CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO:

Diseño e Implementación de un Módulo para la Detección de Aplicaciones Móviles Maliciosas en Mercados Online

Autor: Silvia Sebastián González

Director: Manuel Carro Liñares
Cotutor: Juan Caballero

MADRID, JUNIO 2017

Resumen

En este Trabajo de Fin de Grado se pretenden realizar unos algoritmos capaces de detectar aquellas personas que tienen más de una cuenta de desarrollador en Google Play Console. La motivación es ser capaces de, una vez identificado que un desarrollador es malicioso, eliminar todas las aplicaciones que haya subido sin importar desde qué cuenta lo hiciera.

Palabras clave: metadatos, análisis, clustering, código malicioso, aplicaciones no deseadas, desarrollador, publisher, identificador de una aplicación.

Abstract

On the present Final Term Project the aim is to create some algorithms that could be able to find out developers that have more than one account on Google Play Console. The main motivation is being able to identify all the apps that belongs to a malicious developer, no matter how many accounts he had.

Keywords: metadata, analysis, clustering, malware, grayware, developer, publisher, package name.

Agradecimientos

Me gustaría agradecer...

A mis padres, por su apoyo durante todos estos años. Por enseñarme a que una persona es mucho más que sus resultados académicos.

A María, sujeto de absoluta admiración desde la infancia.

A mi familia. A los que están y a los que faltan. Os llevo siempre conmigo.

A Dani, por saber ser mi oasis frente al desierto.

A Sergio, por ser quien siempre y sin condiciones me saca del pozo. A Anaïs, por transmitir la perfección en cada cosa que toca.

A Histrión, una gran familia que no debería desaparecer.

A mi colegio, por ser capaz de formar una mente pensante.

A la Facultad de Informática, por darme todas esas alegrías y desesperaciones que conforman la informática.

Al IMDEA Software y, en especial a Juan Caballero, que han hecho posible la realización de este proyecto.

Índice de Contenidos

Índice de Tablas

Índice de Figuras

Figura **Página**

1

Introducción

En este Trabajo de Fin de Grado se pretenden realizar unos algoritmos que demuestren si dos aplicaciones alojadas en Google Play con nombres de desarrollador distintos son publicados, en definitiva, por la misma persona. Esta posibilidad nace de la manera en la que una persona se da de alta como desarrollador de aplicaciones de Google Play.

Para registrarse como tal, lo primero que se debe hacer es vincular una cuenta de *gmail* a una de Google Play Console. Para ello, debe aportar datos personales como nombre de usuario, datos de contacto e información bancaria para efectuar el pago de la cuenta. A partir de ese momento puede subir APKs (del inglés *Android Application Package*; es decir, *Aplicación Empaquetada de Android*) las cuales contienen el código ejecutable de aplicaciones Android.

Una misma persona puede crearse más de una cuenta de desarrollador con el mismo o diferentes nombres de usuario. Como la única información visible de los desarrolladores es la que aparece en la página del market, no se tiene ninguna

posibilidad de comparar los identificadores que incluye en sus cuentas personales de Google Play Console.

Las razones por las que una persona puede querer generar varias cuentas pueden ser variadas. Sin embargo, este proyecto se centrará en las razones maliciosas. Se parte de la hipótesis de que un desarrollador puede generar varias cuentas para alojar en ambas aplicaciones maliciosas con diferente apariencia exterior. De esta manera, si una de ellas resulta reportada, puede seguir manteniendo la otra en el mercado para continuar con sus actividades.

Como se puede comprobar, el caso expuesto asume una simultaneidad temporal, por lo que si se detecta un desarrollador malicioso sería idóneo poder relacionarlo con sus otras cuentas para poder suspender todas a un mismo tiempo. Sin embargo, se pueden encontrar casos de no simultaneidad temporal: un desarrollador podría decidir generar una nueva cuenta tras comprobar que la primera ha sido inutilizada para así subir de nuevo la aplicación (a este subconjunto de aplicaciones las hemos denominado *reborn*). Por tanto, lo ideal en este caso sería que, cada vez que un desarrollador intenta subir una aplicación, se fuera capaz de identificar si es el mismo desarrollador que otro previamente eliminado por ser malicioso.

Las soluciones propuestas para identificar estos casos se resumen en la obtención de una función capaz de determinar si dos cuentas con nombre de desarrollador distinto son mantenidas por la misma persona. Dicha función puede basarse en la comparación de aplicaciones de ambos desarrolladores (tanto sus metadatos como el código extraído del APK) ya que si se encuentran dos aplicaciones iguales en cuentas distintas queda demostrado el hecho de que tras esos nombres se encuentra la misma persona. Otra forma de generar esta función puede realizarse comparando metadatos de alto nivel; los cuales no deben ser confundidos con los mencionados previamente. En este caso se refiere a los metadatos extraídos de los

certificados, al developer name e información extraída del package name (es decir, del identificador unívoco de la aplicación).

Gracias a esta función se puede realizar un problema más complejo en el que el input no son dos nombres de desarrolladores y se busca un veredicto booleano (o son el mismo o no lo son y con qué distancia), sino que se busca obtener ejemplos de desarrolladores que cumplan estas características. Para ello, un primer enfoque consiste en tomar como muestra publishers o desarrolladores conocidos por no ser del todo fiables en otros mercados y comprobar si también tienen actividades en Android. De ser así, se puede empezar realizando clusterings con dicho input. La estrategia para agrupar a los desarrolladores será utilizar la función descrita anteriormente para calcular la distancia entre todos los pares de developers y así generar un patrón que permita emplear algoritmos de clustering.

Para realizar este proyecto es preciso contar con una base de datos de aplicaciones lo suficientemente grande para poder evaluar los módulos implementados. En este caso, la base de datos que da soporte a este trabajo se ha realizado gracias a una herramienta denominada Tacyt, desarrollada en Telefónica gracias al departamento especializado en ciberseguridad llamado ElevenPaths. En ella se encuentran las aplicaciones y versiones presentes (o que han alojado en el pasado aunque ahora estén eliminadas) en diversos mercados, tanto en plataformas iOS (AppleStore) como Android (PlayStore, mobogenie, aptoide, userUpload).

A pesar de que Tacyt contiene información de varios mercados, la decisión de centrarse en sólo Google Play se debe a que el estudio que se pretende hacer es de plataformas Android y quien domina dicho mercado es Google. Es esta la razón por la que Tacyt contiene muchos más datos de este mercado. Además, la posibilidad de descargar las aplicaciones que no son de pago permite realizar un análisis más profundo en caso de que sea necesario.

La información que ofrece de las aplicaciones se obtiene de dos maneras: o extrayendo los metadatos que ofrece Google en la página de descarga de las aplicaciones o analizando el código de la aplicación para obtener datos como los certificados, nombres de directorios y ficheros y realizar análisis estáticos o reportes de VirusTotal.

2

Estado del Arte

Para la elaboración de este estudio es imprescindible revisar soluciones parecidas y herramientas desarrolladas previamente que resulten útiles para el análisis que nos ocupa. En un primer momento eran interesantes aquellos trabajos que hablaban sobre cómo saber encontrar aplicaciones maliciosas o grayware, tanto observando sólo los metadatos como atendiendo también el código. Una de las soluciones planteadas [10] identifica el malware mediante footprinting gracias a métodos heurísticos, los cuales hacen posible que, aun no teniendo almacenada en su base de datos un malware determinado, sean capaces de detectarlo gracias a la modelización del comportamiento de diferentes familias de código malicioso. Este algoritmo conlleva un especial cuidado a la hora de no cometer falsos positivos ya que las aplicaciones son muy variadas y podría suceder que una aplicación benigna solicite unos permisos justificados que no se esperen.

Éste no es el único trabajo originado con el fin de encontrar aplicaciones maliciosas. Otro ejemplo de esto es DREBIN [2], el cual permite encontrar malware directa-

mente desde el propio dispositivo que sospechamos que puede estar infectado. Lo que lo diferencia de otros antivirus es su poco peso (característica vital para los usuarios de smartphones, ya que el hecho de que por norma general los antivirus convencionales consumen muchos recursos del teléfono ocasiona que caigan en desuso). Este rasgo distintivo se debe a que para detectar malware se realiza un potente análisis estadístico capturando la mayor cantidad posible de features de las aplicaciones. Dichos features se incluyen en un vector y se buscan patrones semejantes a otros identificados como malware. Esta herramienta funciona notablemente bien ya que en una evaluación en un repositorio de 123.453 aplicaciones con 5.560 muestras de malware, este algoritmo era capaz de encontrar 94% del malware con apenas falsos positivos (1%). Es decir, funciona tan bien como 9 de los 10 antivirus más populares.

Una práctica bastante frecuente es no ubicar el código malicioso directamente en la aplicación, sino que se encuentre en las librerías que incluye. Unas de las características utilizadas en este aspecto son las conexiones y permisos VPN [6] ya que pueden ser explotadas mediante manipulación del tráfico mientras se hace creer al usuario que trabaja en un ambiente seguro. El problema surge al emplear protocolos de tunneling inseguros, uso de IPv6 o filtraciones de tráfico DNS. Tras la realización de un análisis de 283 aplicaciones VPN en Android se obtuvieron 1.4M de aplicaciones en GooglePlay que las empleaban. Los resultados revelaron que aunque un 67% de ellas mantenían la privacidad online, 75% de ellas usaban librerías para hacer tracking y el 82% requería permisos para acceder a información sensible de los usuarios.

Otras librerías, sin embargo, vuelven a ser empaquetadas para, de esta manera, se propague malware al usarlas [3]. Estas librerías son conocidas como PhaLibs . Los estudios de este tipo son bastante complejos, ya que el código de las librerías

no siempre es accesible (particularmente en sistemas iOS). Como el binario no es fácilmente extraíble, esto redunda en que la mayoría de los antivirus no analizan las librerías, lo que favorece la proliferación de este tipo de código malicioso.

Al igual que existen librerías "potencialmente maliciosas", también podemos encontrar PHAs (Potentially Harmful Apps); es decir, aplicaciones potencialmente maliciosas o también denominadas Grayware [1]. Como su nombre indica, este tipo de aplicaciones entra en una escala de grises, ya que no conlleva un ataque al dispositivo propiamente, pero sí se pueden producir efectos no deseados por el usuario (de acuerdo a la privacidad, incorrecto desempeño de la función que prometían realizar o disminución de la eficiencia del dispositivo). Según el daño producido y sus características, se han clasificado en grandes grupos:

1. Impostores: se hacen pasar por otra aplicación con la finalidad de obtener más ingresos gracias a las descargas. A veces llegan a adoptar en el mercado donde se alojan el mismo título e imagen.

2. Misrepresentors: afirman proveer de unas determinadas funcionalidades a los usuarios cuando en realidad tan sólo las simula. Son casos de algunos de los "antivirus gratuitos" que fingen realizar un escaneado del dispositivo (dando una falsa sensación de seguridad) o aquellos que tratan de convencer que con la cámara de su dispositivo pueden visualizar imágenes mediante rayos X.

3. Madware: son aplicaciones con una cantidad agresiva de publicidad que dificultan mucho la experiencia de usuario. Llega a haber aplicaciones que no tienen ningún contenido, presentan un anuncio tras otro continuamente.

4. Dialers: envían mensajes y realizan llamadas sin que el usuario tenga conocimiento de ello. Las razones pueden ser variadas, tanto para llamar a teléfonos de pago como para molestar a la lista de contactos.

5. Bromas: causan interferencias con otras aplicaciones o irritan al usuario. Normalmente son instalados por alguien cercano al propietario del dispositivo sin que este último tenga conocimiento de ello.

6. Scareware: se intenta provocar miedo o ansiedad en el usuario con la finalidad de provocar en él una reacción. Si esa información proporcionada al usuario es mentira nos encontramos ante grayware.

7. Herramientas de rooting: permiten al usuario ganar privilegios de root en su dispositivo. Al violar los requisitos de seguridad Google los cataloga como PHA.

8. Spyware o trackware: recogen datos de los usuarios y los coleccionan sin la autorización del usuario.

9. RATs (Herramientas de Acceso Remoto): pueden controlar remotamente un dispositivo. No tienen por qué sostener finalidades maliciosas, pero potencialmente podrían realizarlas.

10. Droppers: instalan aplicaciones adicionales sin el consentimiento del usuario.

11. Hijackers: manipulan el sistema o los ajustes de las aplicaciones para redirigir al usuario a otra localización.

Sin embargo, esta no es la única clasificación posible para catalogar las aplicaciones y ser capaz de agruparlas de acuerdo a nuestros objetivos. Atendiendo a las descripciones de las aplicaciones, se pueden clasificar de acuerdo a los tópicos que

comparten. De esta manera es más fácil encontrar entre ellas aplicaciones que se hayan impersonado. Esta solución es la que resuelve la herramienta CHABADA [5]. Para ilustrar esta afirmación propondré un ejemplo: si una aplicación trata sobre viajes, es perfectamente entendible que se precise saber la localización del dispositivo por si es necesaria la ubicación en algún mapa. Sin embargo, si el tema de la aplicación trata sobre el intercambio y mantenimiento de ficheros, no es tan frecuente que se precise un permiso de ubicación. Por tanto, atendiendo a la agrupación a la que pertenecen, se pueden definir los permisos y características más frecuentes de las aplicaciones benignas y aquellos casos que se desvíen sean los que se sometan a estudio.

Estos casos de aplicaciones PHA no están presentes sólo en aplicaciones móviles. Este tipo de programas han sido evaluados anteriormente como PUP [7][8] (Potentially Unwanted Programs). Estos se proliferan en lo que se conoce como servicios PPI (Pay-Per-Install; es decir, de pago por cada instalación) esto se debe a que como desean monetizar su aplicación se afilian a publishers para aparecer como parte de un paquete con otra aplicación al descargarse. La forma de identificarlos se realiza mediante los certificados que firman dichas aplicaciones [9]. A pesar de que este estudio se ha realizado para identificar los PUP en Windows, se podría emular un análisis similar en una plataforma Android, ya que los certificados son de vital importancia en dicha plataforma. Por ejemplo, si una aplicación quiere actualizar su versión es preciso que sea bajo el mismo certificado, si no deberá cambiar su package name. Además, el hecho de que la mayoría son autofirmados puede dar aún más información de los desarrolladores.

3

Definición del Problema

La problemática central de la que este proyecto se ocupa es de la identificación de desarrolladores distintos de Google Play que, en realidad, son una misma persona. Esta situación puede producirse debido a que cualquier persona puede tener varias cuentas en Google Play Console, plataforma específica para los desarrolladores de aplicaciones de Google.

Las razones que puede tener un desarrollador para querer cambiar su nombre y seguir subiendo aplicaciones son variadas. Sin embargo, la realmente importante para este campo de investigación es la de aquellos que pretenden burlar el sistema de análisis de Google: cuando una aplicación es eliminada debido a su condición de malware, el desarrollador podría tener otra cuenta con la misma aplicación (si no con unos mínimos cambios) para así continuar con sus fines.

Esto refleja que, si se fuera capaz de identificar si una aplicación recién subida al mercado es la misma que una aplicación ya señalada como maliciosa, no se tendría que realizar un análisis del código y/o del comportamiento de la aplicación. Además,

si se sabe que el desarrollador suele subir malware, ser capaz de relacionar los nombres de los desarrolladores con las personas reales que los usan, podría ayudar identificar a aquellos que intentan enmascarar su identidad para burlar los análisis de Google.

De esta manera, se puede partir de dos grandes escenarios:

1. Se conocen dos *developer name* y se pretende discernir si son el mismo desarrollador.

   (a) Identificación a partir de comparación de metadatos de alto nivel:

   Compuestos por información de los certificados, developer name, datos sobre análisis de antivirus y package name.

   (b) Identificación a partir de comparación de aplicaciones:

   Se puede realizar de dos maneras: mediante comparación de métodos del código fuente o comparando metadatos de las aplicaciones. Esta segunda manera no se debe confundir con el caso 1.a) ya que estos metadatos comprenden otros aspectos de la aplicación como el título, la descripción o los permisos de la misma.

2. Encontrar ejemplos de personas con varias cuentas en GooglePlay (es decir, realizando búsqueda amplia en el dataset y una posterior clusterización).

En una tercera instancia, si se quisiera determinar cuáles de ellas son específicamente *reborn* (es decir, aquellas aplicaciones que vuelven a aparecer cuando se creían eliminadas), se puede hacer un análisis temporal de las cuentas y aplicaciones que respaldan los inputs analizados.

4

Datasets: Tacyt

Para poder realizar el estudio se hace imprescindible la presencia de datasets que permitan corroborar las hipótesis planteadas, así como testear los módulos que se implementen para comprobar que las clusterizaciones que se realizan son correctas. De esta manera, se ha contado con una herramienta en la que se alojan múltiples aplicaciones de diferentes mercados y con una base de datos de conocidos publishers de *aplicaciones grayware* de sistemas Windows.

4.1 Tacyt

La cantidad de aplicaciones que alojan los servidores de mercados Android (como GooglePlay) tiene unas proporciones realmente grandes. Por esta razón, pocos repositorios logran alojar todas las aplicaciones tanto presentes en ese momento en el mercado, como las aplicaciones ya eliminadas. Por lo tanto, hacer un análisis exhaustivo de todo lo concerniente a dicho store no es sólo ambicioso por las propor-

ciones del problema, sino por la falta de herramientas. Este impedimento hemos podido solventarlo gracias a Tacyt, desarrollado por Eleven Paths en Telefónica.

Esta tecnología almacena un total de 7.684.465 versiones (no estando 1.854.275 versiones disponibles ya en las tiendas, es decir un 24.13% de las aplicaciones han sido eliminadas). Los mercados de los que dispone información son de sistemas iOS (Apple Store) y Android (uptoide, mobogenie, userUpload y GooglePlay). Para comenzar el estudio, si bien investigar sobre iOS era interesante debido al poco trabajo de análisis que hay sobre este tema (en comparación con Android), se desechó enseguida ya que no permitía descargar el código fuente de las aplicaciones y no se disponía de tantos metadatos como en los otros mercados. De esta manera, iOS quedaba relegada a una posterior investigación para centrar el estudio en Android.

Desde un primer momento la atención se dirigía a aquellas aplicaciones que habían sido eliminadas, ya que ser capaces de discernir si habían sido decisión de los mercados o del desarrollador podría arrojar luz sobre aquellas aplicaciones maliciosas que logran ser detectadas. Llevando a cabo un estudio preliminar de dichos mercados, se pudieron obtener algunas características de las aplicaciones que contenían.

Antes de mostrar algunos datos de los mercados señalados anteriormente, es importante señalar cómo están almacenadas las aplicaciones. Cuando un desarrollador sube una aplicación nueva, ésta es registrada en Tacyt incluyendo su día de creación (es decir, desde qué día se alojó en dicho mercado) y día que la encontró Tacyt; además de otros datos. Si se desea solucionar un error de esa misma aplicación pero sin modificar la versión, se modifica la fecha de actualización. Sin embargo, si se sube una aplicación diferente, si bien en los mercados la versión anterior desaparece y la nueva es la única accesible, Tacyt almacena ambas: las señala

como versiones de una misma aplicación pero se puede acceder al código y los datos como si de dos aplicaciones diferentes se tratase. Aunque la versión antigua ya no sea accesible en los mercados, no se incluye una fecha de muerte hasta que el package name de la aplicación no sea eliminada o deje de estar mantenida.

4.1.1 Apple Store

La base de datos cuenta con 2.549.854 versiones de aplicaciones de Apple Store. De ellas, están muertas 678.967; es decir, un 26.63%. Para poder mostrar una mejor idea del comportamiento de este mercado, se ha extraído información de su *lifetime*; es decir, del tiempo (en días) que suele permanecer una aplicación antes de ser eliminada:

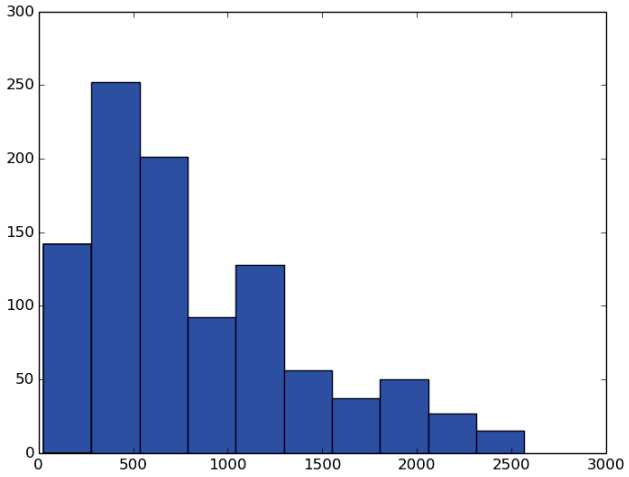

Figura 4.1: Lifetime de las aplicaciones de AppleStore de una muestra de 1.000 aplicaciones

Se puede observar que hay un elevado número de aplicaciones que desaparecen de la tienda antes de los 500 días; es decir, un gran porcentaje de las aplicaciones que son eliminadas tienen un lifetime de alrededor de un año.

Sin embargo, no se pueden obtener rápidamente resultados de esas aplicaciones ya que en Tacyt no se puede obtener el código fuente de las aplicaciones provenientes de iOS, por lo que no se puede determinar cuántas serializaciones de antivirus son positivas en VirusTotal ni realizar un análisis manual de la aplicación. A pesar de ello, sí ofrece bastantes metadatos para poder realizar estadísticas del comportamiento de la AppleStore que se podrá realizar en un trabajo futuro.

4.1.2 UserUpload

De este mercado es del que se obtiene el menor número de aplicaciones: 3.466 versiones. Las versiones eliminadas en este caso son 12, tan sólo un 0.35% del total.

Para hacernos una idea de los resultados obtenidos, mostraremos de nuevo el lifetime de las aplicaciones eliminadas:

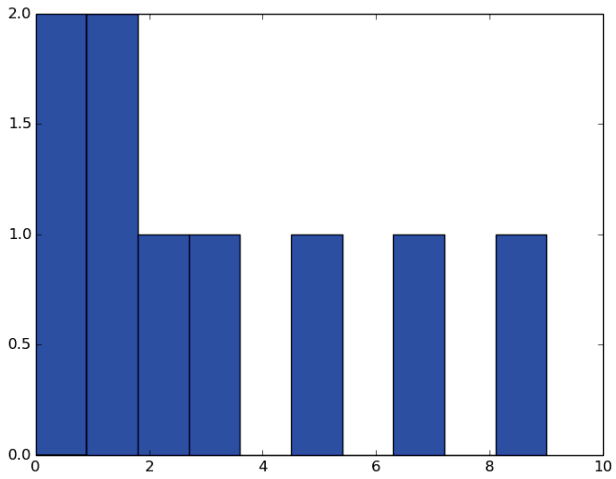

Figura 4.2: Lifetime de las aplicaciones de UserUpload de una muestra de 12 aplicaciones

La gran mayoría de ellas son eliminadas en el mismo día o al siguiente, no llegando a sobrevivir ninguna más de nueve días. Si atendemos a la información que se puede extraer de VirusTotal, observamos que 6 aplicaciones tienen más de tres

detecciones de antivirus, por lo que son consideradas maliciosas. Otras tres ofrecen 1 o 2 positivos, por lo que no se puede afirmar con precisión que contengan malware. Las últimas 3 aplicaciones son corruptas, por lo que no se puede extraer ninguna información de ellas.

Por tanto, parece que el comportamiento de la tienda en este caso es realizar un análisis en los primeros días que está subida una aplicación y, de resultar maliciosa, es eliminada. De no dar positivo, se mantiene en la tienda sin volver a ser comprobado si son benignas.

4.1.3 Mobogenie

Tiene 340.880 versiones en Tacyt, estando muertas tan sólo 151. Es decir, un 0.04% de ellas. Esta cantidad también es muy baja, por lo que se observa su lifetime:

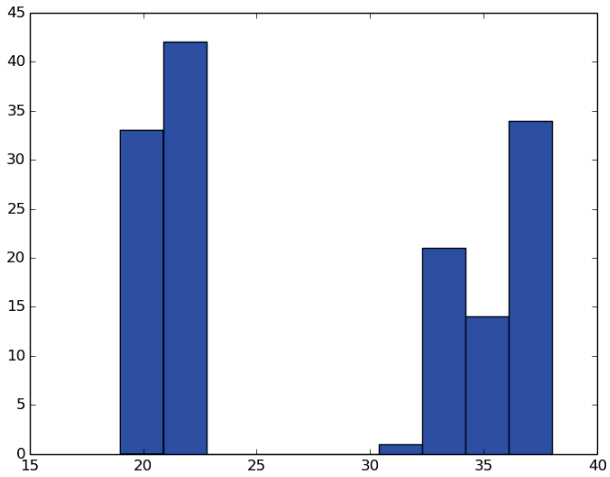

Figura 4.3: Lifetime de las aplicaciones de Mobogenie de una muestra de 144 aplicaciones

Estas aplicaciones permanecen menos de 40 días en servicio, lo cual hace pensar que pueden suceder dos cosas: o bien hacen un análisis de las apk cuando son subidas

a la plataforma y luego dejan de hacerlo, o eran ficheros erróneos o versiones que no deseaban ser mantenidas de aplicaciones y, por tanto, debían ser eliminadas.

Si observamos los metadatos de Tacyt, 47 de las 151 (un 32.6% de las aplicaciones obtenidas) han dado positivo en los análisis de VirusTotal, por lo que se demuestra la hipótesis del único análisis en un tercio de los casos.

4.1.4 Aptoide

De este mercado hay 271.269 versiones. De ellas, no hay ninguna eliminada. Esto hace imposible identificar su lifetime, ya que no llegan a desaparecer.

Observando los resultados se llega a la conclusión de que la política de Aptoide no contempla análisis periódico de las aplicaciones que alojan o que los desarrolladores no se preocupan en mantener las aplicaciones.

4.1.5 GooglePlay

Este es el mercado con una mayor cantidad de versiones alojadas así como una gran variedad de metadatos y posibilidad de obtener el código fuente de todas aquellas aplicaciones que no sean de pago. El gran contenido le convierte en un objetivo claro para nuestro estudio.

El número total de versiones que se pueden obtener son 4.519.081, siendo 1.175.303 de ellas eliminadas (26.01%). Para realizar un muestreo inicial se obtienen 201 aplicaciones, resultando el siguiente lifetime:

Son eliminadas alrededor de los primeros 200 días; es decir, la gran mayoría de aplicaciones acaba desapareciendo en un año más o menos (tal y como sucedía en Apple Store).

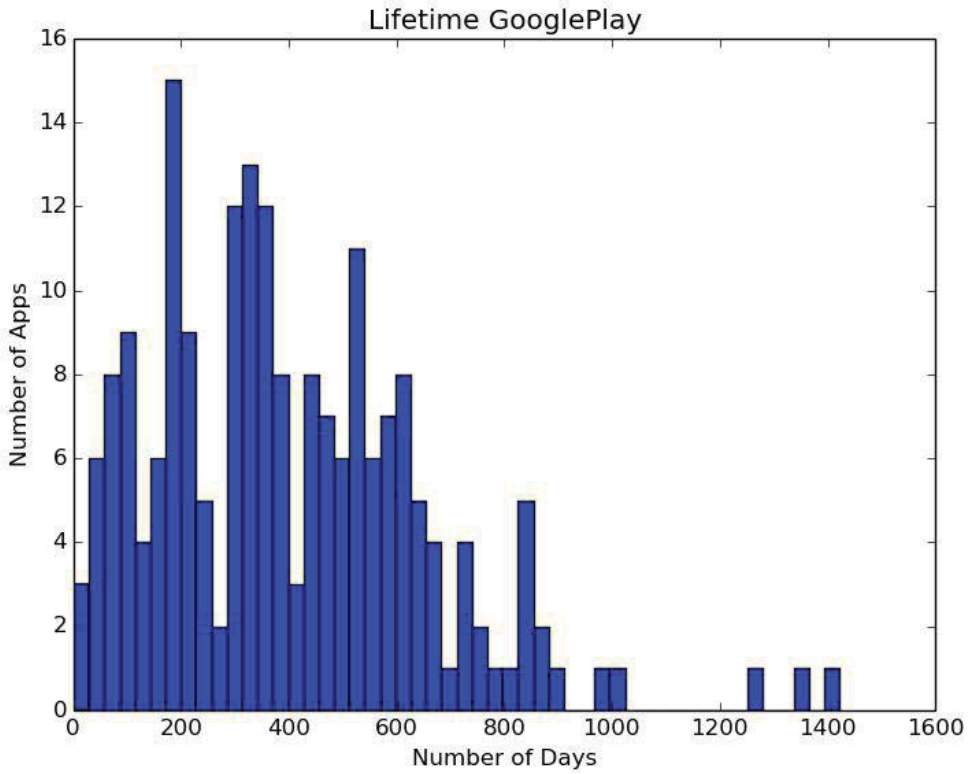

Figura 4.4: Lifetime de las aplicaciones de Google Play de una muestra de 201 aplicaciones

En ocasiones hay metadatos sobre la fecha en la que se ha realizado un análisis en VirusTotal. En este caso, esa fecha la tienen 245 versiones de 115 aplicaciones. De todas ellas, sólamente el 71 versiones recuperan la serialización del análisis (sólo un 28.98% de las analizadas por VirusTotal). Las 174 versiones restantes no han dado ningún positivo en la fecha en la que se realizó el análisis. Sin embargo, hay aplicaciones que, tras volver a analizarlas actualmente, comienzan a dar positivos tras la mejora de los analizadores con el paso del tiempo.

Si atendemos a la gráfica que muestra el número de antivirus que devuelven resultados positivos por aplicación de la Figura 5.5, se puede apreciar que la gran mayoría de las aplicaciones dan menos de 3 positivos (por lo que no se puede afirmar su carácter malicioso sin una elevada probabilidad de dar un falso positivo)

mientras que hay en torno a 20 versiones de las que se puede asegurar que son malware por tener más de tres analizadores afirmándolo.

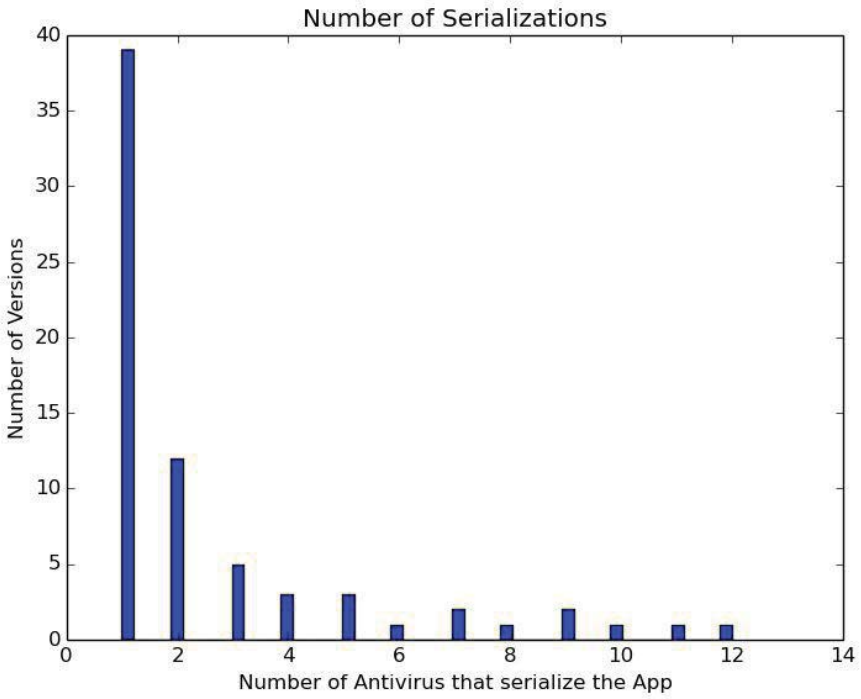

Figura 4.5: Cantidad de aplicaciones que dan positivo en los análisis de VirusTotal en Google Play

Como se ha comprobado, el mercado más popular es GooglePlay; además de ser uno de los que aseguran determinadas medidas de seguridad para dificultar la proliferación de malware y grayware. Por ello, es el más interesante: se pueden observar qué tipo de aplicaciones logran vulnerar sus escáneres y hasta qué punto catalogan una aplicación como grayware o no.

De esta manera, será el mercado seleccionado para llevar a cabo el estudio aunque se mantenga la posibilidad de extrapolarlo al resto de mercados en un futuro.

4.2 Publishers de aplicaciones grayware

A partir de un trabajo anterior [7], se obtuvieron 1386 publishers que distribuían PUP en Windows; es decir, que eran proveedores de grayware. La idea inicial es intentar encontrar cuántos de esos desarrolladores han entrado en el comercio de Android con la finalidad de distribuir sus productos.

Como algunos de ellos tenían caracteres no ASCII, se disminuyó la lista a 1099. La búsqueda de los publishers se llevó a cabo bajo previa limpieza de los strings: palabras como sft. o sl. se eliminaron y no se hacía distinción entre mayúsculas y minúsculas; además de realizar stemming en ocasiones. Esto redundó en la identificación de 77 de ellos.

A partir de ellos, se almacenaron 435 certificados usados por los miembros de la lista inicial para ser identificados. Esta es la base de datos que conforma el conjunto de publishers y certificados con la que se trabajará para realizar el estudio.

5

Soluciones Propuestas

Dependiendo del escenario en el que nos encontremos se deberán seguir diferentes estrategias para lograr el resultado final. Estos escenarios dependen del conocimiento que se tenga del input. Se pueden conocer dos developer name y tratar de adivinar si tras ellas se esconde el mismo desarrollador o, por el contrario, no tener input y disponerse a encontrar casos de desarrolladores con más de una cuenta.

5.1 Escenario 1: Identificar si dos developer name son un solo desarrollador

Si el input contiene dos *developer name* distintos, se pueden buscar todas las aplicaciones subidas por ambos desarrolladores para obtener los certificados con los que se firmaron dichas apps y otros metadatos, así como el código fuente.

5.1.1 Comparación de metadatos de alto nivel

La mayoría de los metadatos de alto nivel considerados se corresponden con los certificados. Esos certificados contienen metadatos que permiten relacionar a los desarrolladores, ya que a diferencia de otros sistemas (como Windows) en los que es preferible que estén firmados por autoridades de certificación confiables, en Android los certificados están, en su mayoría, autofirmados. Por esta razón, aunque un desarrollador genere certificados distintos, suele utilizar los mismos patrones y aplicar mínimas variaciones a dichos certificados. Por ello, si se estudian los siguientes parámetros del metadata se puede conseguir identificar la verdadera autoría de las aplicaciones:

- **Certificate subject common name**: el nombre que se da el propietario a sí mismo en ese certificado.
- **Certificate subject organization name**: el nombre legal de la organización o negocio a la que pertenece. Puede ser un valor vacío.
- **Certificate subject organization unit name**: el nombre del departamento de la organización a la que pertenece. Puede ser un valor vacío.
- **Dirección**: compuesta por la concatenación del país, estado y localización. Cualquiera de los tres puede ser vacío, por lo que si ninguno almacenara un valor, el campo dirección podría estar vacío.
- **Email**: correo electrónico del *publisher*. Puede ser un valor vacío.
- **Fingerprint**: clave pública del certificado. No pude haber varios publisher con el valor. Por ello, no puede ser nulo.
- **Nombre del desarrollador**: no puede ser un valor vacío.

- **ESLD**: generalmente, la estructura de un *package name* es la misma que la inversa de un dominio. Aplicando un *stemming* basándonos en técnicas como el top/second level domain, se puede obtener información del tipo de dominio en el que suscribe un desarrollador sus aplicaciones.

5.1.2 Comparación de aplicaciones

Para ello, hay que realizar una búsqueda en el dataset de aquellas aplicaciones que tengan los *developer name* seleccionados. Una vez se han seleccionado las apps, se realiza una combinatoria entre las aplicaciones para compararlas entre sí. En este caso se pueden seguir dos caminos para realizar el veredicto:

(a) **Comparación de código**

La distancia entre los métodos de las aplicaciones estudiadas se puede implementar manualmente. Sin embargo, la herramienta llamada Androguard tiene una funcionalidad denominada *androsim* que permite evaluar la similitud dadas dos aplicaciones señalando el porcentaje de métodos idénticos, cuántos han sido eliminados y cuántos son nuevos. En función de esos parámetros se establece el valor final de similitud.

(b) **Comparación de metadata**

Si bien el método anterior es el más fiable, no siempre se puede obtener el código fuente (este es el caso de aplicaciones de pago) o son tan grandes que la comparación conlleva un tiempo demasiado grande. Es en estos casos que la evaluación por metadatos es preferible.

Como se intentan relacionar características de las aplicaciones y no de los desarrolladores, los metadatos seleccionados para el estudio son distintos:

- **Título**: nombre de la aplicación que aparece en el market.

- **Tipo de aplicación**: agrupación en la que se clasifica. Algunos ejemplos pueden ser: entretenimiento, libros, viajes, etc.
- **Descripción**: breve explicación (realizada por el desarrollador) de la operativa de la aplicación.
- **Tamaño de la aplicación**: tamaño que ocupa según GooglePlay.
- **Fingerprint**: clave pública del certificado.
- **Listado de API Keys**: son las claves que emplean los desarrolladores para conectarse con las APIs a las que necesitan acceder para el correcto funcionamiento de la aplicación.
- **Permisos**: listado de permisos que precisa la aplicación para poder funcionar y que deben ser aceptados antes de su instalación.
- **Número de *activities***: fragmentos de código destinados a que el usuario interactúe con la aplicación.
- **Número de archivos**: cantidad de ficheros que conforman el APKk de la aplicación.
- **Links**: URLs a las que accede la aplicación cuando está en funcionamiento.
- **Archivos del APK**: hashes de los ficheros que conforman el APK de la aplicación.
- **Nombres de directorios**: directorios o carpetas de la estructura de ficheros del APK.
- **Paths de los ficheros**: rutas para la localización de ficheros en un APK.

5.2 Escenario 2: Encontrar desarrolladores con más de una cuenta

Una idea inicial para enfrentarse a este problema consiste en acudir a otras listas de publishers de actividades cuestionables presentes en otras plataformas para comprobar si también se encuentran en Android. De ser así, con alta probabilidad haya casos de desarrolladores con más de una cuenta.

Otra opción consiste en ayudarse de un parámetro denominado en Tacyt de la siguiente manera: *Metadata API Key List*. Este feature identifica las APIs que utiliza una aplicación así como las claves que emplea identificarse en ellas. Este parámetro es capaz de relacionar de manera indirecta a dos desarrolladores puesto que cada usuario tiene que tener unas claves para cada API. Si dos desarrolladores con nombres distintos tienen las mismas claves, se puede deducir que es una misma persona con varias cuentas. Explotando por tanto este parámetro, se pueden extraer aquellas que tengan algún valor en este parámetro (ya que no todas lo tienen) y agrupar los desarrolladores en función de aquellas aplicaciones que tengan las mismas API keys.

6

Implementación

La implementación de estas soluciones recae en una elaboración de algoritmos de recuperación de datos, estudio de los mismos para analizar cómo de compactos son y sus distancias (así como la redundancia de parámetros que los definen) y unos potentes clasificadores capaces de discernir con eficacia las similitudes entre los conjuntos generados. Para todo ello se puede decidir generar dichos algoritmos o utilizar herramientas especializadas en estos estudios, como son Weka o R.

6.1 Identificar un publisher a partir de dos desarrolladores

6.1.1 Comparación de metadatos de alto nivel

Si se parte de dos nombres de desarrollador, es necesario encontrar todos los certificados correspondientes a ambos como primera instancia. Para ello, se puede

realizar una búsqueda en el dataset que estamos empleando; es decir, Tacyt. Al no ser una base de datos propiamente contiene alguna desventaja. En este caso se encuentra en la manera en la que se debe acceder a la información: no existe ninguna *query* que ofrezca otra cosa que no sean aplicaciones, por lo que no se puede obtener de primera mano una lista de certificados con una única llamada a la API.

De esta manera, el procedimiento para la recolección de certificados debe establecerse en dos etapas: en un primer momento se obtiene la información del primer desarrollador y después se repetirá la misma operación con el segundo. Cada una de estas etapas seguirá estos pasos:

1. **Búsqueda a través de la API**: se trata de una búsqueda por filtros; es decir, se solicitan aplicaciones que cumplan que los valores que toman en los parámetros especificados son los mismos que se indican en la query. En este caso se solicitarán aquellas aplicaciones alojadas en GooglePlay cuyo nombre de desarrollador sea uno de los solicitados. Un ejemplo se muestra a continuación:

   origin:GooglePlay developerName:"nombre desarrollador 1"

   Tras esto se obtiene un JSON como respuesta en el que se detallan todas las características de las aplicaciones que se corresponden con los filtros seleccionados.

2. **Almacenamiento de certificados**: una vez se han recolectado las aplicaciones, es preciso seleccionar de cada una los parámetros relativos a los metadatos correspondientes a cada solución. En Tacyt se almacena mucha información de las aplicaciones. Sin embargo, se debe realizar un estudio para obtener los más entrópicos; es decir, aquellas características que más in-

formación dan, así como las más independientes entre sí. Dichos parámetros seleccionados se han detallado en el *capítulo 5.1.1*.

3. **Set de certificados del desarrollador**: puede suceder que un mismo desarrollador utilice el mismo certificado varias veces, por lo que es necesario eliminar aquellos que se repitan en nuestro conjunto para evitar cálculos redundantes o problemas de precisión en los resultados.

Una vez hemos obtenido los dos conjuntos con los certificados pertenecientes a cada desarrollador, se deben comparar todos los elementos del primer cluster con todos los del segundo, por lo que se deben definir dichas parejas y pasárselo como input a la función que buscará la diferencia entre cada par de certificados. Con que una de las parejas de publishers tenga una distancia lo suficientemente pequeña, se puede asumir que ambos desarrolladores son el mismo.

La distancia entre dichos certificados puede definirse de maneras más o menos estrictas: booleana, definiendo un margen de similitud o con pesos por feature.

Distancia booleana: tan sólo distingue entre publishers idénticos o distintos. Este tipo de distancias son muy agresivas por lo que sirve para hacernos una primera idea de la agrupación de los datos, pero no tiene validez como resultado final.

El proceso es el siguiente: se compara la pareja de certificados y, si al menos uno de los features es idéntico, se asume que su distancia es 0. Si no, son totalmente distintos y su distancia es la máxima: 1.

Margen de similitud: es decir, en lugar de observar si los strings son idénticos, dar un margen de error. Hay features que modifican levemente algún dato como, por ejemplo, su dirección o abrevian y acortan el nombre de su organización. De

esta manera se siguen unos parámetros ya utilizados previamente en estos casos [8]:

- **Distancia entre strings menor de 0.27**: es el caso de las features con mayor variabilidad, como el nombre de la organización, el nombre del departamento de la organización y la dirección en la que afirma ubicarse.
- **Distancia entre strings menor de 0.11**: se aplica a aquellos features más críticos, como el nombre del desarrollador y el del publisher.
- **Distancia cero**: en definitiva es asumir unos pocos features como booleanos. Este es el caso del fingerprint, del cual no tiene sentido pensar en una distancia entre sus strings. Se ha incluido también el ESLD ya que se trata de un package name tratado previamente. Por último, se ha añadido el email en esta clasificación ya que es un parámetro unívoco para cada desarrollador y no puede ser modificado. Para modificar un email es preciso crear uno nuevo.

Pesos por feature: esta modalidad puede combinarse con cualquiera de las dos anteriores. Es entendible que no todos los features deberían tener el mismo peso a la hora de decidir si dos desarrolladores son el mismo. De hecho, si el fingerprint de dos certificados fuera coincidente bastaría para afirmar que se trata del mismo publisher.

De esta manera, antes de determinar la distancia de dos certificados mediante pesos se revisa si el email, nombre, ESLD o fingerprint son idénticos. De ser así el resultado es positivo.

A continuación, se evalúan el resto de parámetros asignando los siguientes pesos:

- **Certificate subject common name**: 0.375.
- **Certificate subject organization name**: 0.25.
- **Certificate subject organization unit name**: 0.25.
- **Dirección**: 0.125.

6.1.2 Comparación de aplicaciones de ambos desarrolladores

Tanto si se puede obtener el código de las aplicaciones que se desean comparar como si no, el comienzo de este módulo es idéntico en ambos casos.

Se realizan búsquedas en Tacyt filtrando por su developer name. Si se quisiera tener un especial interés en aquellas maliciosas habría que refinar más el filtrado con metadatos como el que devuelve serializaciones de antivirus o análisis de descripciones de aplicaciones comparadas con los permisos solicitados.

Una vez recogidos los metadatos o el código de las aplicaciones que se desean comparar, se establecen las parejas que se van a comparar (una versión de cada desarrollador) que servirán de parámetros de entrada de los analizadores. El proceso que determina si son la misma aplicación se realiza de dos maneras:

6.1.3 Comparación de código

Para tratar con las APKs de las aplicaciones que se desesan tratar se ha empleado Androguard; una librería implementada en Python con una funcionalidad denominada Androsim, la cual mide la similitud entre dos aplicaciones.

La información que devuelve es muy completa a la hora de calcular el porcentaje de similitud: devuelve 5 parámetros con la cantidad de métodos que comparten

dicha característica y una evaluación final en la que indica el porcentaje que le otorga a la comparación de ambas aplicaciones. Los parámetros son los siguientes:

- **IDENTICAL** se refiere a aquellos métodos que son exactamente iguales en ambas aplicaciones.
- **SIMILAR** identifica a aquellos métodos que tienen unas leves modificaciones: comparten cabeceras, algoritmos similares, etc.
- **NEW** fragmentos de código que contiene la segunda aplicación y la primera no.
- **DELETED** funcionalidades existentes en la primera aplicación que no aparecen en la segunda.
- **SKIPPED** aquellas secciones del código que no han podido ser comprobadas correctamente y han tenido que ser obviadas para continuar con la comparación. Generalmente suele resultar 0.

Tras mostrar estos datos, se muestra el porcentaje de acierto final otorgado. Por ejemplo:

```
--> methods: 82.589461% of similarities
```

Gracias a Androsim se puede conseguir un veredicto fiable y preciso y la convierte en una herramienta realmente útil en el campo de la seguridad. Un ataque común es ser capaz de obtener el código de una aplicación, modificarla e inyectarle código malicioso y volverla a empaquetar para subir al mercado. Si se observan dos aplicaciones que parecen ser la misma, saber qué métodos son los nuevos puede indicar eficientemente el código preciso encargado de realizar las tareas delictivas.

Además, estudiar una misma aplicación después de haber sido ejecutada también puede mostrarnos los métodos que genera el malware inyecta mientras se ejecuta.

Los parámetros que compara van más allá de una comparación de nombres de métodos y tipos de los valores de input y retorno, sino que revisa otros parámetros como por ejemplo las APIs a las que accede, constantes empleadas, inicializaciones o el control de excepciones.

6.1.4 Comparación de metadatos

Si bien lo más preciso es comparar el código, no siempre es posible ya que en ocasiones no se dispone del APK, como es el caso de aplicaciones de pago; además el tiempo de ejecución es más lento cuando las aplicaciones difieren mucho o las aplicaciones tienen mucho contenido.

A la hora de decidir qué metadatos son óptimos es necesario comprobar cuánta información aportan (evitando aquellos parámetros redundantes) y si son significativamente descriptivos de las aplicaciones. Una primera forma de identificar estos parámetros es escoger un subconjunto de aplicaciones y ver en cuántas de ellas el feature es idéntico y en cuantas es distinto. Si atendemos a la Figura 6.1 se puede observar cómo son más fiables aquellos features con más población (puesto que representa cantidad de aplicaciones que se diferencian en dicho metadato) y en los que menos población tienen casi todas las aplicaciones se asemejan.

Una vez obtenidos los parámetros más diferenciadores se realiza una criba: en este caso se seleccionan aquellos que aparecen con más frecuencia en Tacyt y cuya información aportada sea útil para nuestro estudio.

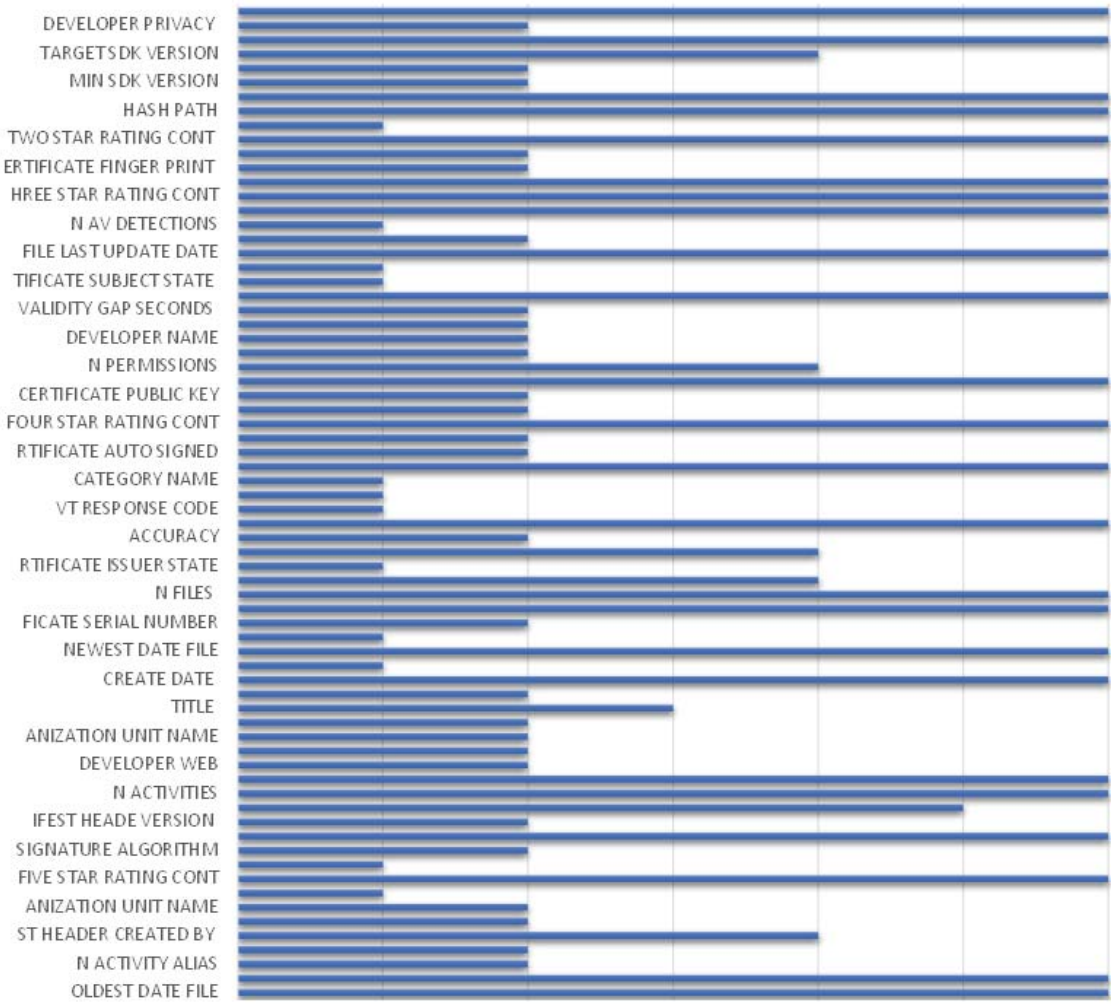

Figura 6.1: Cómo de diferentes son los parámetros de las apps en la base de datos

De la misma manera que se realizaba con los metadatos de los certificados, es preciso valorar cada feature y saber cuánta importancia tiene frente al resto.

Los metadatos estudiados en esta fase del estudio son los que se presentan a continuación:

- **Título**: el nombre de la aplicación. Si se trata de un grayware que intenta impersonar la aplicación previa, probablemente se llame de manera similar.

- **Tipo de aplicación**: agrupación en la que se clasifica. Algunos ejemplos pueden ser: entretenimiento, libros, viajes, etc. Si bien parece que los tipos de las aplicaciones están restringidos, el metadato que aparece en Tacyt es demasiado variado, dando la sensación de arbitrariedad ya que parece escogerlo el desarrollador. Para identificar el tipo de aplicación que la define mejor lo ideal es analizar la descripción que realiza el desarrollador. Esta es la razón por la que los últimos análisis han dejado este parámetro fuera del análisis.

- **Descripción**: breve explicación (realizada por el desarrollador) de la operativa de la aplicación. El análisis de la descripción está recayendo en la misma librería que compara el resto de strings. Si bien funciona correctamente, se podría sacar mucha más información con un debido análisis, pero se ha decidido aplazar a mejoras futuras.

- **Tamaño de la aplicación**: Tamaño que ocupa según GooglePlay. Si se tratara de una aplicación que sustituye a una pasada que ya era malware probablemente su tamaño no se vea modificado. Si en cambio esta sustitución se produce modificando una aplicación benigna para inyectar código malicioso, la nueva será necesariamente mayor.

- **Fingerprint**: clave pública del certificado. Si bien ahora mismo se usa ese parámetro, después se sustituirá por el veredicto de similitud de certificados entre las aplicaciones que se están comparando (obtenido en los capítulos 6.1 y 6.2).

- **Permisos**: listado de permisos que precisa la aplicación para poder funcionar y que deben ser aceptados antes de su instalación. Si se dotara de la suficiente lógica al programa como para relacionar el tipo de permisos requeridos con la descripción de la aplicación se podría identificar a aquellas que parecen

solicitar muchos más de los que precisa, delatando en algunos casos su identidad maliciosa. Además, si son dos aplicaciones similares, los permisos no deberían cambiar en exceso.

- **Número de *activities***: representan los fragmentos de código que permiten al usuario interactuar con la aplicación. El hecho de que en Tacyt no se obtengan las activities empleadas para comparar hace que este parámetro pierda importancia, ya que la cantidad no refleja la misma información que una mera cantidad. Este parámetro es eliminado en los últimos análisis realizados.

- **Número de archivos**: cantidad de ficheros que conforman el APK de la aplicación. La cantidad de ficheros empleados si es próxima podría denotar que son la misma aplicación. Sin embargo, sabiendo el tamaño exacto de la aplicación, este parámetro no aporta información, por lo que es eliminado en los últimos tests.

- **Archivos del APK**: hashes de los ficheros que conforman el APK de la aplicación. La distancia en este caso forzosamente será booleana ya que no se puede obtener el contenido de los archivos, sólo su hash.

- **Nombres de directorios**: directorios o carpetas de la estructura de ficheros del APK. Si bien este parámetro puede aportar mucha información, Tacyt no lo ha extraído de la mayoría de las aplicaciones que aloja, por lo que en la mayoría de tests quedaba desierto este parámetro.

- **Paths de los ficheros**: rutas para la localización de ficheros en un APK. Al igual que el feature anterior, Tacyt no ha extraído esta información para todas las aplicaciones por lo que no se puede aprovechar en los tests.

- **Lista de API Keys**: las aplicaciones emplean frecuentemente APIs para realizar la lógica de sus programas. Para ello, precisan de una identificación en dicha API. Analizando a cuáles acceden y con qué peticiones se puede discernir si usan la misma identificación para la misma API, ligando a los desarrolladores que realizan la aplicación así como la lógica del programa. Para ilustrar este caso, pondré un ejemplo: se supone que una aplicación destinada a viajes no tendría por qué acceder a las mismas APIs que una de juegos. De compartirlas, puede significar que las emplean para realizar adware denotando su carácter PUP.

La diferencia entre cada par de features de cada aplicación se mide de diferente manera según el tipo de dato que contenga:

- **Distancia booleana** se establece entre aquellos features que precisan de exactitud en la comparación. Este es el caso de parámetros que comparan nombres exactos, claves públicas o hashes.

- **Difflib** se trata de una librería de Python la cual se basa en el *Algoritmo de Ratcliff and Obershelp*; el cual calcula el cociente entre la cantidad de caracteres coincidentes y el número total de caracteres. Para ello trata de encontrar la mayor subsecuencia idéntica entre ambas cadenas.

- **Índice o Coeficiente de Jaccard** se emplea cuando hay un conjunto de datos. Este es el caso de un conjunto de API keys, permisos o links de los que se desea comprobar si tienen en común. La manera de calcularlo es la siguiente:

  $$J(A,B) = |A \cap B| \ / \ |A \cup B|$$

  Adoptando valores entre 0 y 1, lo cual es idóneo porque es el rango de distancia empleada en el resto de casos.

6.2 Encontrar desarrolladores con más de una cuenta

Esta solución se basa en la búsqueda de publishers conocidos por publicación de aplicaciones grayware. En este caso se utilizará la lista proveniente de aquellos PUP conocidos de Windows para comprobar si se encuentran también en Android.

La manera en la que se va a pretender discernir si hay desarrolladores con más de una cuenta es empleando la función implementada tras la sección 6.1.1; es decir, se obtendrán los certificados de la lista y se compararán por parejas para indicar qué distancia hay entre cada par de certificados.

Una vez se hayan obtenido dichos valores, se generará una matriz simétrica de distancias que permitirá realizar una clusterización de los desarrolladores; de tal manera que se agrupen de acuerdo a si son la misma persona o no. Para producir dicha clusterización, se puede programar un algoritmo básico o emplear herramientas como weka y R que tienen una gran variedad de funcionalidades para realizarlo.

6.2.1 Aggressive

Se trata de una implementación manual que aglomera demasiadas aplicaciones por cluster, pero permite mostrar una primera idea de la compactación y separabilidad de los desarrolladores de la lista.

Este algoritmo recorre las parejas de certificados para determinar su distancia. Al ser una implementación en python, por cada certificado se originó una clave "FAM_NAME" (es decir, el nombre de la familia a la que parece pertenecer). Sin embargo, si se utilizaran otros lenguajes se podrían generar estructuras de datos similares. El procedimiento es el siguiente:

Si alguno de los features comparados entre dos certificados tiene distancia cero, se entiende que pertenecen a la misma familia. Para reflejarlo, en el campo destinado a *FAM_NAME* se incluye el mismo identificador. Sin embargo, es evidente que si uno de ellos pertenecía a otra familia podríamos estar sobreescribiendo este parámetro sin ninguna consistencia. Por ello, aquel que modifique su *FAM_NAME* deberá comprobar antes si hay algún otro certificado que comparta el mismo para actualizarlo también.

6.2.2 R

Es un entorno en el que se pueden realizar gráficas y cálculos estadísticos muy variados que resultan muy útiles en el campo que nos ocupa. A través de esta herramienta se realizarán clusterizaciones con diferentes clasificadores para comprobar su efectividad. Esta clasificación es necesariamente no supervisada ya que antes de su ejecución no se conoce la distribución de los elementos y, por tanto, se desconoce también el número de clusters.

En este caso los clasificadores utilizados serán *PAM* y *k-means*:

- **PAM**: es la técnica de clustering de partición entorno a centroides. Para llevarlo a cabo, precisa de un número determinado de conjuntos en los que clasificar además de una matriz de similitudes. Puesto que iterar sobre todas las posibles agrupaciones de datos puede tornarse en un problema difícil de escalar, se emplean heurísticas que aproximen las soluciones. En este caso, se agrupan los elementos en torno a los centroides de cada clúster que es considerado un elemento central y representativo del conjunto. Este centroide

puede obtenerse de la siguiente manera:

$$m\ C = arg\min\ \ m \in C\ \ \sum_{j=1}^{n} dist(m, m_j),\ \ m_j \in C$$

Es decir, puede definirse como el elemento que minimiza la suma de las distancia con respecto al resto de componentes de su conjunto. Una vez definidos los centroides, el algoritmo PAM se construye de la siguiente manera (según la iteración de Voronoi):

1. **Elección centroides iniciales**: tantos como k grupos se hayan determinado. Este paso inicial se lleva a cabo de manera aleatoria.

2. **Asignación de clusters a cada elemento**: cada elemento calcula su distancia a todos los centroides. Pertenecerá a aquel conjunto cuya distancia a su centroide sea menor.

3. **Actualización y repetición**: una vez se han clasificado los elementos, se debe recalcular el centroide por cada cluster para refinarlo. Una vez se obtienen los nuevos medoids se vuelve al paso anterior para recalcular la pertenencia de los elementos a cada clase. Estas iteraciones se mantendrán siempre y cuando el cálculo del coste disminuya:

   $$coste(x, c) = \sum_{k=1}^{d} |x_i - c_i|$$

   Siendo x cualquier elemento, c el centroide y d la dimensión (número de features).

- **K-means**: es un método de clasificación que agrupa en k clusters de manera que cada elemento pertenece al conjunto cuya distancia es la más cercana.

Como se puede comprobar, es una generalización de PAM. La distancia que intenta minimizar en este caso es la suma de los cuadrados de las distancias dentro de cada grupo:

$$arg\min_{S} \sum_{i=1}^{k} \sum_{x_j \in S_i} ||x_j - \mu_i||^2$$

Donde μ_i es la media de puntos en S_i.

El algoritmo empleado para realizar la clusterización, por tanto, es el mismo que el empleado en PAM: iteraciones de Voronoi; pero aplicando diferentes distancias.

En ambos clasificadores se precisa indicar el número de clusters, lo cual contradice el principio de algoritmo no supervisado. La manera de arreglar esto es iterar sobre un número mínimo y otro máximo de conjuntos y realizar la clasificación. Una vez obtenida la clasificación se hará uso del índice de dunn para conocer el k óptimo.

Por tanto, el algoritmo tendrá la siguiente estructura:

1. **Distancias**: para cada par de certificados se debe calcular su distancia tal y como se definieron en el Capítulo 6.1.1. Dependiendo de la distancia escogida el resultado variará, por lo que es muy importante definirla correctamente.

2. **Matriz simétrica**: a partir de las distancias se describirá una matriz simétrica de distancias. Su dimensión será NxN, siendo N el número de certificados que se pretenden analizar. De esta manera, cada fila y columna representa un certificado. Por tanto, en cada celda se muestra la distancia (normalizada entre 0 y 1) entre cada par de certificados calculada anteriormente.

   Si el orden de los certificados mostrados en las filas es el mismo que el orden de las columnas, la diagonal resultante tendrá todos los elementos a 0, ya

que la distancia de un elemento consigo mismo es nula. A ambos lados de la diagonal habrá los mismos valores puesto que la distancia de el certificado i al j es la misma que del j al i.

3. **Clasificar**: estableciendo un número mínimo de clusters y otro máximo, se realiza un bucle en el que se realiza la clasificación (sea el clasificador de k-medias o PAM). Por tanto, el tiempo de ejecución depende directamente del número de certificados que se están analizando.

4. **Índice de Dunn**: analiza todos los clusters generados para medir su compactación y su separabilidad, obteniend así el k-óptimo de la clasificación. La manera de calcularlo es la siguiente:

$$D = \min_{1 \leq i \leq n} \left\{ \min_{1 \leq j \leq n, i \neq j} \left\{ \frac{d(i,j)}{\max_{1 \leq k \leq n} d'(k)} \right\} \right\}$$

6.2.3 Weka

Es una herramienta que permite trabajar con algoritmos de aprendizaje automático para problemas relacionados con minería de datos. Es esta razón la que lo hace muy útil para la clasificación que pretendemos realizar ya que, a partir de un conjunto de certificados se pretende extraer *datasets*.

El clasificador empleado en este caso es el denominado *Cobweb*. A través de este algoritmo se obtiene un árbol que ordena clusters de manera jerárquica: en la base del árbol aparece el número máximo de clusters en el que ha agrupado los certificados. En jerarquías superiores, se encuentran clusters que agrupan algunos conjuntos del nivel inferior. Una vez obtenido el árbol, la tarea principal es idear dónde realizar el corte, es decir, en qué nivel se obtiene la agrupación óptima.

Tal y como describe el Instituto Nacional de Astrofísica Óptica y Electrónica [4], cada nodo del árbol será denominado concepto; ya que cada uno de ellos define los elementos ubicados en ese cluster. Por tanto, si $P(C_i)$ es la probabilidad relativa al concepto y las probabilidades condicionales de las combinaciones feature-valor $P(A_i = V_{ij}|C_k)$. Para la construcción del árbol calcula la utilidad de la categoría:

$$CU = \frac{\sum_{k=1}^{n} P(C_k)\left[\sum_i \sum_j P(A_i = V_{ij}|C_k)^2 - \sum_i \sum_j P(A_i = V_{ij})^2\right]}{n}$$

Esta categoría se encarga de obtener un valor que represente la diferencia que supone la cantidad de información recuperada sobre un certificado gracias al nodo C, contraponiéndolo con la información obtenida si dicho C no existiera. Siempre que ese valor sea positivo, esa partición es buena ya que estamos ganando información.

La manera de decidir la pertenencia de un objeto a un nodo u otro reside en el cálculo de su CU y asignarle el nodo con mayor valor. Y, para determinar los casos en los que agrupa o divide nodos, realiza el siguiente algoritmo: en cada paso de ejecución selecciona los dos mejores CU para unificarlos. De no producir un efecto beneficioso se procede a dividirlo.

7

Resultados

En diversas etapas de la implementación se han ido obteniendo resultados de cada sección que han permitido ir mejorando su efectividad. A lo largo de este capítulo se presentan los resultados de los tests realizados a las soluciones propuestas de los problemas planteados. En este capítulo no hay valoraciones acerca de la precisión y usabilidad de los algoritmos, sino más bien de exponer su comportamiento.

Las dos grandes estrategias planteadas, independientemente del input conocido, son o estudiar a los desarrolladores a partir de metadatos de alto nivel (basados sobre todo en análisis de certificados) o revisar las aplicaciones que suben a Google Play; ya que tanto sus metadatos como el código fuente revelan muchas características.

7.1 Clustering de publishers a partir de metadatos de alto nivel

Bajo este epígrafe se pretende mostrar el ratio de acierto y error mostrados al calcular la distancia entre certificados de diferentes publishers. Se procederá a unificar los resultados de la implementación de lo concerniente a los capítulos 6.1.1 y 6.2. La razón de agruparlos es, que para efectuar la implementación de la sección 6.2, hay que rellenar una matriz de distancias. Cada celda será la distancia individual entre dos certificados lo cual fue definido en el capítulo 6.1.1.

Para ello, se explicarán los diferentes matices entre clusters y aplicación de distancias entre certificados específica para cada caso de tal manera que mejore su rendimiento.

7.1.1 Aggressive

A partir de un conjunto de desarrolladores de los cuales se conocía que tenían aplicaciones con una similitud suficiente como para poder determinar que son el mismo desarrollador, se han decidido testear las dos versiones de este algoritmo:

(a) **Distancia booleana**

Al medir la distancia entre features de una manera tan estricta (o el parámetro es idéntico o totalmente distinto), hay diversos casos en los que no es capaz de obtener un resultado de similaridad cuando es evidente que nos encontramos ante un mismo publisher.

Esta forma, por tanto, parece que no debería generar falsos positivos puesto que tiende a afirmar que son distintos ante la duda. Sin embargo, sí se demuestran casos en los que asume que dos publishers son el mismo sin que

haya un patrón claro de similitud directa tal y como se muestra en la tabla que se muestra a continuación.

CN	**O**	**OU**	**Address**	**[...]**
Feelingtouch	Feelingtouch	Feelingtouch	hejiangCNCN	
Bobisoft	Bobisoft	?	Hungary3636	

[...]	**Email**	**Fingerprint**	**Dev**	**ESLD**
	?	9C24[...]AC30	Feelingtouch Inc.	pgsoul.com
	?	F258[...]C0E2	bobisoft	bobisoft.com

Tabla 7.1: Dos aplicaciones de publishers distintos agrupadas en el mismo aggressive cluster basado en la distancia booleana

Esto se debe a que otra pareja de certificados de dos clusters distintos deben tener algún parámetro lo suficientemente cercano que ha hecho que ambos conjuntos se unan conformando uno sólo. De esta manera se aprecian certificados que denotan clusters distintos en una misma familia.

(b) **Margen de similitud**

En este caso, al permitirle a los strings tener cierta distancia; es decir, poder agrupar a los siguientes publishers como el mismo: `CBS Interactive Inc.` y `CBS Inc.`.

Si bien tiene un mejor rendimiento que la solución booleana no llega a ajustarse de una manera correcta, mostrando tanto falsos positivos como falsos negativos.

7.1.2 R

Los resultados de este apartado están simplificados en la Tabla 7.2. En función de la distancia y del clasificador la cantidad de clusters generados ha sido muy dispar. El subconjunto con el que se contaba cuenta con 436 certificados distintos.

Matriz de medias	kmeans	54
	PAM	120
Matriz de pesos	kmeans	179
	PAM	15

Tabla 7.2: Resultados de clustering con R

En la medida en que los resultados no iban cuadrando con el output esperado, se han ido refinando los parámetros de clasificación:

(a) **Matriz de medias de features**

Exceptuando a los parámetros que revelan una misma identidad de manera directa (como por ejemplo dos certificados con idéntico Fingerprint), el resto de features tienen el mismo peso; es decir, la distancia entre dos certificados se corresponderá como la media de las distancias que hay entre cada feature.

- **k-means**: en este caso el número de clusters generado es muy inferior al esperado: 54 con un índice de Dunn de 0.3823625. Se generan muchos falsos positivos.
- **PAM**: la cantidad de clusters es realmente recomendable:120 a pesar de tener un índice de Dunn realmente bajo: 0.06423333. Se aproxima al ideal de clasificación deseado.

(b) **Matriz ponderada de features**

A diferencia del cálculo de la matriz precedente, en esta ocasión la semántica de los parámetros se tiene en cuenta, dándole más peso a aquellos features que aporten una información más directa sobre el publisher.

- **k-means**: si bien el anterior algoritmo de k-means generaba pocos clusters, en este caso se produce lo contrario: 179 agrupaciones para la misma base de datos previa. Se aprecia que realiza overclustering; es decir, ante la duda da un negativo, disminuyendo la probabilidad de encontrar falsos positivos.
- **PAM**: la cantidad afirmada de clusters óptimos es 15, lo cual sólo demuestra que esta combinación de parámetros está errada.

7.1.3 Weka

En un primer momento se realizó un cluster muy laxo en el que se aplicacba una distancia de 0.3. Esto redundó en una agrupación poco exhaustiva generando tan sólo nueve clusters. Por tanto los grupos no eran representativos en realidad de desarolladores, sólo de características comunes.

Como el problema era la distancia empleada se decidió rebajarla poco a poco. El siguiente clasificador jerárquico generado fue con una distancia de 0.25. Seguía siendo bastante poco puesto que se generaban sólo 12 agrupaciones. Sin embargo, cuando se realizó un test con una distancia de 0.2, el número de clusters generados contaba con 117.

El árbol generado es bastante complejo con muchas ramificaciones, por lo que lo importante es discernir a qué altura se debe cortar el árbol; es decir, qué nivel del algoritmo realiza correctamente la agrupación.

7.2 Clustering de desarrolladores a partir de sus aplicaciones

7.2.1 Evaluación de metadatos

Al probar con aplicaciones iguales o similares, la media de los features permanecía en los valores más altos, demostrando que los agrupaba bajo un mismo publisher (tal y como muestra la Tabla X). Sin embargo, al probar con aplicaciones que tuvieran la misma temática pero fueran distintas, había algunas que obtenían una similitud entre el 0.5 y el 0.7, tal y como se muestra en la siguiente tabla:

Media:	0-0.3	0.3-0.4	0.4-0.5	0.5-0.6	0.6-0.7	0.7-0.8	0.8-0.9	0.9-1
Comparaciones:	0	0	0	0	3	11	16	76

Tabla 7.3: Comparación de metadatos de aplicaciones iguales o muy próximas

Media:	0-0.1	0.1-0.2	0.2-0.3	0.3-0.4	0.4-0.5	0.5-0.6	0.6-0.7	0.7-1
Comparaciones:	0	19	89	55	20	3	4	0

Tabla 7.4: Comparación de metadatos de aplicaciones distintas de igual temática

Con el objetivo de separar los valores de similitud cuando no son el mismo publisher, se estudian esas siete aplicaciones que parecen aproximarse más. Como se puede comprobar, en un inicio se cuenta con una cantidad de features mayor que con la que después se trabaja verdaderamente. Features como GMT y AuthorityKId desaparecerán en la versión final:

	0.5-0.6			0.6-0.7			
APPS COMPARADAS ->	2 | 3	4 | 9	4 | 12	6 | 20	7 | 10	15 | 16	18 | 20
PackageName	0,7586	0,5	0,553	0,8	0,386	0,9411	0,59
Title	0,8333	0,6667	0,8	0,5	0,847	1	0,75
Description	0,06	0,03	0,05	0,7	0,1367	0,022	0,218
Fingerprint	0	?	1	?	1	?	?
AuthorityKId	?	?	?	?	?	?	?
MetadataApiKeyList	?	?	0	?	1	?	?
Permissions	0,333	0,32	0,9	0,84	0,7097	0,25	0,526
GMT	?	?	?	?	?	?	?
CategoryName	1	1	0,1176	0,37037	0,11767	1	1
DirName	?	?	?	?	?	?	?
FilePaths	?	?	?	?	?	?	?
FileName	?	?	?	?	?	?	?
Links	0,81818	?	0,7778	?	0,769	?	?
MEDIAS FEATURES ->	0,5435	0,50355	0,5249	0,64288	0,6208	0,64269	0,617

Tabla 7.5: Análisis de features para comparar aplicaciones

Analizando la tabla se pueden extraer diversas conclusiones:

Al ser aplicaciones con la misma temática, es esperable que la similitud sea notablemente elevada en metadatos como *package name*, *title* y *category name*. A pesar de ello, las aplicaciones 15 y 16 eran significativamente parecidas por lo que, al revisarla manualmente descubrí que el muestreo aleatorio que había realizado sobre aplicaciones de idéntica temática, había dado con una pareja con estas características: la aplicación 15 había sido eliminada de Google Play y subida de nuevo bajo otro package name, el de la aplicación 16. A pesar de ello, el valor que toma en el parámetro descripción es muy bajo (0.022).

En cuanto al feature que compara las *categorías* de las aplicaciones, si bien en este caso se piensa que va a coincidir en la mayoría de las situaciones, está fallando en

3 de estas 7 aplicaciones. La razón es que se permite matizar mucho la tipología, bajando así la fiabilidad de este feature.

Estudiando el *authority key id*, se llega a la conclusión de que es un parámetro que se debe eliminar, ya que todas las aplicaciones son autofirmadas y, si se mostrase su valor y no fuera desconocido, repetiría la información que aporta el *Fingerprint*.

El parámetro de *Metadata API Key List* permite relacionar indirectamente dos desarrolladores que tienen un nombre distinto en el mercado: si el valor es el mismo, se puede afirmar que tienen el mismo desarrollador (como es el caso de las aplicaciones 7 y 10). Sin embargo, si son diferentes, no se puede afirmar con exactitud que se trate de distintos publishers.

En cuanto a los *links*, tan sólo tres comparaciones tenían metadatos suficientes para poder compararlos. Por tanto, se listan las URL que tienen en común para decidir si hay alguno que siga siendo muy genérico y sea necesario incluir en la lista de links que obviar (como, por ejemplo, google analytics):

- `https://csi.gstatic.com/csi`
- `http://www.androidcentral.com/root`
- `https://ssl.google-analytics.com/`
- `https://login.yahoo.com/`
- `http://schemas.android.com/apk/lib/com.google.android.gms.plus`
- `https://www.googletagmanager.com/`

7.2.2 Evaluación de código fuente

El dataset con el que trabajamos no permite una descarga automática de aplicaciones. Si se quieren descargar muchas, se debe enviar un correo a los responsables del mantenimiento en el que se indican mediante los hash de la aplicación cuáles se quieren obtener.

Esto hace que no sea muy escalable la investigación a partir del código fuente. Sin embargo sí se han realizado algunos avances.

Esta manera es la más efectiva en lo que se refiere a comparación de aplicaciones. Sin embargo, si son muy dispares y/o muy grandes, el tiempo que conlleva la ejecución se hace insostenible, no pudiendo llevar a cabo el veredicto.

8

Evaluación y Conclusiones

Tras los resultados obtenidos de la implementación de los algoritmos, se procede a valorar la efectividad y precisión de cada solución. Esta labor es fundamental de cara a una proyección a futuro si se plantea continuar con el proyecto y además nos permite ser capaces de valorar la utilidad del trabajo realizado.

8.1 Resultados de identificación a partir de metadatos de alto nivel

De las 3 maneras posibles en las que se puede realizar la clasificación basada en metadatos de alto nivel, la que peor rendimiento da es el método denominado aggressive. La elevada cantidad de falsos positivos y negativos en comparación con el resto de metodologías hacen que no sea fiable. Por ello, la discusión se va a centrar en los resultados concernientes a R y Weka

8.1.1 R

(a) **Matriz de medias de features**

Es decir, en lo que respecta a la matriz de medias, se generan situaciones muy balanceadas en las que los features pesan lo mismo y permite que si por azar un solo parámetro de ellos es muy similar no le da la suficiente confianza hasta que algún otro metadato acompañe en la decisión.

Sin embargo, esta propiedad beneficiosa no se ve reflejada en el algoritmo de clustering de k-medias, el cual encuentra una compactación y separabilidad mucho antes que el algoritmo dependiente de los *medoids* produciendo una mayor aglomeración.

- **k-means**: esta clasificación no es permisible para un total de 436 certificados. La agrupación es muy fuerte por lo que este método queda totalmente invalidado.
- **PAM**: la cantidad de clusters (120) es próxima a la ideal. Se identifican falsos positivos cuando el certificado no contiene información sobre la organización y departamento al que pertenece y basa casi toda la decisión en el nombre del publisher del certificado que a veces es común entre varios.

(b) **Matriz ponderada de features** A diferencia del cálculo de la matriz precedente, en esta ocasión la semántica de los parámetros se tiene en cuenta, dándole más peso a aquellos features que aporten una información más directa sobre el desarrollador. De esta manera, si un parámetro es propenso a dar falsos positivos por sí mismo, se disminuye su peso para que su influencia afecte menos que los más fiables.

- **k-means**: esta solución es la mejor conseguida hasta el momento: no aporta falsos positivos. Es algo superior al *ground truth* ya que en algunos casos no es capaz de identificar certificados de un mismo desarrollador:

	CN	**O**	**OU**	**Address**	**[...]**
	Mds	Manipal Digital Systems Pvt. Ltd.tbf	software	91 Karnataka Manipal	
	MDS	MDS	MDS	Karnataka Manipal	
[...]	**Email**	**Fingerprint**	**Dev**	**ESLD**	
	?	E53D[...]E862	Manipal Digital Systems Pvt. Ltd.	mds.com	
	?	E53D[...]E862	Manipal Digital Limited	mdl.com	

Tabla 8.1: Mismo desarrollador en clasificaciones diferentes

- **PAM**: esta solución debe ser descartada puesto que una clusterización de este dataset en tan sólo 15 elementos denota una muy mala sinergia con respecto a la matriz de distancias.

Por tanto, a la vista de estos resultados se puede concluir que si se quiere evitar un estudio para localizar qué metadatos tienen más peso que otros y cómo de flexible debe ser esa distancia, se puede resolver mediante la primera solución. Basándonos en la media de la distancia de los features aplicada al cluster de medoids; es decir, PAM se obtiene un gran porcentaje de acierto aunque se produzcan falsos positivos.

En cambio, si se le dedica tiempo al estudio de los metadatos, calibrando en diferentes iteraciones, se puede conseguir no dar falsos positivos con una matriz ponderada y el algoritmo de clustering de k-means.

8.1.2 Weka

Como se ha indicado en el capítulo de implementación, lo primordial es conseguir encontrar el nivel del árbol en el que se debe realizar el corte; es decir, en qué nivel se obtiene la mejor clusterización de los datos de input.

Ya que los dos primeros tests generaban pocos clusters, se realiza únicamente un estudio del tercero: el que aplica una distancia de 0.2 a los parámetros.

Si se realizara un corte en el nivel 3, las agrupaciones son bastante pequeñas y contienen a los mismos desarrolladores en la mayoría de los casos. Sin embargo, hay algunos casos en los que incluye varios desarrolladores distintos en un mismo cluster (llega a haber un nodo con 163 desarrolladores que, evidentemente no son el mismo).

Tras estudiar los datos generados, parece que el mejor resultado surge al escoger en algunas ramas el nivel 3 y en otras el nivel 4, lo cual hay que intentar parametrizar automáticamente o escoger otro método de clusterización.

8.2 Resultados de identificación a partir de comparación de aplicaciones

Para discernir el ratio de acierto o error de la solución que analiza los metadatos, se escogen unas cuantas aplicaciones que parecen ser la misma y de las que se pueda obtener el APK para comparar los resultados de la identificación por metadatos frente a la comparación automática del código. Los resultados obtenidos tras comparar la información alojada en Tacyt se encuentran en la tabla que se muestra a continuación:

Apps	Metadata								
	Package	**Title**	**Descr.**	**Key**	**API**	**Perm.**	**Type**	**Links**	**Total**
1,2	0,5079	0,541	0,955	0	1	1	1	1	0,75
3,4	0,41666	0,28	0,201	0	1	0,7	0,26666	0,5	0,42
8,9	0,2916667	0,2857	0,034	1	1	0,758	0,63636	0,4444	0,559
10,11	0,7619	0,864	0,67148	0	1	1	1	0,5333	0,728
12,13	0,27778	0,125	0,05265	0	1	1	1	0,875	0,5413
14,15	0,73913	0,1739	0,024	1	0,5	0,63636	1	0,5217	0,574
16,17	0,81	0,65	1	1	1	1	1	1	0,932
21,22	0,3846	0,25	0,022	0	1	1	0,5454	0,259	0,432
23,24	0,776	1	0,8	0	1	0,5	0,63636	1	0,714
25,26	0,163	0,25	0,0069	0	1	1	0,3846	0,625	0,428
27,28	0,32558	0,18181	0,0048	0	1	0,57	0,72727	0,18181	0,374
29, 30	0,26667	0,105	0,01	0	1	0,69	0,63636	0,66667	0,422
31,32	0,367	0,055	0,022727	0	1	0,531	0,0769	0,5	0,319
40,41	0,217	0,114	0,02857	0	1	0,90909	0,54545	1	0,476
42,43	0,0317	0,232	0,01	0	1	0,5	1	0,714	0,472
48,49	0,3	0,256	0,026	0	1	0,8889	0,23	0,5	0,400
50,51	0,50909	0,3333	0,033	0	1	1	0,45454	0,3	0,453
52,53	0,73	0,28	0,0039	1	0,5	1	0,63636	1	0,644
54,55	0,55	0,1	0,55	0	1	1	0,54545	1	0,594

Tabla 8.2: Comparación por metadatos vs. análisis de código: Metadatos

Solamente hay una comparación que ha afirmado que tienen una similitud del orden de 0.9. Otras tres se encuentran en el intervalo de 0.7 y el resto entre rangos del orden de 0.3 y 0.6.

Para generar el *ground truth* compararemos dichas apps con Androsim:

Apps	Androsim					
	Identical	**Similar**	**New**	**Deleted**	**Skipped**	**Total(%)**
1,2	6025	5	2	0	0	99,98996
3,4	823	63	75	63	0	91,245223
8,9	257	695	743	435	0	47,657896
10,11	1895	628	5607	144	0	83,678
12,13	3381	0	0	0	0	100
14,15	958	2051	2370	1005	0	52,42
16,17	2145	2	0	0	0	99,991232
21,22	1491	2243	1654	695	0	60,838672
23,24	1832	9	0	4	0	99,622442
25,26	233	2007	2920	802	0	71,000598
27,28	2437	1904	956	1351	0	61,638719
29, 30	694	195	1014	39	0	90,91
31,32	2985	1253	256	1659	0	62,8936
40,41	4366010	10	5	3	0	99,877
42,43	950	947	1768	339	0	69,95
48,49	2614	2232	1986	315	0	77,14
50,51	797	646	234	514	0	60,703969
52,53	3162	742	768	257	0	87,253
54,55	2186	19	7	2	0	99,512

Tabla 8.3: Comparación por metadatos vs. análisis de código: Código

Hay una mayor cantidad de comparaciones en rangos en torno al 80% o 90% que las que presentan los metadatos. Para solventar este problema es necesario replantearse si los metadatos escogidos son los correctos así como el tratamiento que se les da. Se debe realizar un estudio similar al efectuado con los certificados para encontrar qué features deberían tener más peso que otros y cómo balancearlo.

9

Líneas de trabajo futuras

Tan importante es la realización de un proyecto como la evaluación del alcance que puede tener. Este estudio precisa de algunas mejoras en diversos puntos de la implementación, los cuales se pueden vislumbrar tras la obtención y posterior evaluación de los resultados. Dichas mejoras se comprenden desde cambios en algoritmos hasta nuevos mercados de aplicaciones en los que aplicarlos.

A la hora de generar la comparativa de aplicaciones a partir de sus metadatos, se llega a la conclusión de que hay parámetros muy restrictivos que pueden emplear las técnicas utilizadas en la comparación de certificados: análisis de importancia de features, utilizar información ya obtenida de los certificados en lugar del Fingerprint y considerar nuevos parámetros en el análisis.

Además, se le puede sacar un mayor partido a las descripciones de las aplicaciones ya que dan mucha información sobre las tareas que debe desempeñar. Esta información es muy importante ya que si dos aplicaciones tienen las mismas finalidades

aumenta la probabilidad de que sean la misma además de poder revisar otros parámetros como los permisos para comprobar si tienen sentido con respecto a los que verdaderamente necesitaría.

La recolección de grayware y malware no ha sido del todo metódica y dado que se cuenta con una herramienta muy grande y versátil se podría aprovechar las ventajas que aporta.

También se pueden ampliar los horizontes del proyecto y aplicar algunas de estas implementaciones en el mundo de iOS a pesar de las limitaciones que tiene el no poder acceder al código fuente para extraer más información.

BIBLIOGRAFÍA

[1] B. ANDOW, A. NADKARNI, B. BASSETT, W. ENCK, AND T. XIE, *A Study of Grayware on Google Play*, Proceedings - 2016 IEEE Symposium on Security and Privacy Workshops, SPW 2016, (2016), pp. 224–233.

[2] D. ARP, M. SPREITZENBARTH, M. HÜBNER, H. GASCON, AND K. RIECK, *Drebin: Effective and Explainable Detection of Android Malware in Your Pocket*, Proceedings 2014 Network and Distributed System Security Symposium, (2014).

[3] K. CHEN, X. WANG, Y. CHEN, P. WANG, Y. LEE, X. WANG, B. MA, A. WANG, Y. ZHANG, AND W. ZOU, *Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS*, Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016, (2016), pp. 357–376.

[4] I. N. DE ASTROFÍSICA ÓPTICA Y ELECTRÓNICA, *Cobweb*.

[5] A. GORLA, I. TAVECCHIA, F. GROSS, AND A. ZELLER, *Checking app behavior against app descriptions*, Proceedings of the 36th International Conference on Software Engineering - ICSE 2014, (2014), pp. 1025–1035.

[6] M. IKRAM, N. VALLINA-RODRIGUEZ, S. SENEVIRATNE, M. A. KAAFAR, AND V. PAXSON, *An Analysis of the Privacy and Security Risks of Android*

VPN Permission-enabled Apps, Proceedings of the 2016 ACM on Internet Measurement Conference - IMC '16, (2016), pp. 349–364.

[7] P. KOTZIAS, L. BILGE, AND J. CABALLERO, *Measuring PUP Prevalence and PUP Distribution through Pay-Per-Install Services*, Usenix Security, (2016).

[8] P. KOTZIAS, S. MATIC, R. RIVERA, AND J. CABALLERO, *Certified PUP: Abuse in Authenticode Code Signing*, Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, (2015), pp. 465–478.

[9] K. THOMAS, J. A. E. CRESPO, R. RASTI, J.-M. PICOD, D. MCCOY, L. BALLARD, E. BURSZTEIN, M. A. RAJAB, AND N. PROVOS, *Investigating Commercial Pay-Per-Install and the Distribution of Unwanted Software*, 25th USENIX Security Symposium - USENIX Security '16, (2016), pp. 721–738.

[10] Y. ZHOU, Z. WANG, W. ZHOU, AND X. JIANG, *Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets*, Proceedings of the 19th Annual Network and Distributed System Security Symposium, (2012), pp. 5–8.

Este documento esta firmado por

Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
Fecha/Hora	Thu Jun 08 23:26:51 CEST 2017
Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
Numero de Serie	630
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)