchas características y usualmente sólo realizan la acción maliciosa. Por ejemplo, enviar SMS premium, instalar otra aplicación o enviar los datos del usuario a un servidor remoto. De esta forma, calzan con pocos flags y reciben bajo puntaje.

Sin embargo, existe un rango indefinido correspondiente al 32.2 % de las muestras analizadas. En efecto, si una muestra tiene entre 12 y 25 puntos incluidos, no existe mucha claridad sobre si corresponde a un malware o no. Esta indecisión se debe a que existen muchas aplicaciones limpias que utilizan funciones y comportamientos sospechosos. En particular, muchos de los paquetes de publicidad (ads) utilizan encriptación para la comunicación con su servidor, trucos rebuscados para evitar que las publicidades sean removidas, obtienen el IMEI o IMSI para identificar al usuario o utilizan Internet y URLs para informar acerca de las publicidades vistas. De esta forma, calzan con varios comportamientos sospechosos sin que sean necesariamente maliciosos. Y de igual forma, existe malware más básico que no ocupa muchas técnicas sospechosas y por ende no obtiene suficiente puntaje para resaltar.

En el área de seguridad computacional, a la hora de decidir si una aplicación es segura o no, existen principalmente dos formas de proceder: optimista y pesimista. El ser optimista prefiere disminuir los falsos positivos mientras que un ser pesimista prefiere disminuir los falsos negativos. Un usuario común puede preferir borrar una aplicación limpia pensando que es malware y asegurarse de estar protegido (pesimista). Mientras que un anti-virus puede preferir mantener un malware pensando que está limpio y evitar borrar un archivo imprescindible para el sistema por considerarlo malware (optimista). Generalmente, disminuir la cantidad de un tipo de error aumenta la cantidad del otro tipo. En el caso de Samsaraa, dado que su decisión no es final, es mejor ser pesimista. De esta forma, dado que las aplicaciones con mayor puntaje serán analizadas a mano, si una aplicación limpia obtuvo alto puntaje, esta será descubierta por el análisis manual.

Podemos concluir que, aunque el sistema cumple el objetivo principal de calificar y resaltar las muestras más sospechosas (creando el ranking), aún se puede perfeccionar para que la zona indefinida sea menor. De esta manera, la calificación será más confiable y se podrá usar, no sólo para detectar las muestras más peligrosas, si no que también para determinar si una muestra es malware o está limpia. En particular, una de las mejoras realizables es implementar y utilizar los *AdvancedFlag*. Este tipo de flags permitiría crear flags con excepciones, por ejemplo, marcar las aplicaciones que utilicen archivos binarios (elf) pero que no sean binarios conocidos (como librerías gráficas o matemáticas); marcar las aplicaciones que usen criptografía u obtengan el IMEI/IMSI, siempre y cuando no sea en un package de publicidad; o utilizar "Site Advisor" (servicio de McAfee para validar URLs) para medir la confiabilidad de las URLs encontradas y según éso subir el puntaje. Esta medida perfeccionaría el puntaje entregado pues el análisis de las características de una muestra sería realizado con más detalle, lo que permitiría distinguir de mejor forma, una función sospechosa usada maliciosamente de una que no.

Otra manera de perfeccionar el sistema es utilizar la información obtenida mediante minería de datos. En particular, luego de analizar varios grupos de aplicaciones limpias y malware, se puede utilizar la minería para caracterizar los dos grupos en forma más precisa (utilizando clusterización por ejemplo) y así definir cuáles son los flags que caracterizan un malware o una aplicación limpia, modificando y ajustando los pesos de éstas para que califiquen mejor.

5.3. Comparación con softwares similares

Al inicio del trabajo (Febrero 2012) no existían sistemas que ofrecieran la misma funcionalidad, o parecida. Sin embargo, durante el transcurso de éste, se publicaron al menos tres softwares similares. La aparición de estos sistemas aportó negativa y positivamente al proyecto. Negativamente, pues Samsaraa dejó de ser el único; pero positivamente pues si varias personas están trabajando en lo mismo, significa que es un tema interesante, importante y actual. Además, los trabajos que hicieron una publicación (particularmente el de Axelle Apvrille [2], Mayo 2012) se aprovecharon para aportar nuevas ideas al diseño del sistema, principalmente al módulo de Flags. A continuación se realiza una comparación entre los distintos sistemas.

Una de las funciones del software desarrollado por Axelle Apvrille y Tim Strazzere [2] consiste en una heurística para detectar aplicaciones sospechosas entre una gran cantidad descargada por un crawler (ver sección 2.2.3). Una de las ventajas de Samsaraa sobre el de Apvrille es que analiza recursivamente las aplicaciones. En otras palabras, el sistema de Apvrille detecta cuando un apk contiene otro apk, sin embargo no lo analiza; por otro lado, Samsaraa detecta y analiza recursivamente cualquier apk que encuentre. Ésto permite generar un análisis más profundo y preciso sobre la aplicación en cuestión.

Si bien los autores no publicaron el código fuente o binario ni explicaron extensivamente el funcionamiento de su sistema, de lo expuesto en su trabajo se puede inferir que el resultado de su análisis no es almacenado. Es decir que cuando su sistema analiza una aplicación, el resultado sólo se muestra en consola. Ésto dificulta la capacidad para calcular estadísticas o realizar minería de datos al tener que agregar otro módulo o programa que adapte el resultado a un formato práctico para trabajar.

Un segundo sistema similar a Samsaraa, es un servicio llamado Andrubis [16] (ver sección 2.2.9). El usuario puede subir una aplicación y recibir un informe de su análisis. Una ventaja de Andrubis frente a este sistema es que, además del análisis estático, realiza uno dinámico. El informe generado permite entonces conocer algunas acciones que realizó la aplicación mientras se ejecutaba. Sin embargo, el informe sobre el análisis estático no es tan completo como el generado por Samsaraa. No entrega un reporte o nivel de riesgo sobre la aplicación, sino únicamente el resultado del análisis. Dado que Andrubis funciona como servicio, su código es privado y no hay documentación sobre su funcionamiento, no es posible comparar más funcionalidades.

Finalmente, el tercer sistema con funcionalidades parecidas es Marvin Safe, de Mobile Security (ver sección 2.2.8). A diferencia de los anteriores, Marvin es una aplicación para Android que permite elegir una aplicación del dispositivo, subirla a la nube, realizar un análisis y recibir un informe. Al igual que Andrubis, realiza un análisis estático y dinámico, pero a diferencia de éste no entrega un informe completo. Probablemente debido a que está orientado a un usuario común (sin muchos conocimientos de computación), el informe sólo contiene un resumen de las características y comportamientos sospechosos (alarmas) que se detectaron en la aplicación. La desventaja frente a Samsaraa, es que no permite comparar fácilmente entre aplicaciones o grupos de aplicaciones. Además, como el código es privado, no han realizado publicaciones y el servicio está orientado a usuarios comunes, la información que entrega no es muy útil para realizar futuros estudios.

5.4. Trabajo futuro y posibles extensiones

Samsaraa se encuentra actualmente en un buen estado para entorno de producción. Sin embargo, existen muchas mejoras y módulos o funcionalidades que se pueden agregar para perfeccionarlo. A continuación se explican algunas de ellas.

Una de las maneras de enriquecer el sistema es extender el módulo Analizador. Por ejemplo, se puede agregar un nuevo *InformationPicker* para realizar un análisis estático del flujo de ejecución de la muestra. Ésto permitiría crear un árbol de ejecución de cada aplicación (similar al generado por IDA Pro [27]) y poder comparar cada muestra por su árbol. También se puede agregar la capacidad de análisis para archivos jar, zip u otros posibles tipos de archivos ejecutables por Android.

Una manera para incrementar la precisión del sistema de flags es utilizar toda la información recolectada para realizar minería de datos. En particular, se podrían usar técnicas de clusterización para agrupar el malware según sus características o perfeccionar la separación entre malware y aplicaciones limpias. Ésto serviría para definir de mejor manera el peso de los flags. De igual forma, se pueden usar técnicas de minería de datos para buscar características comunes para un grupo de muestras y crear nuevos flags en base a estos descubrimientos.

Otra línea de posibles mejoras es aumentar la escalabilidad del sistema. Dado que fue diseñado para analizar grandes cantidades de aplicaciones, es necesario que posea una infraestructura que permita almacenar y procesar gran cantidad de información y analizar varias muestras simultáneamente. Si bien el módulo de análisis fue diseñado para ser escalable fácilmente, existe una modificación que puede facilitar la tarea cuando haya que consultar grandes cantidades de información. Esta mejora consiste en invertir los ciclos de análisis. Es decir, actualmente el sistema analiza cada *InformationPicker* por muestra, lo que genera muchas consultas a la base de datos (varias por cada *InformationPicker* y por cada muestra). La idea es analizar cada muestra por *InformationPicker* y de esta manera agrupar las consultas por *InformationPicker* y disminuir la cantidad considerablemente (un par por cada *InformationPicker* y grupo de muestras). Gracias a este cambio, se puede implementar un módulo que espere una cantidad definida de muestras y luego vaya tomando los *InformationPickers* uno por uno y analizando todas las muestras antes de pasar al siguiente (procesamiento por lotes). Si bien el tiempo en que se complete el análisis de una muestra aumentará, el tiempo promedio agregado disminuirá.

Una funcionalidad que fue sugerida por McAfee es implementar un módulo de reportes. El objetivo es crear fácilmente (vía la interfaz web) distintos tipos de informes (estadísticas, características en común, patrones similares) sobre un grupo de muestras en particular. Esto permitiría a los investigadores de McAfee elegir un grupo de aplicaciones y rápidamente obtener una visión general de éstas para ser entregada a clientes o usuarios con menos conocimientos en computación.

También, aunque requiere más tiempo y diseño que las mejoras anteriores, para completar el sistema se puede agregar un módulo de análisis dinámico. Una idea es crear una granja de servidores que tengan corriendo varios emuladores de Android y que reciban las muestras para que se ejecuten durante un tiempo. De esta forma se puede analizar el comportamiento que realmente tiene la aplicación y complementar el informe de análisis estático. Además, la información del

análisis estático sirve de guía preliminar al ejecutar la muestra. Por ejemplo, se podría realizar un análisis de las interfaces de usuario (definidas en archivos xml en el caso de Android) para detectar los botones y poder realizar una ejecución más real.

Además, para el usuario común se puede ofrecer el servicio de análisis remoto. En otras palabras, si un usuario posee una aplicación de la cual sospecha, él la puede enviar a Samsaraa para que sea analizada. Luego el usuario recibiría un informe sobre el análisis, adaptado para usuarios sin conocimientos en computación.

Finalmente, la lógica del sistema puede ser exportada a iOS. Si bien son sistemas con muchas diferencias, la teoría detrás del análisis implementado se puede utilizar en aplicaciones de iOS. Sin embargo, ésto implica investigar y analizar aplicaciones para iOS, estudiar el entorno de ejecución de las aplicaciones en iPhone y diseñar flags adecuados para el sistema.

Capítulo 6

Conclusiones

Para poder realizar este proyecto, una de las tareas iniciales fue estudiar el funcionamiento del sistema operativo Android, de sus componentes y principalmente de sus aplicaciones. De esta forma se pudo comprender el entorno sobre el cual funciona el malware que se quiere estudiar. Por otro lado, también se investigó sobre las diversas técnicas para analizar aplicaciones de Android y las herramientas disponibles para realizarlo. Ésto dio una visión de las capacidades actuales de análisis y de lo que aún hace falta para alcanzar el nivel de herramientas que existen para analizar malware en PC.

Del análisis y estudio de las técnicas de ataque y características del malware actual para Android, se concluye que las formas más comunes para lograr que un usuario instale malware en un dispositivo móvil son: troyanizar una aplicación conocida con código malicioso (Repackaging), pretender ser una aplicación inofensiva que descarga el malware atacante (Update Attack) y convencer al usuario que la aplicación es interesante para que instale el malware voluntariamente (Drive-by Download). Por otro lado, las funcionalidades y tipos de ataques más frecuentes son: enviar mensajes a servicios premium de SMS, obtener permisos de administrador, espiar y obtener información del usuario, capacidad para recibir comandos remotos (Botnet) e instalar otras aplicaciones o malware. Hasta ahora, sólo se ha descubierto un tipo de malware que destruya datos del usuario.

Actualmente, la mayor parte del malware descubierto es bastante simple en términos de ingeniería y programación. Sólo se conocen 6 tipos de exploits, que son utilizados por un 36,7 % del malware, y la mayoría utiliza ingeniería social para poder entrar al dispositivo. Si bien, casi todas las aplicaciones maliciosas se han encontrado en markets alternativos, también han habido algunos casos de malware en Google Play. Aunque la complejidad encontrada es muy inferior a la del malware de PC, se ha observado una rápida evolución en los tipos de ataques, los objetivos y las metodologías utilizadas para lograrlos. Y mientras el celular pueda hacer más cosas (como transferencias bancarias) y esté cada vez más presente en la vida de las personas, existen más incentivos para que los desarrolladores de malware ataquen.

Finalmente, el objetivo principal de este proyecto fue diseñar e implementar una herramienta para analizar aplicaciones de Android en busca de malware. Samsaraa, el sistema desarrollado, tiene la capacidad para analizar estáticamente grandes cantidades de muestras. El análisis y la calificación se realizan paralelamente para cada muestra, demorando un promedio de 2,5 segundos. El resultado obtenido del análisis es un informe completo y detallado sobre todas las características de la muestra. El resultado de la calificación es un puntaje de riesgo potencial y la lista de funcionalidades

sospechosas que posee la muestra. Esta información es utilizada para generar un ranking y guiar a los investigadores hacia las aplicaciones mas sospechosas. Además, Samsaraa almacena toda la información recolectada en una base de datos adecuada, que permite calcular estadísticas y realizar minería de datos para un estudio agregado más acabado. Por último, el sistema ofrece la opción de comparar los resultados entre dos conjuntos de aplicaciones y facilitar la generación de informes por grupo.

De las 510 muestras analizadas, un 55.9 % fue bien clasificada, con un 5.9 % de falsos positivos, un 6.1 % de falsos negativos y un 32.2 % de indefinidos. Aunque el sistema cumple el objetivo de generar un ranking para seleccionar las aplicaciones más riesgosas, aún se puede mejorar. En particular, se puede implementar el último tipo de flag: *AdvancedFlag*, que permitiría definir flags más detallados o utilizar la información recolectada y técnicas de minería de datos para descubrir nuevos flags o ajustar el peso de éstos.

En comparación con los pocos sistemas parecidos que existen (publicados a partir de mayo 2012), Samsaraa tiene ventajas como el análisis recursivo de apk, el nivel de detalle del informe entregado, la capacidad para comparar grupos de muestras y/o la facilidad para realizar estudios sobre la información recolectada.

Así como el malware está evolucionando rápidamente en sus capacidades, también lo tienen que hacer las herramientas y sistemas para combatirlos. De esta forma, Samsaraa se convierte en uno de los sistemas que facilitará el análisis, el estudio y la prevención de malware sobre dispositivos móviles, una de las áreas importantes en la realidad actual y que afecta a la gran mayoría de las personas.

Bibliografía

[1] Apvrille, Axelle: *Android Reverse Engineering Tools*. < http://www.fortiguard.com/sites/default/files/insomnidroid.pdf >, [Marzo 2012].

[2] Apvrille, Axelle y Tim Strazzere: *Reducing the window of opportunity for Android malware Gotta catch 'em all*. J. Comput. Virol., 8(1-2):61–71, Mayo 2012, ISSN 1772-9890. <http://dx.doi.org/10.1007/s11416-012-0162-3>.

[3] AV-Test: *AV-TEST - The Independent IT-Security Institute*. < http://www.av-test.org/en/home/ >, [Septiembre 2012].

[4] Blog, The iSecLab: *Andrubis: A Tool for Analyzing Unknown Android Applications*. < http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2/ >, [Junio 2012].

[5] Castillo, Carlos A.: *Android Malware Past, Present, and Future*. < http://www.mcafee.com/us/resources/white-papers/wp-android-malware-past-present-future.pdf >, [Noviembre 2011].

[6] Desnos, Anthony: *Androguard*. < http://code.google.com/p/androguard/ >, [Septiembre 2012].

[7] Dupuy, Emmanuel: *JD-GUI*. < http://jd-gui.softonic.com/ >, [Septiembre 2012].

[8] Enck, William, Damien Octeau, Patrick McDaniel y Swarat Chaudhuri: *A Study of Android Application Security*. En *Proceedings of the 20th USENIX Security Symposium*, August 2011. <http://www.enck.org/pubs/enck-sec11.pdf>.

[9] Google: *Android and Security*. < http://googlemobile.blogspot.com/2012/02/android-and-security.html >, [Febrero 2012].

[10] Google: *Android SDK Developing Tools*. < http://developer.android.com/guide/developing/tools/ >, [Septiembre 2012].

[11] Google: *The Developer's Guide*. < http://developer.android.com/guide/ >, [Septiembre 2012].

[12] Google: *Google Play Market*. < https://play.google.com/store >, [Septiembre 2012].

[13] Gruver, Ben: *baksmali disassembler*. < http://code.google.com/p/smali/ >, [Septiembre 2012].

[14] Holz, Thorsten: *Systems Security Research at Ruhr-University Bochum*. SysSec Workshop, 0:131–134, 2011.

[15] Kouznetsov, Pavel: *JAD Java Decompiler*. < http://www.varaneckas.com/jad/ >, [Septiembre 2012].

[16] Lab, International Secure Systems: *Andrubis*. < http://anubis.iseclab.org/ >, [Septiembre 2012].

[17] Lantz, Patrick: *Droidbox*. < http://code.google.com/p/droidbox/ >, [Septiembre 2012].

[18] McAfee: *Malware on Android jumps 37 percent over previous quarter*. < http://www.androidguys.com/2011/11/21/mcafee-malware-on-android-jumps-37-percent-over-previous-quarter/ >, [Septiembre 2012].

[19] MobWorm project the: *Mobile Sandbox*. < http://www.mobile-sandbox.com/ >, [Septiembre 2012].

[20] National Security Systems (CNSS), Committee on: *National Information Assurance Glossary*. < http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf >, [Abril 2010].

[21] Neshkov, Atanas: *DJ Java Decompiler*. < http://dj-java-decompiler.softonic.com/ >, [Septiembre 2012].

[22] Octeau, Damien, William Enck y Patrick McDaniel: *The ded Decompiler*. Informe técnico NAS-TR-0140-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, September 2010. <http://siis.cse.psu.edu/ded/papers/NAS-TR-0140-2010.pdf>.

[23] Paller, Gabor: *dedexer*. < http://dedexer.sourceforge.net/ >, [Septiembre 2012].

[24] Parkour, Mila: *contagio mobile*. < http://contagiominidump.blogspot.com/ >, [Septiembre 2012].

[25] Project, The Honeynet: *Android Reverse Engineering Virtual Machine*. < https://redmine.honeynet.org/projects/are >, [Septiembre 2012].

[26] Project, The Honeynet: *The Honeynet Project Web Site*. < http://www.honeynet.org/ >, [Septiembre 2012].

[27] SA, Hex Rays: *Ida Pro*. < http://www.hex-rays.com/products/ida/ >, [Septiembre 2012].

[28] Security, Lookout Mobile: *Mobile Threat Report*. < https://www.mylookout.com/mobile-threat-report >, [Octubre 2011].

[29] Security, Marvin Mobile: *Marvin [Mobile Security]*. < http://marvinsafe.com/ >, [Septiembre 2012].

[30] Share, Net Market: *Browsing by Device Category Trend*. < http://marketshare.hitslink.com/report.aspx?qprid=61 >, [Septiembre 2012].

[31] Share, Net Market: *Mobile/Tablet Operating System Market Share*. < http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1 >, [Septiembre 2012].

[32] TechTerm: *TechTerms.com*. < http://www.techterms.com/ >, [Septiembre 2012].

[33] VirusTotal: *virustotal.com*. < https://www.virustotal.com/ >, [Septiembre 2012].

[34] Wiśniewski, Ryszard: *Android APK Tools*. < http://code.google.com/p/android-apktool/ >, [Septiembre 2012].

[35] [x]cubelabs: *The Android Story*. < http://www.xcubelabs.com/the-android-story.php >, [Septiembre 2012].

[36] Zhen, Cong: *ApkInspector*. < http://code.google.com/p/apkinspector/ >, [Septiembre 2012].

[37] Zhou, Yajin y Xuxian Jiang: *Android Malware Genome Project*. < http://www.malgenomeproject.org/ >, [Mayo 2012].

[38] Zhou, Yajin y Xuxian Jiang: *Dissecting Android Malware: Characterization and Evolution*. En *Security and Privacy (SP), 2012 IEEE Symposium on*, páginas 95 –109, Mayo 2012.

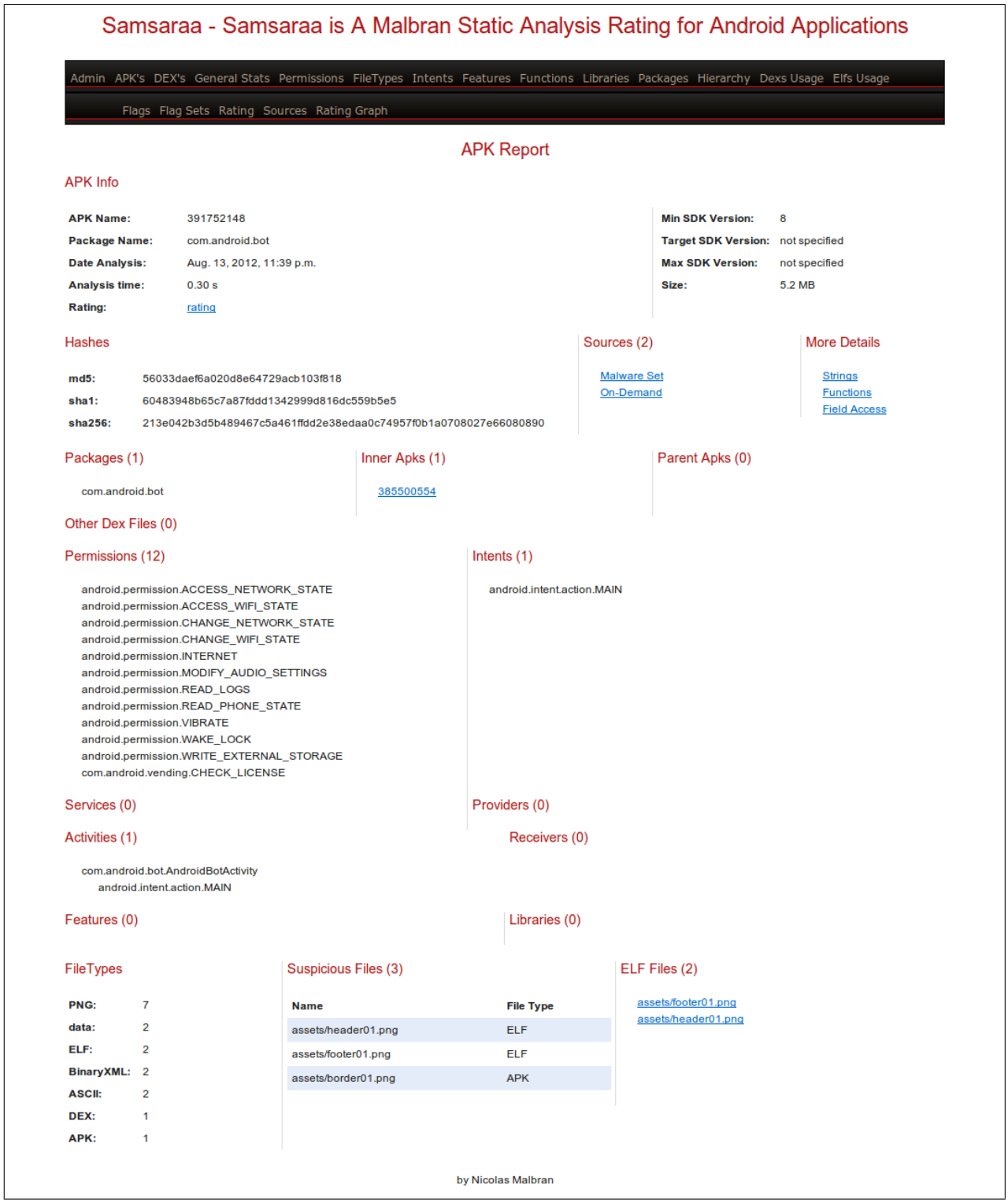
Apéndices

Apéndice A Ejemplo de archivo de configuración *samsaraa_config*

Código A.1: samsaraa_config

```
TEMP_DIR = "/tmp/samsaraa/"
DATABASES = {
    'default': {
        'ENGINE'   : 'django.db.backends.mysql',
        'NAME'     : 'samsaraa',
        'USER'     : '###',
        'PASSWORD' : '###',
    }}
APK_INFORMATION_PICKER_LIST = [
    {'package': 'samsaraa.analyser.information_pickers.apk',
     'classes': [
        'PackageInformationPicker',
        'ActivitiesPicker',
        'PermissionsPicker',
        'ReceiversPicker',
        'IntentsPicker',
        'ServicesPicker',
        # ...
        'InnerAPKAnalyser',
        'InnerELFAnalyser',
        'InnerDEXAnalyser',
        ]
    }]
DEX_INFORMATION_PICKER_LIST = [
    {'package': 'samsaraa.analyser.information_pickers.dex',
     'classes': [
        'InnerPackagePicker',
        'StringsPicker',
        'FunctionsPicker',
        'FieldAccessPicker',
        ]
    }]
VALID_FILES = {
    DEXFILENAME: DexSample,
    APKFILENAME: ApkSample,
}
```

Apéndice B Informe de análisis de un apk



Samsaraa - Samsaraa is A Malbran Static Analysis Rating for Android Applications

Admin APK's DEX's General Stats Permissions FileTypes Intents Features Functions Libraries Packages Hierarchy Dexs Usage Elfs Usage

Flags Flag Sets Rating Sources Rating Graph

APK Report

APK Info

APK Name:	391752148	**Min SDK Version:**	8
Package Name:	com.android.bot	**Target SDK Version:**	not specified
Date Analysis:	Aug. 13, 2012, 11:39 p.m.	**Max SDK Version:**	not specified
Analysis time:	0.30 s	**Size:**	5.2 MB
Rating:	rating		

Hashes

md5:	56033daef6a020d8e64729acb103f818
sha1:	60483948b65c7a87fddd1342999d816dc559b5e5
sha256:	213e042b3d5b489467c5a461ffdd2e38edaa0c74957f0b1a0708027e66080890

Sources (2)

Malware Set
On-Demand

More Details

Strings
Functions
Field Access

Packages (1)

com.android.bot

Inner Apks (1)

385500554

Parent Apks (0)

Other Dex Files (0)

Permissions (12)

android.permission.ACCESS_NETWORK_STATE
android.permission.ACCESS_WIFI_STATE
android.permission.CHANGE_NETWORK_STATE
android.permission.CHANGE_WIFI_STATE
android.permission.INTERNET
android.permission.MODIFY_AUDIO_SETTINGS
android.permission.READ_LOGS
android.permission.READ_PHONE_STATE
android.permission.VIBRATE
android.permission.WAKE_LOCK
android.permission.WRITE_EXTERNAL_STORAGE
com.android.vending.CHECK_LICENSE

Intents (1)

android.intent.action.MAIN

Services (0)

Providers (0)

Activities (1)

com.android.bot.AndroidBotActivity
android.intent.action.MAIN

Receivers (0)

Features (0)

Libraries (0)

FileTypes

PNG:	7
data:	2
ELF:	2
BinaryXML:	2
ASCII:	2
DEX:	1
APK:	1

Suspicious Files (3)

Name	File Type
assets/header01.png	ELF
assets/footer01.png	ELF
assets/border01.png	APK

ELF Files (2)

assets/footer01.png
assets/header01.png

by Nicolas Malbran

Figura B.1: Ejemplo de un informe de análisis para apk.

Apéndice C Informe de análisis de un dex

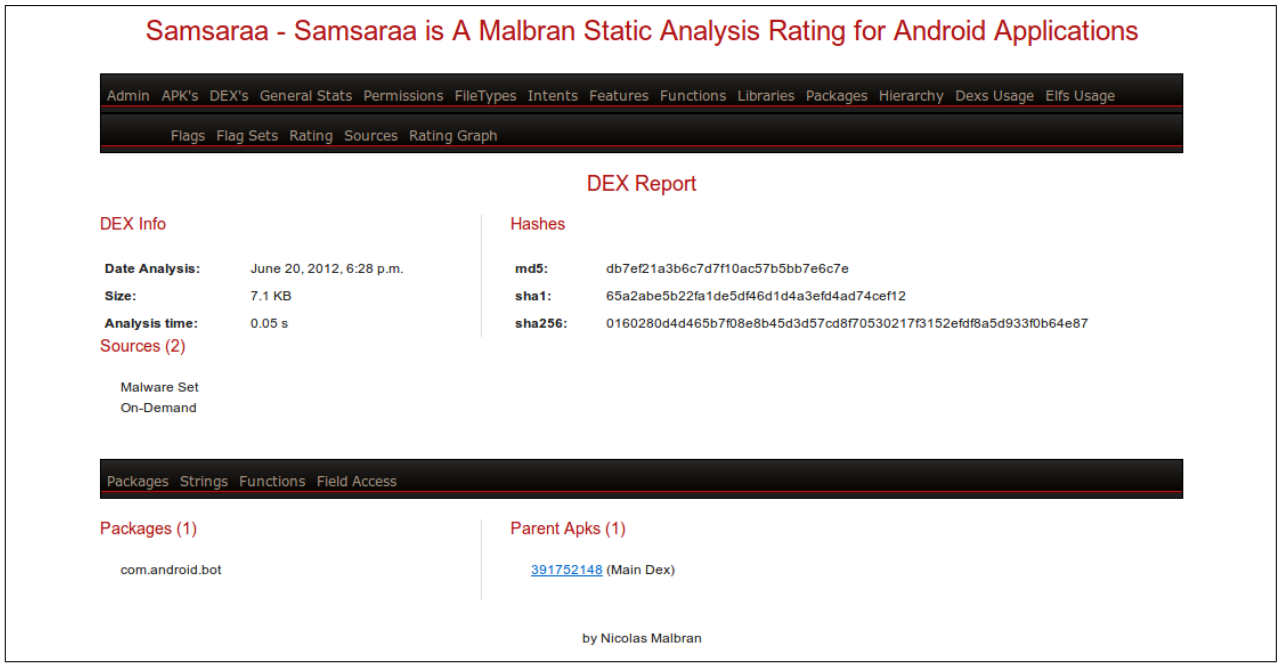


Figura C.1: Ejemplo de un informe de packages de un dex.

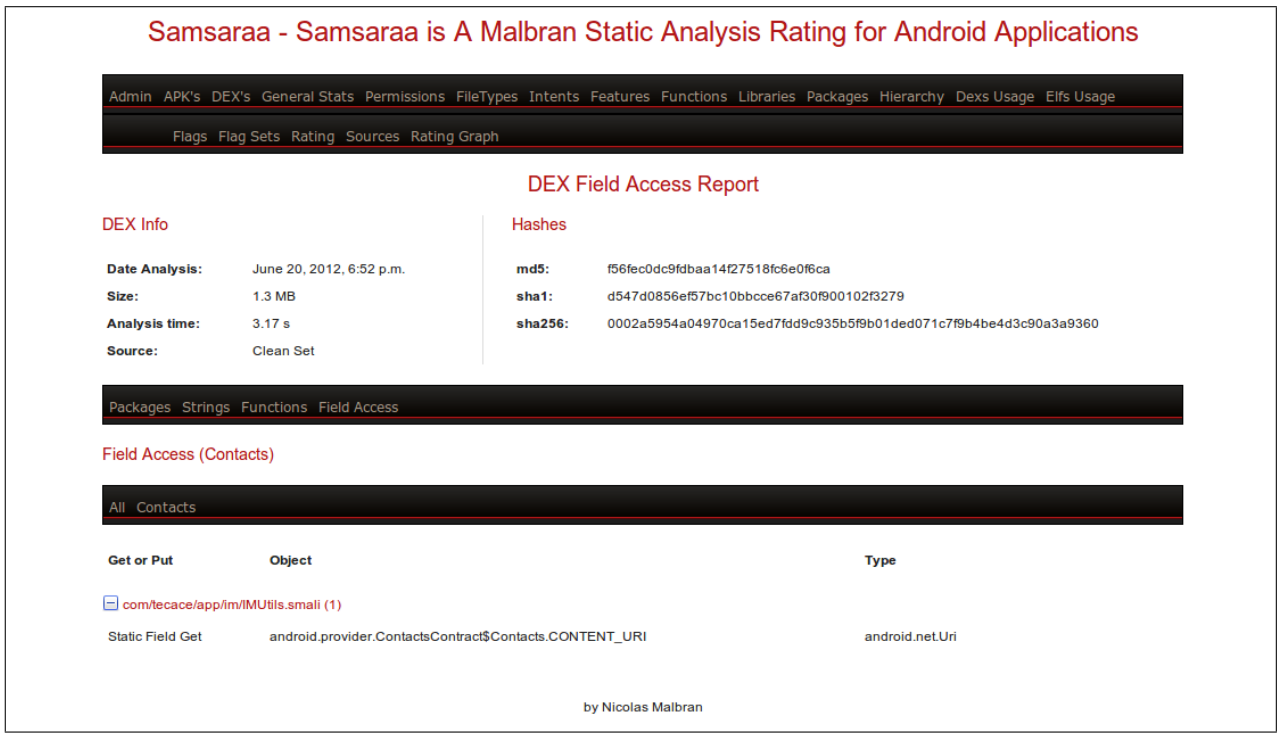
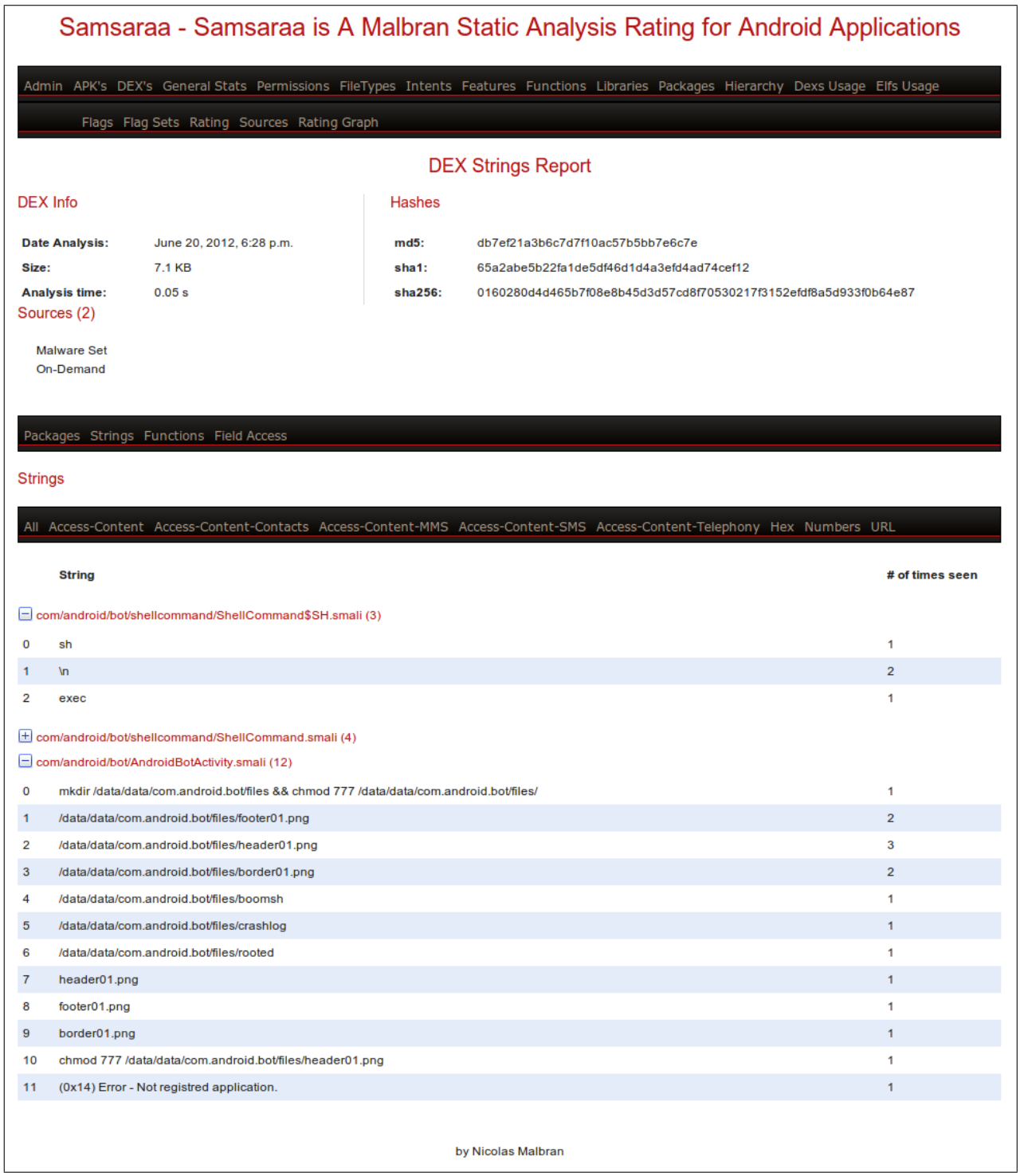


Figura C.2: Ejemplo de un informe de accesos de campo de un dex.



Samsaraa - Samsaraa is A Malbran Static Analysis Rating for Android Applications

Admin APK's DEX's General Stats Permissions FileTypes Intents Features Functions Libraries Packages Hierarchy Dexs Usage Elfs Usage

Flags Flag Sets Rating Sources Rating Graph

DEX Strings Report

DEX Info

Date Analysis:	June 20, 2012, 6:28 p.m.
Size:	7.1 KB
Analysis time:	0.05 s

Hashes

md5:	db7ef21a3b6c7d7f10ac57b5bb7e6c7e
sha1:	65a2abe5b22fa1de5df46d1d4a3efd4ad74cef12
sha256:	0160280d4d465b7f08e8b45d3d57cd8f70530217f3152efdf8a5d933f0b64e87

Sources (2)

Malware Set
On-Demand

Packages Strings Functions Field Access

Strings

All Access-Content Access-Content-Contacts Access-Content-MMS Access-Content-SMS Access-Content-Telephony Hex Numbers URL

	String	# of times seen
⊟ com/android/bot/shellcommand/ShellCommand$SH.smali (3)		
0	sh	1
1	\n	2
2	exec	1
⊞ com/android/bot/shellcommand/ShellCommand.smali (4)		
⊟ com/android/bot/AndroidBotActivity.smali (12)		
0	mkdir /data/data/com.android.bot/files && chmod 777 /data/data/com.android.bot/files/	1
1	/data/data/com.android.bot/files/footer01.png	2
2	/data/data/com.android.bot/files/header01.png	3
3	/data/data/com.android.bot/files/border01.png	2
4	/data/data/com.android.bot/files/boomsh	1
5	/data/data/com.android.bot/files/crashlog	1
6	/data/data/com.android.bot/files/rooted	1
7	header01.png	1
8	footer01.png	1
9	border01.png	1
10	chmod 777 /data/data/com.android.bot/files/header01.png	1
11	(0x14) Error - Not registred application.	1

by Nicolas Malbran

Figura C.3: Ejemplo de un informe de strings de un dex.

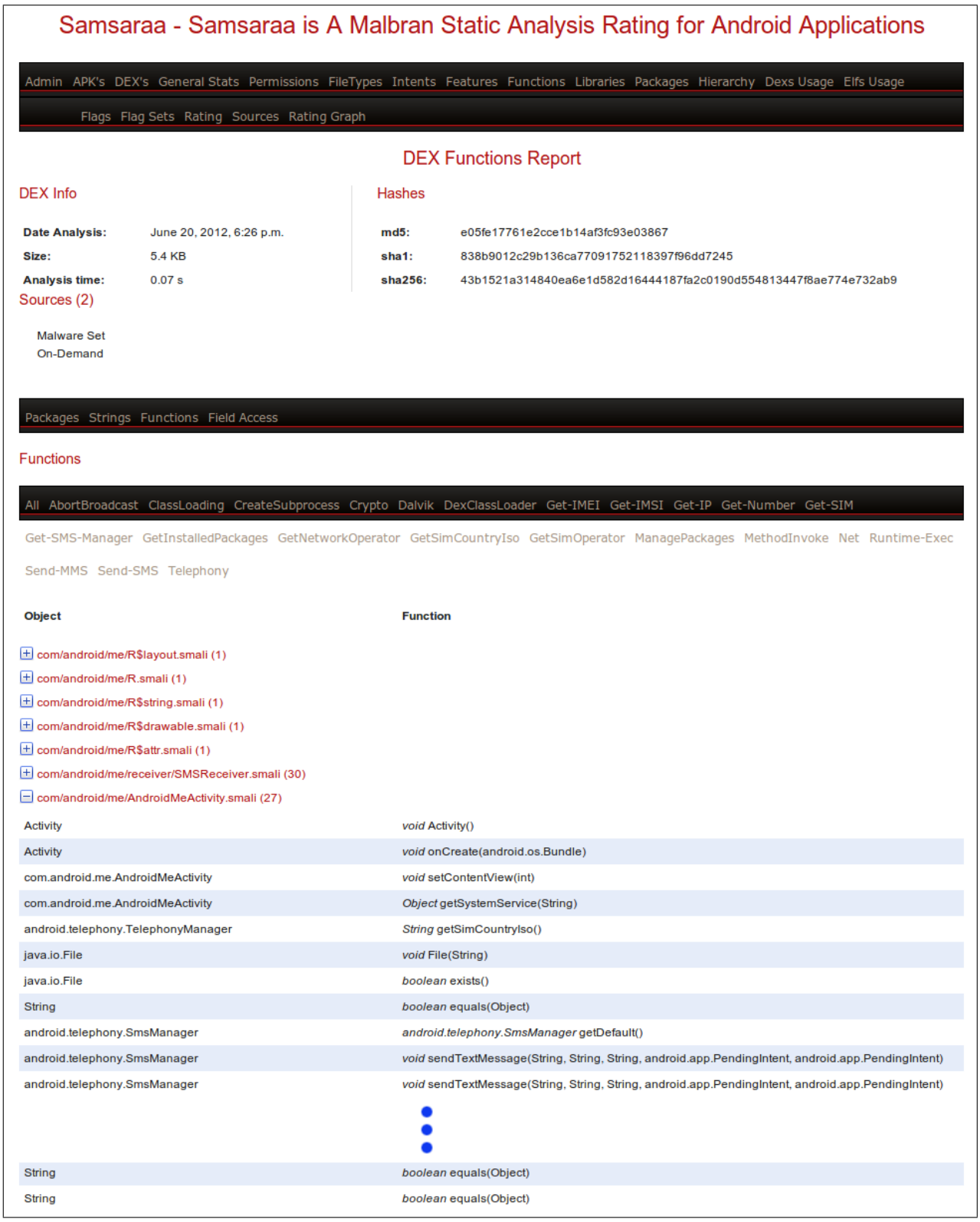


Figura C.4: Ejemplo de un informe de funciones de un dex.

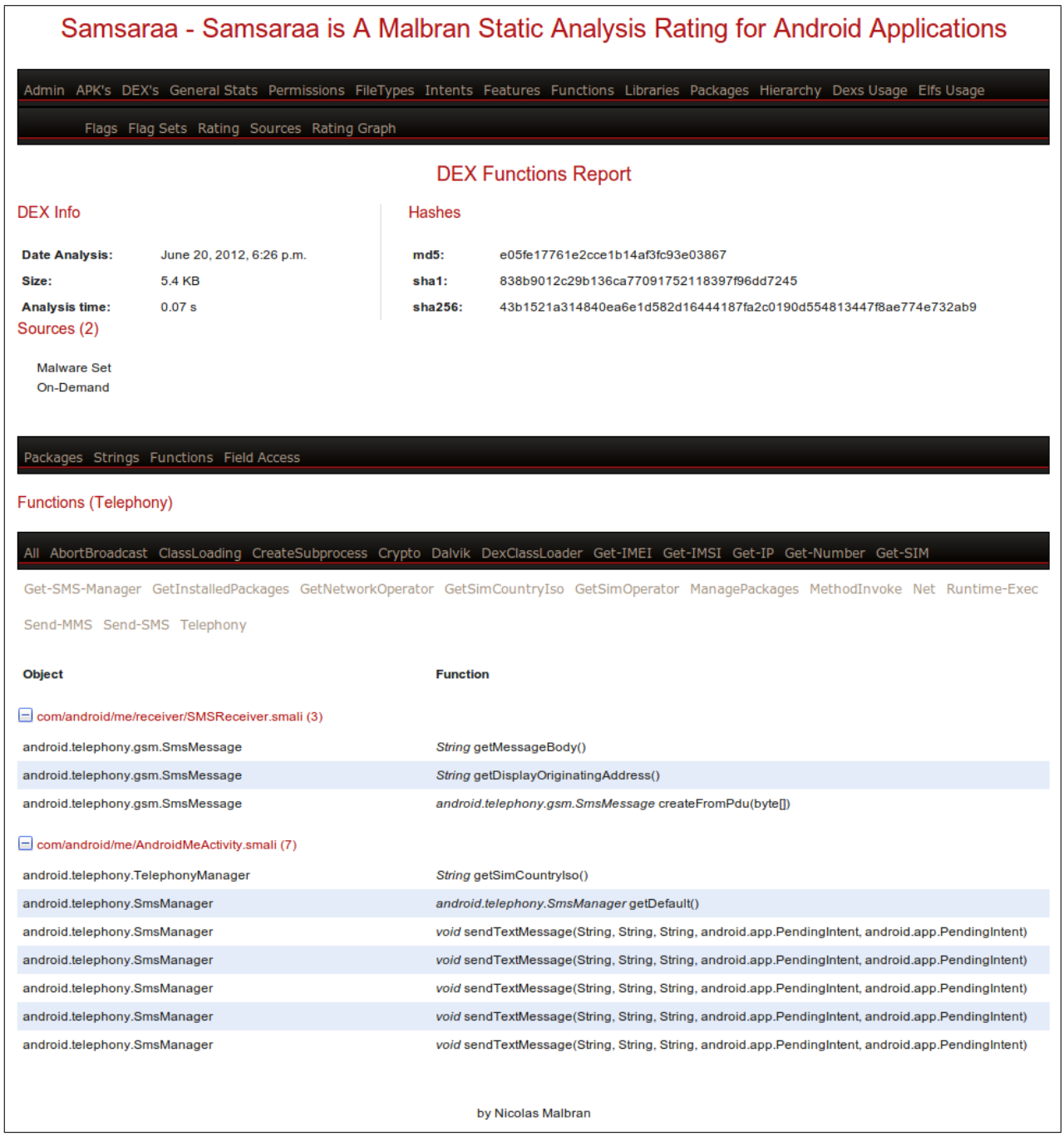
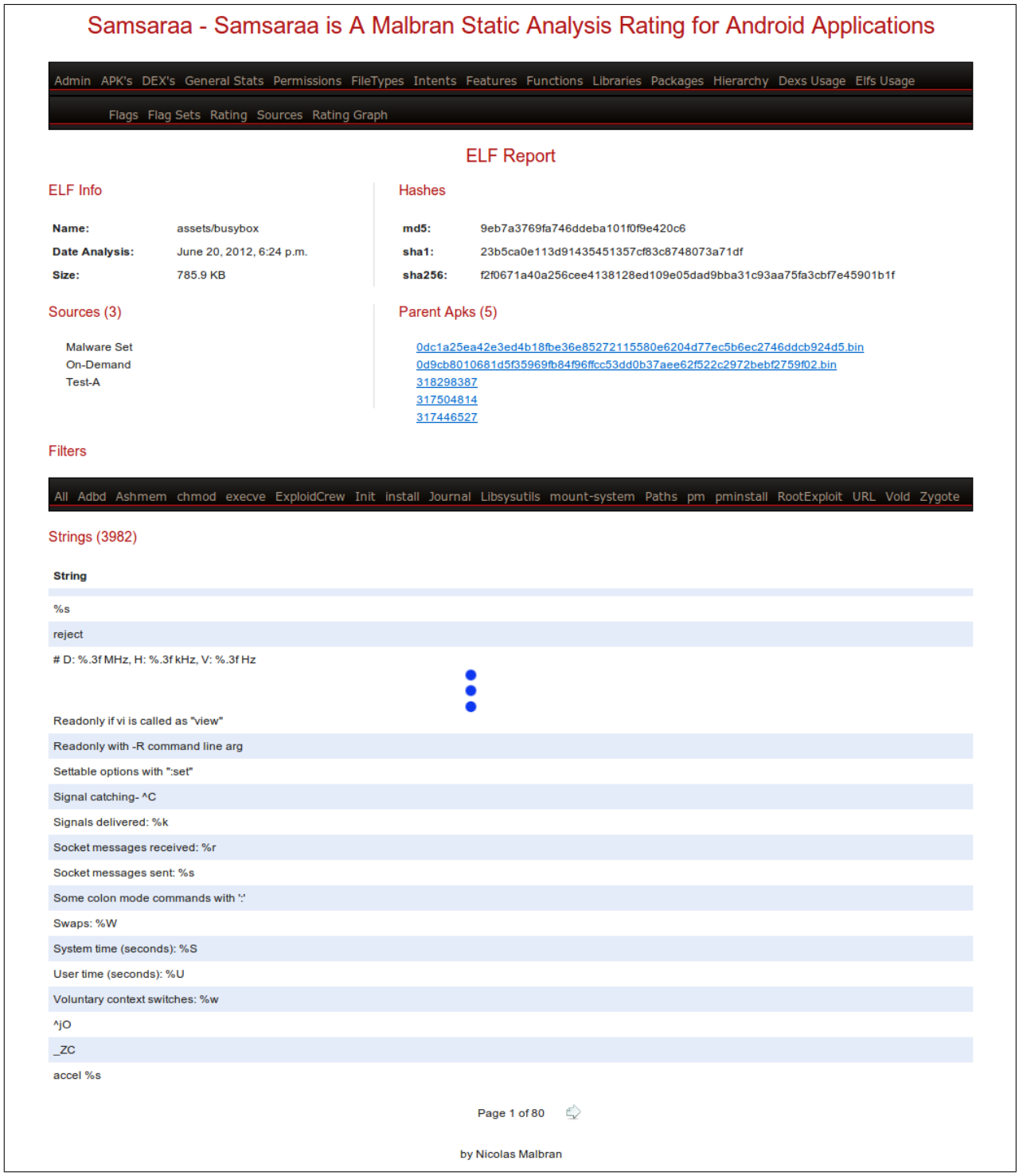


Figura C.5: Ejemplo de un informe de funciones filtradas de un dex.

Apéndice D Informe de análisis de un elf



Samsaraa - Samsaraa is A Malbran Static Analysis Rating for Android Applications

Admin APK's DEX's General Stats Permissions FileTypes Intents Features Functions Libraries Packages Hierarchy Dexs Usage Elfs Usage

Flags Flag Sets Rating Sources Rating Graph

ELF Report

ELF Info

Name:	assets/busybox
Date Analysis:	June 20, 2012, 6:24 p.m.
Size:	785.9 KB

Hashes

md5:	9eb7a3769fa746ddeba101f0f9e420c6
sha1:	23b5ca0e113d91435451357cf83c8748073a71df
sha256:	f2f0671a40a256cee4138128ed109e05dad9bba31c93aa75fa3cbf7e45901b1f

Sources (3)

Malware Set
On-Demand
Test-A

Parent Apks (5)

0dc1a25ea42e3ed4b18fbe36e85272115580e6204d77ec5b6ec2746ddcb924d5.bin
0d9cb8010681d5f35969fb84f96ffcc53dd0b37aee62f522c2972bebf2759f02.bin
318298387
317504814
317446527

Filters

All Adbd Ashmem chmod execve ExploidCrew Init install Journal Libsysutils mount-system Paths pm pminstall RootExploit URL Vold Zygote

Strings (3982)

String
%s
reject
D: %.3f MHz, H: %.3f kHz, V: %.3f Hz
Readonly if vi is called as "view"
Readonly with -R command line arg
Settable options with ":set"
Signal catching- ^C
Signals delivered: %k
Socket messages received: %r
Socket messages sent: %s
Some colon mode commands with ':'
Swaps: %W
System time (seconds): %S
User time (seconds): %U
Voluntary context switches: %w
^jO
_ZC
accel %s

Page 1 of 80

by Nicolas Malbran

Figura D.1: Ejemplo de un informe de análisis para elf.