ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

SISTEMA DE APRENDIZAJE AUTOMATICO PARA LA DETECCIÓN DE MALWARE EN ANDROID.

MACHINE LEARNING SYSTEM FOR MALWARE DETECTION IN ANDROID.

Realizado por
Daniel García Frías
Tutorizado por
Enrique Domínguez Merino
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DE 2018

Fecha defensa:
El Secretario del Tribunal

Resumen: Las nuevas tecnologías, especialmente los dispositivos móviles, han ido evolucionando hasta convertirse en una parte imprescindible de nuestro día a día. No es de extrañar, por ello, que el sistema operativo (SO) Android, como máximo representante dentro de los sistemas operativos móviles, se haya convertido en objetivo de ciber-criminales. Para velar por nuestra privacidad y seguridad, se necesitan sistemas capaces de detectar las amenazas antes de que el daño sea irreparable. Debido a la cantidad de malware que aparece diariamente, analizar las muestras una a una se convierte en una tarea compleja e inviable, por tanto, surge la necesidad de automatizarla. Es aquí donde los sistemas de inteligencia artificial (IA) están empezando a desempeñar un papel importante. En este trabajo se pretende desarrollar un sistema inteligente completo compuesto por un motor de IA (encargado de decidir si una muestra es malware o no), un servidor (encargado de gestionar la información y de crear trabajo para el motor de IA), y por último, una aplicación Android, que se encarga de conectar todo el trabajo anterior con el usuario final. Para realizar todo esto, durante el proyecto, se estudian diferentes técnicas y se evalúan los resultados obtenidos.
Palabras claves: android, malware, inteligencia artificial (IA), clusters, redes neuronales

Abstract: New technologies, especially mobile devices, have evolved to become an essential part of our day to day. It is not surprising, therefore, that the Android operating system (OS), as the maximum representative within mobile operating systems, has become the target of cyber-criminals. To ensure our privacy and security, we need systems capable of detecting threats before the damage is irreparable. Due to the amount of malware that appears daily, analyzing samples one by one becomes a complex and unworkable task, therefore, the need arises to automate it. This is where artificial intelligence (AI) systems are beginning to play an important role. In this work, we intend to develop a complete intelligent system composed of an AI engine (responsible for deciding whether a sample is malware or not), a server (responsible for managing information and creating work for the AI engine), and finally, an Android application, which is responsible for connecting all the previous work with the end user. To do all this, during the project, different techniques are studied and the results obtained are evaluated.
Keywords: android, malware, artificial intelligence (AI) clusters, neural networks

ÍNDICE

1. Introducción……………………………………………………………………2
1.1. Motivación……………………………………………………………2
1.2. Objetivos…………………………………………………………...3
1.3. Estructura de la memoria…………………………………………...3

2. Estado del arte……………………………………………………………...4
2.1. Estado actual del sistema Android………………………………..…4
2.1.1. Crecimiento de Android como sistema operativo………………..4
2.1.2. Factores influyentes en el éxito de Android………………………6
2.1.3. Problemática existente en torno a Android………………………7
2.2. Android como objetivo del malware………………………………………9
2.3. Lucha contra el malware en Android……………………………………13

3. Diseño del sistema inteligente…………………………………..…………..16
3.1. Extracción de características……………………………………………..16
3.1.1. Extracción de información………………………………………16
3.1.2. Construyendo nuestro extractor de características……………17
3.1.3. Estudio del dataset…………………………………………………19
3.2. Estudio y desarrollo de los diferentes algoritmos………………………27
3.2.1. Desarrollando el sistema de clusters……………….……………27
3.2.2. Desarrollando la red neuronal……………………………………31
3.3. Evaluación de resultados……………………………………………………39

4. Diseño del detector de malware………………………………….…………42
4.1. Servidor Web…………………………………………………………………42
4.2. Aplicación cliente Android…………………………………………………43
4.3. Coordinación de los módulos……………………………………………..47

5. Conclusiones y posibles mejoras futuras……………………………………49
6. Referencias bibliográficas……………………………………………………51

1. Introducción

1.1. Motivación

Las nuevas tecnologías ocupan un papel muy relevante en nuestra sociedad, siendo unos de los temas más latentes en la actualidad. Los dispositivos electrónicos han ido evolucionando con el fin de adaptarse a los nuevos tiempos, tiempos en los que todo y todos estamos conectados. Poder, por ejemplo, controlar las luces de tu casa desde cualquier parte del mundo utilizando tan solo el dispositivo móvil, ya es una realidad. Por lo tanto, el dispositivo móvil ha pasado de ser una herramienta meramente para realizar llamadas y enviar mensajes a ser el centro de nuestra vida personal y profesional. Desde él, entre otras muchas, recibimos y enviamos emails, leemos las noticias, capturamos, con fotos y videos, momentos especiales de nuestra vida, efectuamos nuestras compras y realizamos operaciones bancarias. En definitiva los dispositivos móviles han evolucionado posicionándose como una de las herramientas más usadas hoy día. Poca gente imagina pasar un día entero sin su smartphone.

Por todo ello, no es de extrañar que los dispositivos móviles se hayan vuelto el centro de las miradas para los ciber-delincuentes, y en especial el SO Android, el cual actualmente es el SO más usado del mundo, superando incluso a Windows. Y donde hay ciber-delincuentes, hay expertos en ciberseguridad para luchar contra ellos y mantenernos a salvo de estas amenazas.

Google, como desarrollador del SO Android, ha estado intentando poner trabas, a través de sus actualizaciones, al desarrollo de aplicaciones maliciosas (malware), pero aun así, a diario se encuentra una gran cantidad de malware distribuyéndose desde internet, tiendas de terceros e incluso desde la tienda oficial de Android. Es por ello que las distintas empresas especializadas en ciberseguridad desarrollan sus herramientas para detectar y neutralizar estas amenazas.

Ante todo esto, la inteligencia artificial (IA) se posiciona como una tecnología muy potente para la detección de malware, al automatizar aún más el proceso y convertirse en un apoyo muy importante en la toma de decisiones.

Para finalizar, me parece relevante exponer que este proyecto une Android e IA, dos de mis grandes pasiones que se han ido desarrollando durante mi formación académica por una parte, pero también durante mi carrera profesional. Durante mi etapa como desarrollador Android, dos de mis principales desempeños han sido aplicaciones de motores antivirus (AVs) como VirusTotal y Koodous, por lo que siempre he querido desarrollar un motor, o una parte importante de uno usando mis conocimientos en esta materia.

1.2. Objetivos

Este trabajo tiene tres objetivos:

1. Realizar una investigación acerca del estado del arte, intentando dar una visión general del impacto del SO Android en el entorno actual, así como las principales amenazas a las que se enfrenta este, y los sistemas que luchan para intentar conseguir mantenerlo a salvo de los ciber-delincuentes.

2. Hacer un estudio con el posterior desarrollo de un sistema inteligente para la detección de malware. En este abarcamos desde la extracción de toda la información de una aplicación Android (APK), hasta el desarrollo de estos modelos, que, posteriormente serán usados para la detección del malware.

3. Diseñar un sistema completo antimalware para Android partiendo de un servidor web que contendrá el sistema inteligente y dará respuesta a las peticiones de la aplicación Android.

1.3. Estructura de la memoria

Cada objetivo mencionado anteriormente se corresponde con una parte de la memoria. En la segunda sección, “Estado del arte”, tratamos el primer objetivo, dar

una visión general del estado del arte y la evolución, tanto de Android y sus amenazas, como de los sistemas existentes para combatir estas.

A continuación, en la tercera sección, “Diseño de sistema inteligente detector de malware”, haremos un estudio de las características que podemos obtener de un fichero APK, se experimentará con distintos algoritmos de AI y finalmente, con los resultados obtenidos, se desarrollará el núcleo de nuestro motor AV.

En la cuarta sección, “Diseño sistema completo detector de malware”, se explicará el desarrollo de las dos partes que faltan para completar un sistema detector de malware completo, que son el servidor y la aplicación cliente en Android.

2. Estado del arte

2.1. Estado actual del sistema Android

2.1.1. Crecimiento de Android como Sistema Operativo

A día de hoy, prácticamente cualquier persona dispone de, al menos, un smartphone con tarifa de datos, y la mayoría del consumo de internet lo hace a través de este, pero no hace tanto tiempo la gente solo usaba ordenadores para consumir internet, y el claro líder de los SO de uso doméstico, en aquel entonces, era Windows, con una cuota de mercado muy superior a cualquiera de sus competidores como OSX o Linux.

En el artículo de Cnet se recoge la entrevista a Steve Jobs durante la noche de apertura de la conferencia All Things Digital, donde este anunció que la humanidad había entrado en la era Post-PC, y aunque muchos seguimos usando a diario nuestros ordenadores, sí que se puede decir que las ventas de ordenadores han disminuido en beneficio de dispositivos como smartphones y tablets. Esto se debe a que mucha gente solo usa el ordenador para consumir contenidos multimedia y/o las operaciones que realizan con ellos pueden ser fácilmente realizadas en alguno de los nuevos dispositivos.

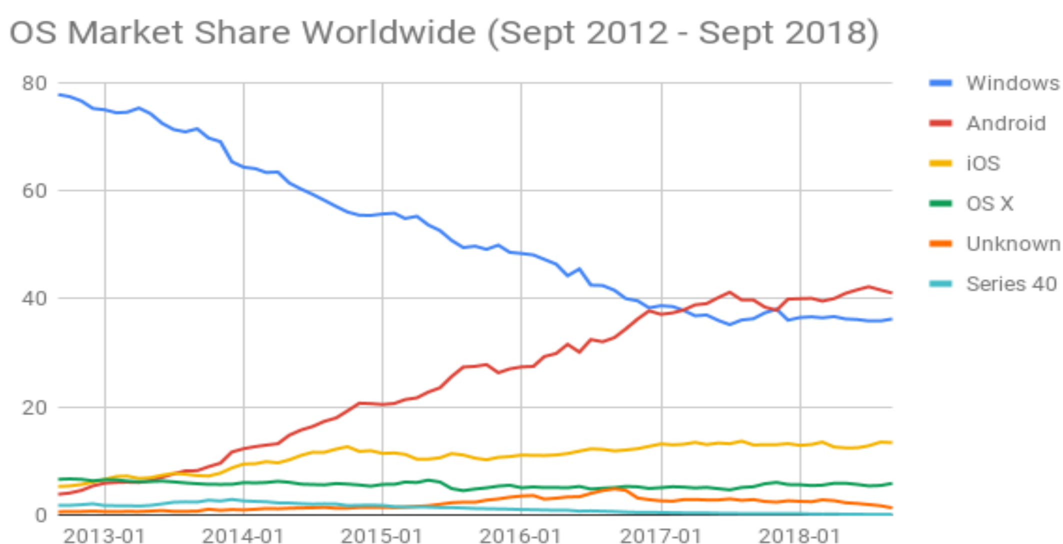
También han ido surgiendo propuestas para unificar entornos, como Windows, que apostó por un SO para todos sus dispositivos (ordenadores, tablets, consolas y smartphones). Linux intentó lo mismo pero ambas ideas, aunque atractivas, no terminaron de cuajar.

Empresas como Samsung empiezan a apostar por aprovechar al máximo la potencia de sus dispositivos dando la posibilidad, si el usuario quiere, de conectar este a un monitor y usarlo como si de un ordenador normal se tratase. En el caso de Samsung se requiere de la compra de su DeX Station que permite conectar el smartphone al monitor y a este teclado y ratón. Huawei también está dando pasos en la misma dirección, aunque en este caso solo es necesario un cable.

Puede que fracase, como pasó con la convergencia a un SO único en las distintas plataformas, pero teniendo en cuenta que cada vez estos dispositivos son más potentes, y ahora más entornos y herramientas son portados al cloud, se ve algo bastante plausible.

En la figura 1, veremos la evolución del mercado de los SSOO en los últimos 6 años, desde septiembre de 2012, donde el claro líder era Windows con un 77.84% de la cuota muy lejos de Android, que en ese entonces tan solo tenía un 3.84%, hasta septiembre de 2018 donde Windows ha ido perdiendo cuota hasta quedarse en un 35.9% frente a Android que se ha convertido con un 41.55% en el SO líder.

Figura 1. Evolución de los sistemas operativos los últimos 6 años

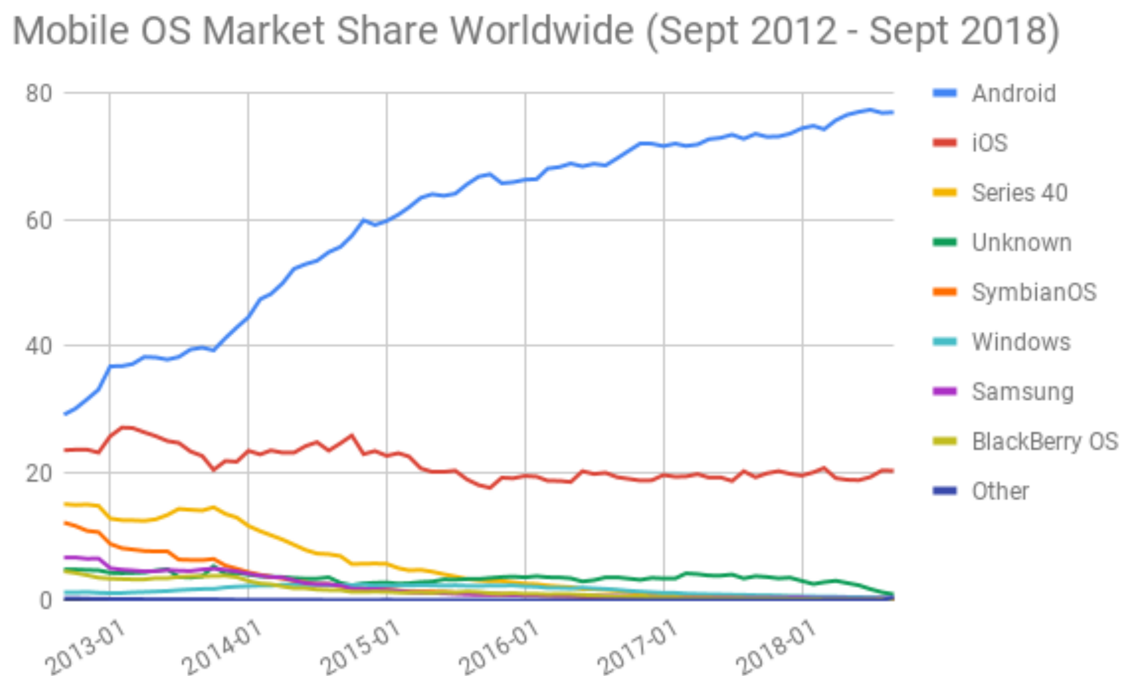


Fuente: herramienta de análisis de trafico web StatCounter

En la figura 2, podemos ver cómo Android se posiciona como líder indiscutible dentro de los SO móviles con un 77.09%, seguido de iOS con un 20.34%.

También vemos como Blackberry SO, que gozó de gran popularidad, se ha ido desvaneciendo hasta casi desaparecer, lo mismo pasa con Windows Phone, que aun intentando fortalecerse con la compra de Nokia, no pudo evitar el fracaso de su SO móvil.

Figura 2. Evolución de los sistemas operativos móviles los últimos 6 años



Fuente: herramienta de análisis de trafico web StatCounter

2.1.2. Factores influyentes en el éxito de Android

Analizando los factores que han influido en el éxito de Android como SO, destacamos los siguientes:

1. **Sistema Operativo abierto**, lo que posibilita que distintos fabricantes lo puedan usar en sus dispositivos como sistema base. Hasta 2016, cuando Google lanzó un dispositivo propio bajo la marca Pixel, ésta se aliaba con los diferentes fabricantes para desarrollar dispositivos bajo la marca de Nexus.

2. **Variedad de dispositivos:** El que haya una gran cantidad de compañías desarrollando dispositivos con el SO Android, posibilita que haya un extenso número de dispositivos, y de gamas. Gracias a esto el usuario puede adquirir un dispositivo más acorde con sus necesidades: tamaño del mismo, cámara, almacenamiento, potencia, calidad de pantalla,...

3. **Variedad de precios:** La gran variedad de dispositivos, con diferentes características y funcionalidades posibilita una ventana de precios bastante amplia. En Android podemos encontrar desde dispositivos por menos de 100€, con lo básico, hasta los terminales más top en torno a los 1000 €, por lo que el usuario adquiere el dispositivo que más se adecue a sus necesidades y presupuesto. Existen desde compañías que solo fabrican dispositivos de gama baja, a compañías que desarrollan dispositivos para todas las gamas como Samsung, Xiaomi, Huawei,...

4. **Facilidad de inicio para los desarrolladores:** Android está basado en el kernel de Linux, y en un primer momento, el lenguaje que se usaba para desarrollar las aplicaciones Android era Java, actualmente es Java junto con Kotlin.
   Que el lenguaje base fuera Java hizo mucho más accesible para muchos desarrolladores, que ya trabajaban con Java en otras plataformas, empezar a programar en Android. También, a diferencia de iOS, no se requería equipo especial para desarrollar, solo instalar Eclipse, en su inicio y después Android Studio, en tu ordenador personal y a programar.

2.1.3. Problemática existente en torno a Android

1. **Fragmentación:** El que haya muchos compañías usando Android como SO base, y desarrollando productos de distintas gamas, y con hardware tan distinto provoca que a la hora de desarrollar una aplicación se tengan que

tener todas estas variantes en mente, esto dificulta enormemente el desarrollo de las aplicaciones y por consiguiente la calidad de las mismas.

También está el problema de que muchas compañías usan las llamadas capas de personalización, como en el caso de Samsung con Touchwiz, Xiaomi con MIUI,... esto sumado a la cantidad de dispositivos con hardwares distintos provoca que cuando Google lanza una nueva versión del SO, los fabricantes tarden mucho en actualizar sus dispositivos, quedando muchos de ellos fuera de estas actualizaciones. No es extraño que los fabricantes mantengan varios años actualizados sus buques insignia y dejen de lado la gama baja y media.

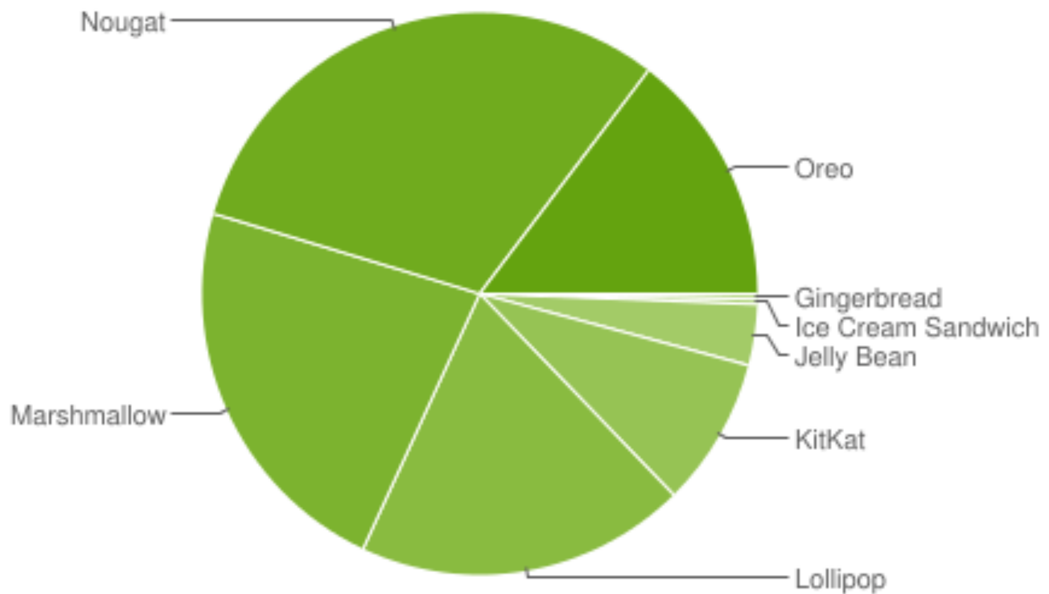
En septiembre de 2018 tras la reciente presentación oficial de Android Pie, podemos observar (en la tabla 1 y figura 3) como la última versión Android Oreo (8, 8.1), tan solo tiene un 14,6% del total de dispositivos activos en Android, siendo las que más dispositivos tienen Android Nougat (7, 7.1) con un 30,8% y Android Marshmallow (6) con un 22,7%.

Tabla 1. Distribución de las versiones de Android

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.3%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.3%
4.1.x	Jelly Bean	16	1.2%
4.2.x		17	1.8%
4.3		18	0.5%
4.4	KitKat	19	8.6%
5.0	Lollipop	21	3.8%
5.1		22	15.4%
6.0	Marshmallow	23	22.7%
7.0	Nougat	24	20.3%
7.1		25	10.5%
8.0	Oreo	26	11.4%
8.1		27	3.2%

Fuente: Estadísticas oficiales de Google (2018)

Figura 3. Distribución de las versiones de Android



Fuente: Estadísticas oficiales de Google (2018)

2. **Descontrol en el market de aplicaciones:** Android en sus inicios necesitaba una gran cantidad de aplicaciones para que su SO fuera atractivo, por tanto, su política a la hora de aceptar una aplicación en el market era muy poco restrictiva. Esto posibilitó el que se encontrarán aplicaciones de pésima calidad, plagios de otras aplicaciones, malware,...
   A todos nos viene a la mente la linterna que pedía muchísimos permisos, que en realidad recolectaba información del dispositivo.
   A día de hoy todo está mucho más controlado y se han seguido distintas vías de actuación para mitigar este problema.

2.2. Android como objetivo del malware

Como ya se comentó en el punto 2.1, Android es actualmente el SO más utilizado del mundo, y, por lo tanto, no es de extrañar que los creadores de malware hayan puesto sus ojos sobre esta plataforma. Asimismo, este malware, es de lo más

variado, y según el informe Android Security (2017), podemos encontrarnos con la siguiente clasificación:

- **Adware:** Es uno de los más comunes, su función es conseguir clic en banner de publicidad, publicidad no deseada. Si bien este tipo de malware no es muy dañino, puede ser muy molesto, mostrando continuamente ventanas de publicidad, algunas difíciles de cerrar que harán a los usuarios desesperar.
  También los hay que no muestran los anuncios, y mediante distintas técnicas consiguen el clic deseado.

- **Suscriptores SMS y Web:** Conocidos como clicker troyanos, su trabajo es robar datos de la cuenta móvil del usuario. Este malware consigue registrar al usuario en servicios de pago, e intenta ocultar todo el rastro para que tarde el mayor tiempo posible en detectar estas suscripciones.

- **Ransomware:** Ya que el móvil se ha convertido en una herramienta tanto personal como profesional imprescindible, no es de extrañar que haya empezado a aparecer ransomware para esta plataforma. El ransomware consiste en el "secuestro" del dispositivo, o más bien de su contenido. Normalmente realiza un cifrado del contenido del mismo y exige un rescate para poder recuperar este.

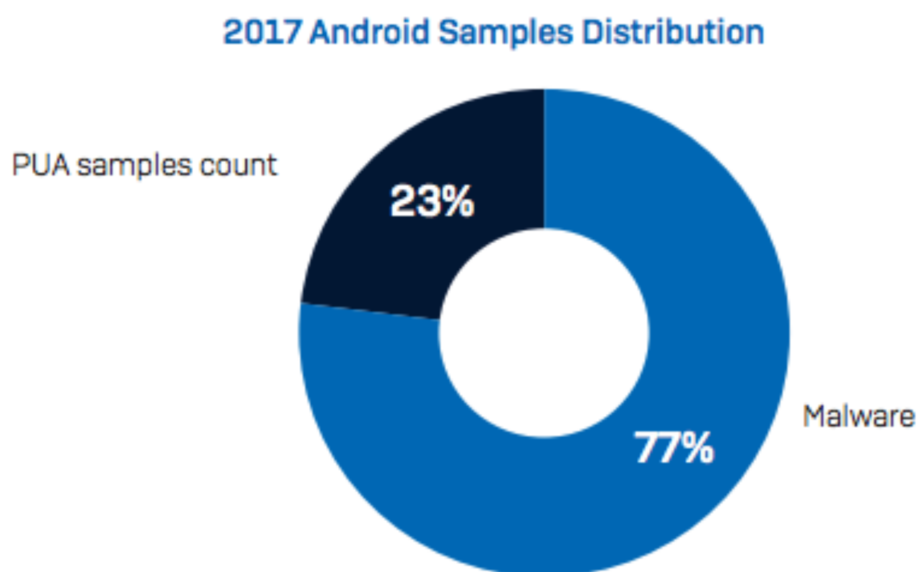
- **Troyanos bancarios:** Cada vez más, los usuarios empiezan a usar aplicaciones bancarias en sus dispositivos. Los troyanos bancarios intentan obtener los datos bancarios de los usuarios usando diferentes técnicas. Algunos de ellos atacan a una lista específica de bancos, y clonan sus ventanas de inicio de sesión para de esta forma recolectar las credenciales.

- **Minemos móviles:** Este malware usa el dispositivo para minar criptomonedas en beneficio del atacante. Al igual que los otros tipos de malware, irá oculto dentro de otra aplicación, pero en segundo plano estará minando criptomonedas. No roba datos del usuario, ni le suscribe a nada ni secuestra el dispositivo, "solo" utiliza los recursos del móvil para su cometido. Esto se traduce en que el usuario notará como la batería decrece más rápido

de lo habitual y sobrecarga al dispositivo, lo que puede llevar a que se recaliente.

La empresa Sophos, en un su informe de predicción de malware para el año 2018 (2017), estiman que para final del año 2017 habrían analizado 10 millones de aplicaciones potencialmente no deseadas (PUAs) enviadas por sus clientes, lo cual representa un aumento desde los 8,5 millones del 2016.

Figura 4. Distribución de las aplicaciones PUAs para el año 2017

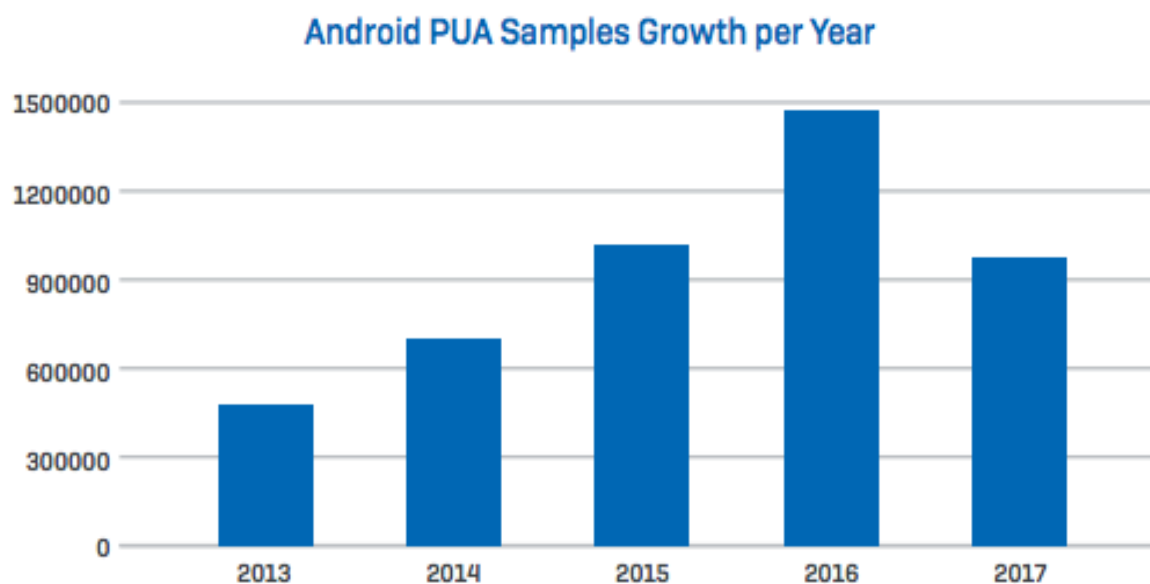


Fuente: Informe de predicción de malware para el año 2018 (2017) de la empresa Sophos.

El número total de aplicaciones maliciosas ha ido en aumento constante en los últimos cuatro años. En 2013, algo más de medio millón de muestras fueron maliciosas. Para 2015 había aumentado a algo menos de 2,5 millones. Para 2017, la cantidad es de prácticamente 3,5 millones. De estas muestras, el 77% resultaron ser malware, como podemos ver en la figura 4.

La figura 5, también muestra un descenso en el número de PUAs, aunque existía un claro aumento entre 2013 y 2016, en 2017 se redujo de 1.4 millones a menos de 1 millón.

Figura 5. Crecimiento en la detección de PUAs desde 2013 hasta 2017



Fuente: Informe de predicción de malware para el año 2018 (2017) de la empresa Sophos.

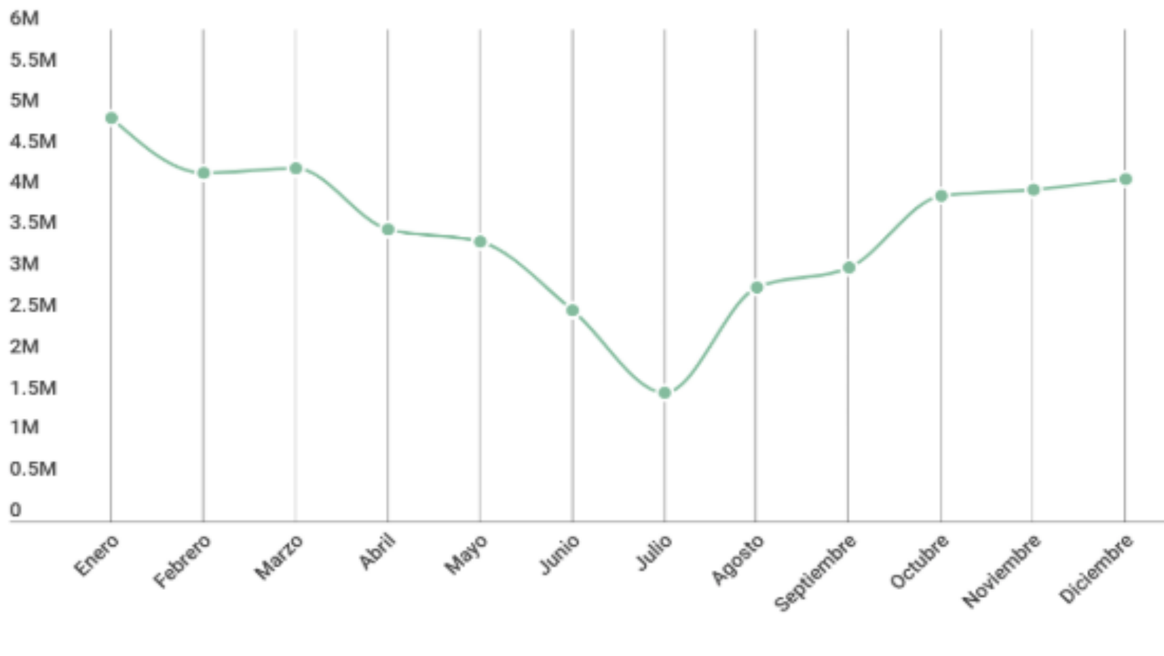
Por su lado, la empresa antivirus Kaspersky destaca que aunque el número de usuarios atacado por el malware que se aprovecha de los permisos root (dispositivos con permisos de administrador) ha disminuido en 2017, estos siguen siendo una de las mayores amenazas para los usuarios de Android. El retorno del malware que hace clic en WAP (enmarcado dentro de los suscriptores web y SMS) y el activo desarrollo de los troyanos bancarios móviles, son también algunas de las tendencias del 2017 según este informe.

En 2017 el Antivirus detectó más de 5.5 millones de paquetes de instalación de programas maliciosos, casi el 33% menos que el año anterior, y casi el doble del total del 2015.

A pesar de esta disminución, en 2017 registraron un aumento en el número de ataques software malicioso para dispositivos móviles: 42.7 millones frente a los 40 millones del 2016.

Figura 6. Ataques neutralizados por los productos de Kaspersky durante el 2017

.



Fuente: Unuchek, R. (2018). *Virología móvil 2017*. Boletín de seguridad de Kaspersky

2.3. Lucha contra el malware en Android

Desde sus inicios, Android ha ido incluyendo, a lo largo de sus actualizaciones, métodos y herramientas para dificultar, en la medida de lo posible, el desarrollo de malware, así como herramientas para luchar contra este.

Un ejemplo es la actualización a Android 6 (Marshmallow), en donde se incorporó el administrador de permisos, el cual nos da la posibilidad de administrar que permisos concederle a cada aplicación. Esto dificulta a los desarrolladores de malware crear aplicaciones con un supuesto propósito y luego pedir permisos que no correspondían y usarlos para acceder a datos privados del usuario. Si bien es verdad, que al instalar una aplicación de Google Play, te aparece la lista completa de permisos, también es verdad que poca gente revisa esta lista, pero con el nuevo

sistema, el usuario tendrá que conceder permisos expresamente a los más conflictivos.

En el informe de informe Android Security (2016), se explica el proceso que debe de pasar una aplicación desde que el desarrollador la sube a Google Play, hasta que esta se publica en el market.

Google tiene un sistema automático de detección de aplicaciones peligrosas, este sistema realiza un análisis estático y dinámico. Si alguna aplicación se detecta como posible riesgo, esta se marca como aplicación potencialmente dañina (Potentially Harmful Applications - PHA), enviándola para un análisis manual si fuera necesario.

Algunas de las vías usadas por Google para enseñar a los sistemas de ML (Machine Learning) qué aplicaciones son buenas o malas, son:

- **Análisis estático.**
- **Análisis dinámico.**
- **Análisis de terceros:** Provenientes de las diferentes empresas antivirus.
- **Relaciones del desarrollador:** No solo se estudia el código o lo que hace la aplicación, si no que se buscan ciertos datos o patrones que puedan asociar al desarrollador con otros malwares descubiertos con anterioridad.
- **Firmas:** Usan firmas de aplicaciones para compararlas contra una base de datos de malwares conocidos.
- **Heurísticas y análisis de similitud:** Analizan la similitud y comparan esta con otras buscando tendencias.
- **SafetyNet:** Este sistema proporciona información sobre la seguridad del dispositivo en el mundo real. Detectando así aplicaciones que puedan estar haciendo un uso indebido del dispositivo, como abusar de los SMS Premium,...

En este año, Google empezó también a usar un sistema de machine learning que se encarga de hacer un seguimiento de las aplicaciones marcadas como PHA, para observar sus patrones de instalación. Este se vio beneficiado de la incorporación de SafetyNet, ya que mientras los sistemas tradicionales de ML se centraban en el análisis de código, al usar SafetyNet, se pudo entrenar una red

neuronal para detectar automáticamente grupos de aplicaciones basadas en patrones de instalación.

En el informe Android Security (2017), se pone de manifiesto la reducción del número de PHAs en los dispositivos y en Google Play. Esto es en parte gracias a la incorporación de Google Play Protect, y la estrecha colaboración con los fabricantes de dispositivos, desarrolladores de los chips (SoC), operadores de telecomunicaciones, investigadores y académicos.

Gracias a esto, se ha conseguido que los dispositivos que solo descargan aplicaciones de Google Play, tienen 9 veces menos probabilidades de instalar un PHA que los que descargan de otras fuentes.

Google Play Protect es un conjunto de herramientas, que permiten mantener el dispositivo protegido en tiempo real. Utiliza los servicios de verificación de aplicaciones basados en la nube para determinar si las aplicaciones son PHA. En caso de encontrar una PHA, Google Play Protect advierte al usuario y le da la opción de inhabilitar o eliminar esta.

Para esto, antes se realizaban escaneos del dispositivo de forma regular, cada 6 días aproximadamente (si el dispositivo tenía indicadores de PHA instalados u otros factores de riesgo, se escanea con más frecuencia). A partir de 2016, se comenzó a realizar escaneos diarios, esto permite que aumente la capacidad de respuesta, minimizando los posibles daños.

Durante el 2017, además, también se hicieron avances en los sistemas de ML, utilizando, también los datos obtenidos de Google Play Protect.

Por otro lado, equipos como VirusTotal (equipo perteneciente a Chronicle Security - Google) o Koodous, usan, también sistemas de análisis estático, y recientemente están incorporando análisis dinámico. VirusTotal, por su parte, extrae estos informes, para que puedan ser usados por las distintas empresas antivirus, que están adheridos a él.

No podemos pasar por alto, que estas empresas, aunque estén implementando análisis dinámico, les falta toda la información que Google recolecta desde hace años, con herramientas como Google Play Protect.

Para finalizar, hay que decir, que hay muchos estudios donde se estudian distintos algoritmos y acercamientos a la detección de malware en Android. Muchos de ellos hacen uso de las redes neuronales. La mayor dificultad con la que nos encontramos, es que no podemos hacer una comparación real de ellos, ya que para ello deberíamos contar con un dataset único, al que se le apliquen las distintas aproximaciones y de este modo, poder tener una visión más real de cuál es la aproximación más acertada.

3. Diseño del sistema inteligente.

3.1. Extracción de características

3.1.1. Extracción de información

Para desarrollar nuestro sistema inteligente, primero necesitamos una gran cantidad de muestras, para ello usamos la plataforma VirusTotal, la cual nos ha concedido una cuenta “Intelligence” para nuestro proyecto. Para nuestro experimento hemos usado un conjunto de cerca de 33000 muestras, en este conjunto hay tanto malware como goodware.

Una vez tenemos las muestras necesitamos saber cómo extraer información de ellas, para lo cual estudiamos las técnicas actuales, y encontramos los dos principales tipos de análisis para sacar información de estas:

- **Análisis Estático:** Consiste en analizar una aplicación sin correrla. Las características de la aplicación son extraídas con distintas herramientas que decompilan el fichero APK. Uno de los mayores beneficios de este análisis es la velocidad, pero esta está limitada por las técnicas de ofuscación (técnicas para proteger y hacer más ágil el código, con las cuales se vuelve ilegible). Para hacer este tipo de análisis hay muchas herramientas, aunque la principal es AndroGuard.

- **Análisis Dinámico:** Este análisis consiste en ejecutar una aplicación en un entorno controlado (SandBox), pudiendo ver así cómo interactúa con otras aplicaciones, con el sistema y las distintas peticiones que hace a través de internet,... Este análisis, a diferencia del estático, funciona bien cuando el código está ofuscado, pero su problema es que al ser un proceso automatizado y generalizado, ejecuta un único camino, no siendo capaz de adaptarse a cada aplicación, como hacen los humanos.
  Es obvio que este análisis nos aporta información muy importante que de otra manera no se podría conseguir. Es por ello que el malware más sofisticado implementa medidas anti-sandboxing, que hacen a este detectar cuando se ejecuta en este entorno, y comportarse de forma distinta a como lo haría si se ejecuta en un dispositivo real.

3.1.2. Construyendo nuestro extractor de características

Tras investigar las diferentes opciones que disponíamos, y las limitaciones del equipo, que se iba a usar en la extracción de estas características, consultamos a los equipos de VirusTotal y Koodous, ambos especializados en la detección de malware, y se llegó a la conclusión de que la mejor opción era optar por una análisis estático, usando la librería Androguard como núcleo para la extracción.

Androguard es una librería muy potente, pero también muy pesada, por tanto, desde un inicio teníamos claro que se podría convertir en el cuello de botella de nuestro proyecto. Para ejemplificar esto, el tiempo aproximado de generar los reportes de 1000 muestras puede ser entorno a unas 24 horas.

En un primer momento, se desarrollaron los scripts para sacar todas las características, tratarlas y que nos ofreciera un dataset final para usar. Pero tras analizar estos dataset y hacer varias pruebas, descubrimos que había características que no se habían tenido en cuenta, y que podrían ser muy interesantes. Por tanto, debido al tiempo que supone, no podíamos estar continuamente extrayendo todas las características desde el fichero. Para solucionar esto, se optó por sacar un reporte inicial (en bruto) de cada app, en el cual se saca el máximo número de características, aunque luego no se vayan a usar todas.

Con esta misma filosofía, se desarrollaron varios scripts a lo largo de los cuales se iba transformando este reporte hasta convertirlo en un dataset, facilitando esto el que si en algún momento se quiere modificar algo, solo se tenga que ir al script que saca ese dato y modificarlo, sin tener que rehacer todo el análisis desde 0.

Aún con estas modificaciones, la extracción de características fue un proceso lento, que duró varios meses, sumándole a que por problemas técnicos perdimos todas las muestras que teníamos y tuvimos que descargar otra vez las cerca de 400GB de muestras. Debido a este contratiempo, se optó por mejorar los algoritmos de extracción para permitir realizar la descarga de aplicaciones en paralelo con la extracción de características.

Para seleccionar las características más interesantes, nos hemos basado en los reportes que proporcionan VirusTotal y Koodous, así como el conocimiento propio en la plataforma Android y el adquirido a la hora de trabajar conjuntamente con estos equipos en el desarrollo de sus aplicaciones Android.

3.1.3. Estudio del dataset

Una vez tenemos un dataset completo, con todas aquellas características que consideramos importantes, usamos Weka[1], para ver cuán representativas son estas.

A continuación se muestran las gráficas de varias características, en relación a lo decisivas que han sido a la hora de clasificar malware.

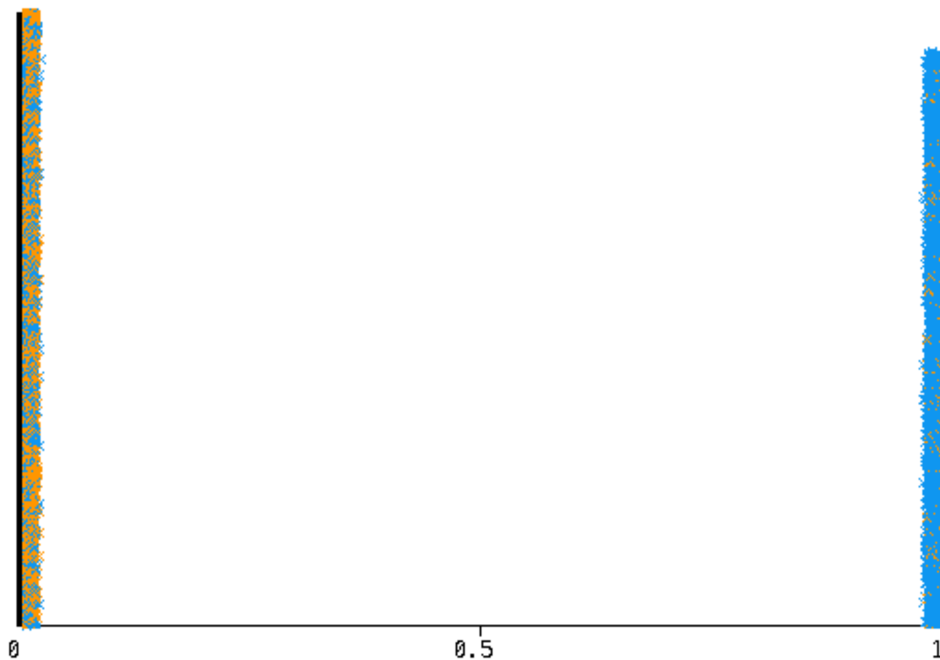
Se mostrará en naranja el malware, en azul el goodware, cada punto es una muestra, para lo cual se usa el hash (parámetro único). En el eje ‘x’ se sitúa la característica a estudiar en cada momento.

En primer lugar mostramos la gráfica para la característica “gplaydata_exists” lo cual indica si ese nombre de paquete (no tiene por qué ser la misma muestra) está en Google Play.

Como podemos apreciar en la figura 7, que una aplicación se encuentre en Google Play, o que una con el mismo nombre de paquete, es un buen indicador de que la muestra no será malware.

[1] Según la página web de Weka, éste programa *“es una colección de algoritmos de machine learning y minería de datos. Este contiene herramientas para la preparación de los datos, clasificación, regresión, clustering, reglas de asociación y visualización*”

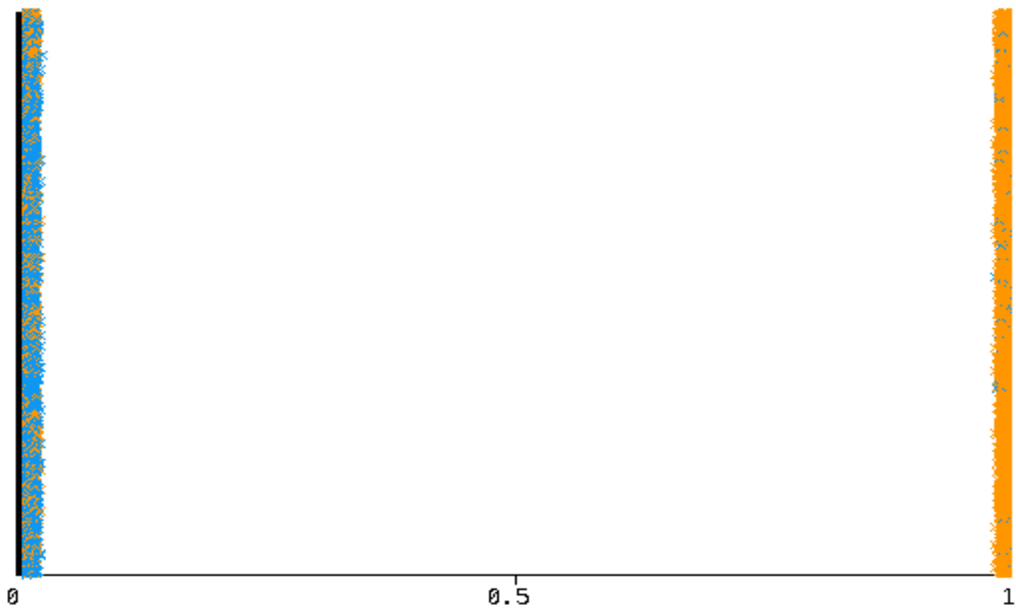
Figura 7. Distribución del malware con respecto a la variable gplaydata_exists



Fuente: Elaboración propia

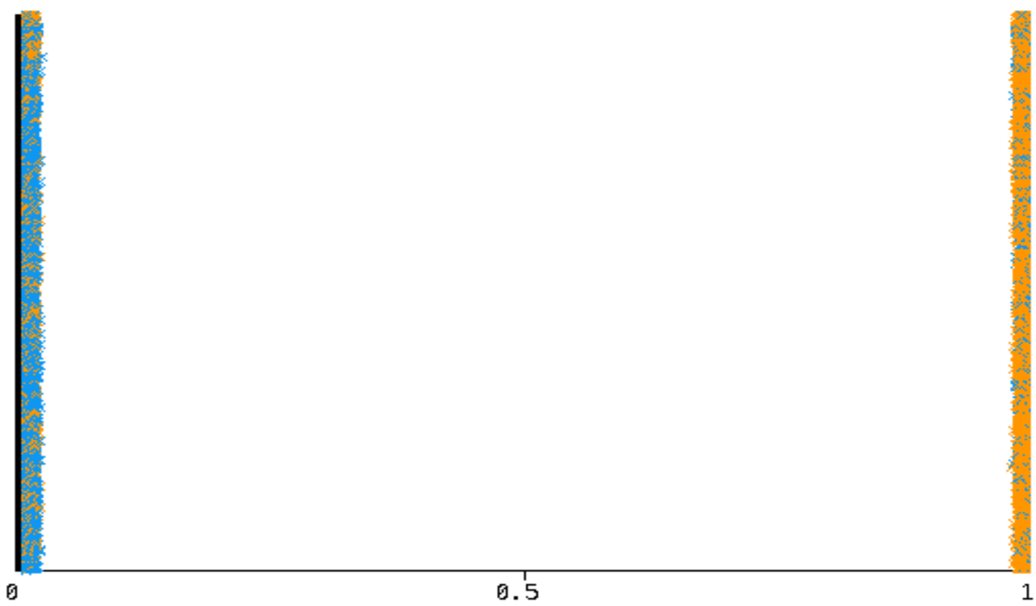
Por otro lado, como era de esperar, que una aplicación tuviera ciertos permisos, iba a ser un buen indicador de que esta es malware, es el caso de, por ejemplo el permiso “RECEIVE_MMS”, “SEND_SMS”, “MOUNT_FORMAT_FILESYSTEM”, “DISABLE_KEYGUARD”, “INSTALL_SHORTCUT”,... Los cuales, si bien hay muchas aplicaciones que hacen uso de ellos, sin tener que ser malware, es un muy buen indicador de posible malware. Esto podemos apreciarlo en las figuras 8 y 9.

Figura 8. Distribución del malware con respecto a la variable receive_mms



Fuente: Elaboración propia

Figura 9. Distribución del malware con respecto a la variable send_sms



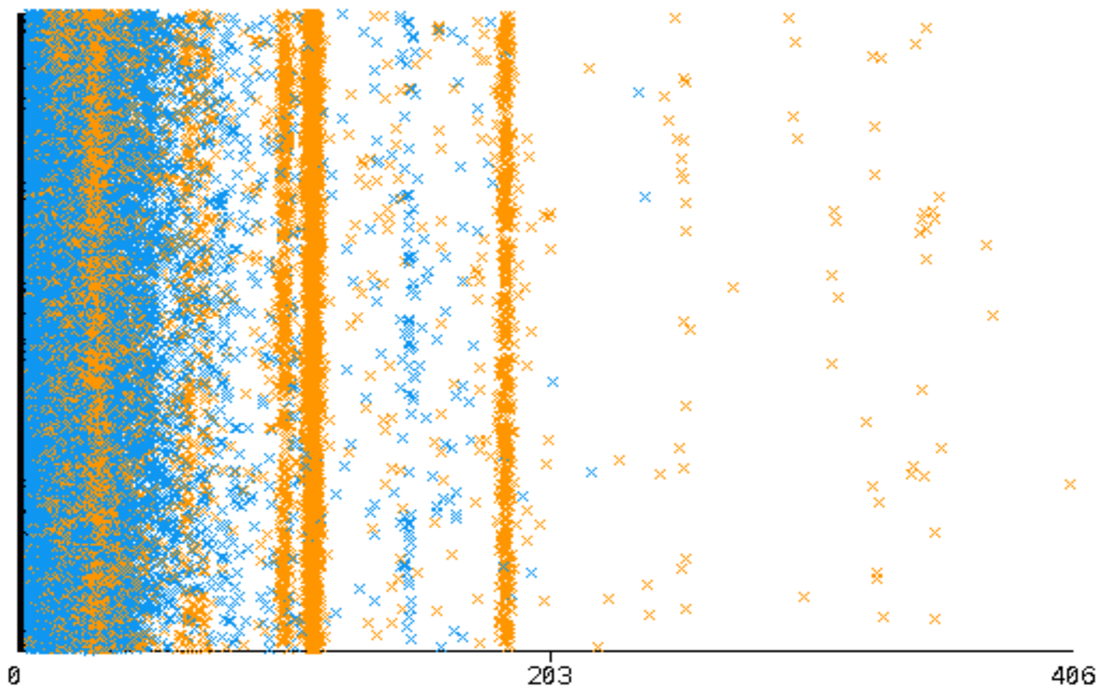
Fuente: Elaboración propia

Por otro lado, encontramos características, en las cual el malware se comporta de manera similar. Como por ejemplo “permission_total” que hace referencia al número total de permisos solicitados por una aplicación.

En la figura 10 podemos observar que, aunque el malware y goodware están por partes iguales en algunos puntos, hay otros puntos donde se concentra el malware, esto da que pensar que igual las agrupaciones se refieren a familias de malware.

También se deduce que una aplicación con más de 100 permisos (donde vemos las dos columnas de malware) es una clara indicación de que es malware, ya que el goodware tiende a limitar el número de permisos.

Figura 10. Distribución del malware con respecto a la variable permissions_total

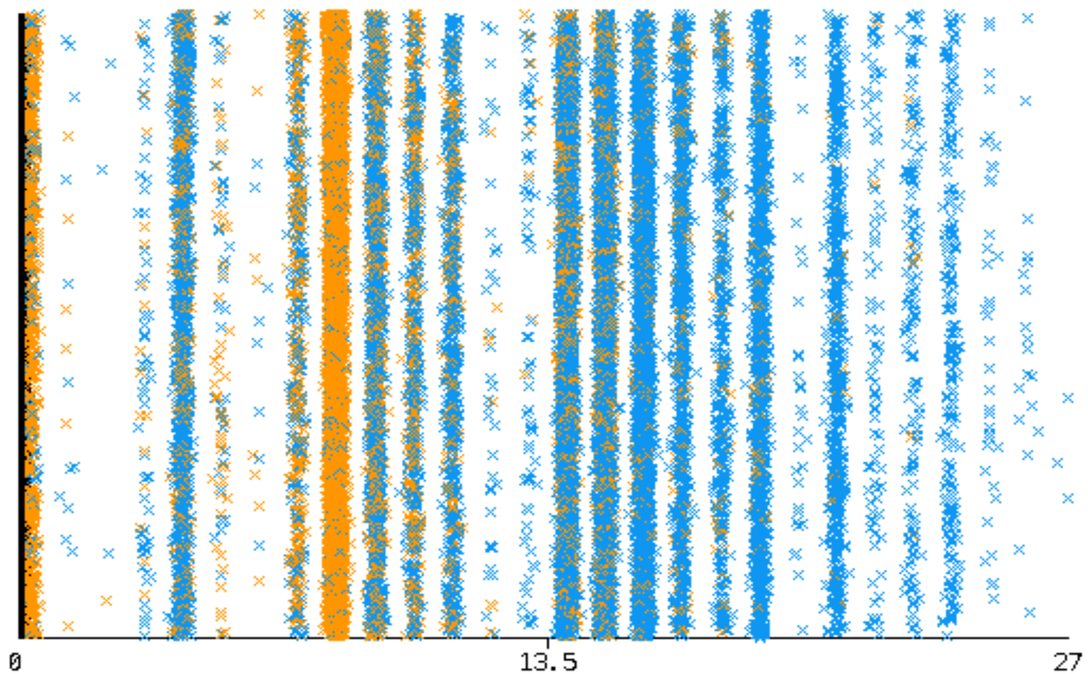


Fuente: Elaboración propia

Un caso similar ocurre con la característica “general_minimun_sdk_version” la cual, como se puede imaginar, hace referencia a la versión mínima que soporta la aplicación. Es normal que un malware intente dar soporte al mayor número de versiones para llegar al máximo de dispositivos. Por eso en la gráfica se ve una concentración tan grande con el min sdk 8, la cual corresponde con Android 2.2 (Froyo) la cual fue una de las versiones más extendidas.

De igual modo, como podemos ver en la figura 11, es muy difícil ver un malware usando como min sdk las últimas versiones del SDK, ya que el número de dispositivos que podrían ejecutarla se vería drásticamente reducido

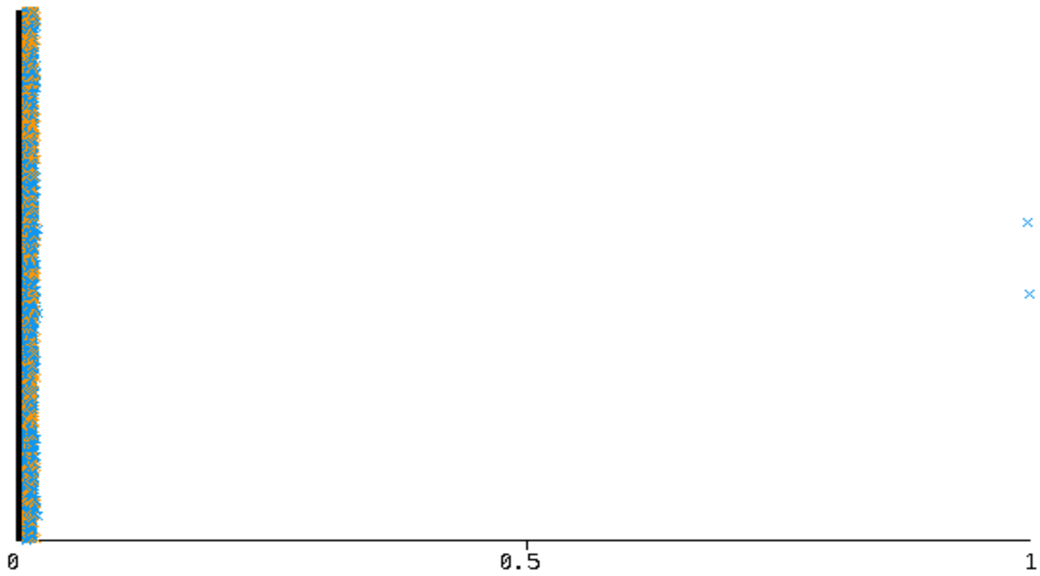
Figura 11. Distribución del malware con respecto a la variable general_minimun_sdk_version



Fuente: Elaboración propia

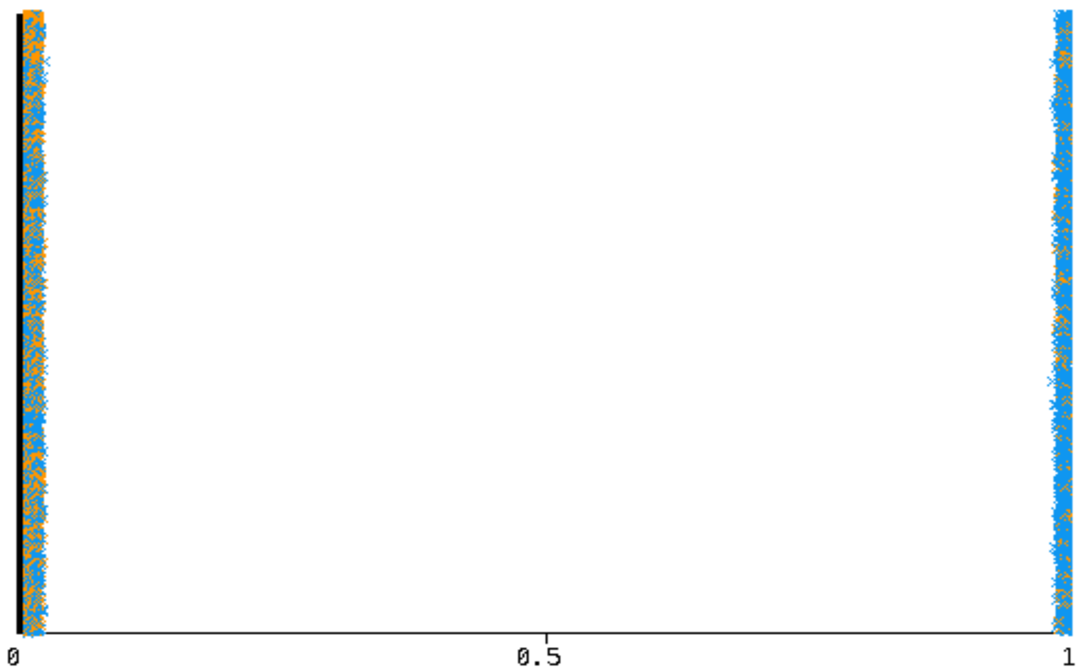
También hemos detectado que existen características que no son resultan muy decisivas, como es el caso de la procedencia, dato que se saca de la firma. Esto es algo que se esperaba desde el inicio, ya que existen países en donde apenas se desarrollan aplicaciones, como el caso de Grecia (figura 12), y otros donde se desarrollan gran cantidad de estas (goodware y malware), como el caso de los EEUU (figura 13). Pero teníamos que cubrir todas las posibilidades a la hora de realizar el estudio, y por eso están presentes todas las posibilidades (países y continentes), ya se encargará la red de tenerlos más o menos en cuenta.

Figura 12. Distribución del malware con respecto a la variable certificate_countr_name_gr



Fuente: Elaboración propia

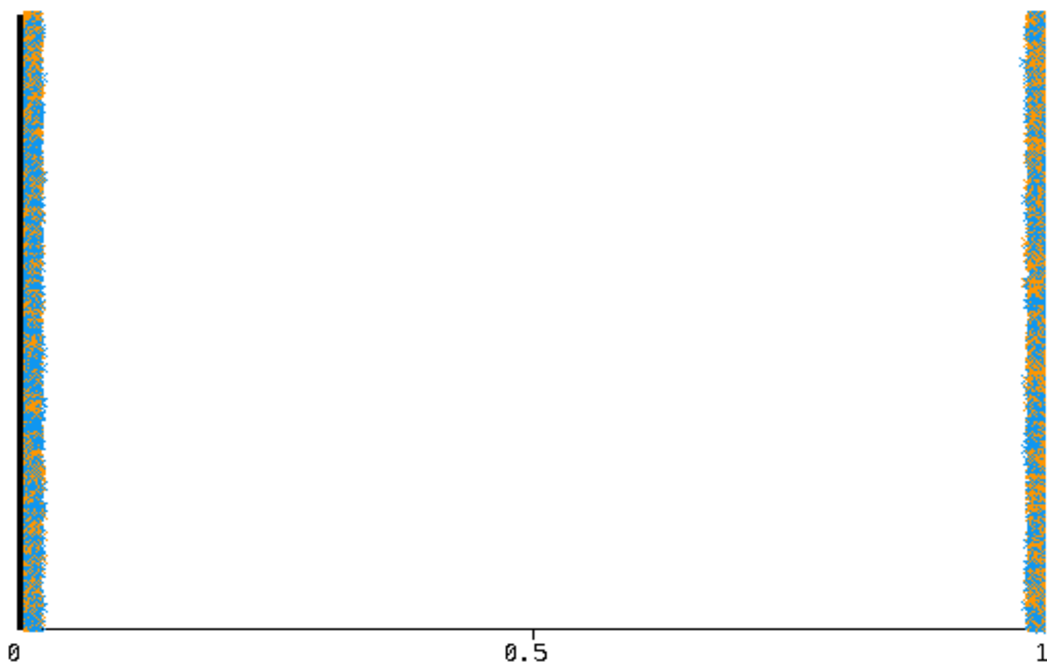
Figura 13. Distribución del malware con respecto a la variable certificate_countr_name_us



Fuente: Elaboración propia

Algunas nos han llegado a sorprender, como es el caso del permiso "RECEIVE_BOOT_COMPLETED", figura 14, que si bien es verdad que es un permiso muy usado en goodware, ya que lo que permite es que la aplicación pueda ejecutar código cuando se enciende el dispositivo, esperaba que este fuera más ampliamente usado en malware. Y como se puede ver en la gráfica, está bastante nivelado el uso de este permiso.

Figura 14. Distribución del malware con respecto a la variable permission_receibe_boot_completed



Fuente: Elaboración propia

Tras consultar diferentes fuentes, encontramos en un artículo de Emerging Threats, que un malware que usa este permiso es, MazarBot, del cual podemos ver a continuación un reporte de VirusTotal en la figura 15.

Figura 15. Resultado de un ejemplo de MazarBot analizado en VirusTotal

SHA256: 73c9bf90cb8573db9139d028fa4872e93a528284c02616457749d40878af8cf8

Nombre: malware.apk

Detecciones: 35 / 61

Fecha de análisis: 2018-08-27 00:04:59 UTC (hace 3 semanas, 6 días)

Análisis | Detalles | Información adicional | Comentarios 3 | Votos | Información

The file being studied is Android related! APK Android file more specifically. The application's main package nam number of the application is **1**. The displayed version string of the application is **1.0**. The minimum Android API leve (MinSDKVersion) is **9**. The target Android API level for the application to run (TargetSDKVersion) is **22**.

Required permissions

android.permission.SEND_SMS (*send SMS messages*)

android.permission.RECEIVE_BOOT_COMPLETED (*automatically start at boot*)

android.permission.INTERNET (*full Internet access*)

android.permission.SYSTEM_ALERT_WINDOW (*display system-level alerts*)

android.permission.WRITE_SMS (*edit SMS or MMS*)

android.permission.ACCESS_NETWORK_STATE (*view network status*)

android.permission.WAKE_LOCK (*prevent phone from sleeping*)

android.permission.GET_TASKS (*retrieve running applications*)

android.permission.CALL_PHONE (*directly call phone numbers*)

android.permission.RECEIVE_SMS (*receive SMS*)

android.permission.READ_PHONE_STATE (*read phone state and identity*)

android.permission.READ_SMS (*read SMS or MMS*)

Fuente: VirusTotal (2018) *Reporte de la muestra Mazar BOT.*

3.2. Estudio y desarrollo de los diferentes algoritmos.

Desde un primer momento, sabíamos que queríamos un sistema compuesto de dos partes, un sistema con un cluster, que agrupa las distintas aplicaciones y una red neuronal.

Al inicio se propuso un sistema de clusters, el cual usaría todo el dataset, creando tanto clusters de goodware, malware como de una mezcla de ambos. Tras discutirlo, llegamos a la conclusión de que era mejor crear un cluster para el goodware, para el cual solo se usarían las muestras goodware del dataset, de modo que si una muestra no se podía asignar a ningún cluster existente, posiblemente fuera malware. Conforme se fue desarrollando, viendo lo difícil que es clusterizar el goodware, se optó por crear dos sistemas de clusters (uno entrenado con las muestras goodware y otro con las muestras malware), y que cooperaran entre ellos.

Para la red neuronal se tenía claro, se entrenaría con todo el dataset. Solo hacía falta escoger los parámetros adecuados para la formación de esta.

Para el desarrollo, tanto de los sistemas de clusters como de la red neuronal, se ha realizado una separación de nuestro dataset. Siguiendo el corte de 80% para entrenamiento (X_train, y_train), y 20% para testeo (X_test, y_test).

3.2.1. Desarrollando el sistema de Clusters

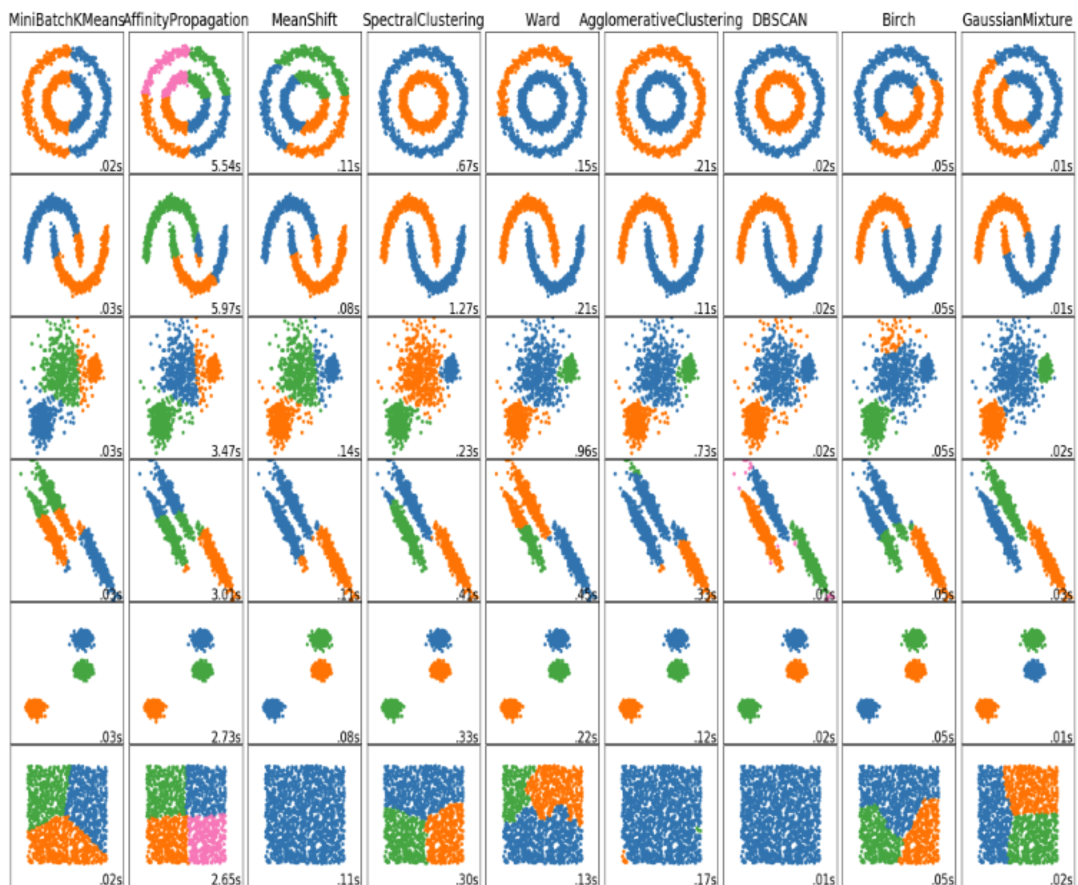
Para desarrollar el sistema de cluster, en principio nos fijamos en **DSCAN**. En la figura 16 vemos una comparación de este con otros algoritmos.

Para evaluar los distintos algoritmos y sus configuraciones, usamos el coeficiente Silhouette:

The silhouette value is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The silhouette ranges from −1 to +1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters. If most objects have a high value, then the clustering configuration is appropriate. If many points have a low or negative value, then the clustering configuration may have too many or too few clusters.

Silhouette (Clustering), s.f.

Figura 16. Comparativa de algunos de los más famosos algoritmos de clustering.



Fuente: Scikit-Learn. *Comparing different clustering algorithms on toy datasets.*

Para realizar nuestras pruebas, creamos dos subconjuntos a partir del conjunto de entrenamiento (X_train), estos dos subconjuntos serán el subconjunto de goodware y el subconjunto de malware. Cada uno de estos se usará para la creación de uno de los clusters.

Al realizar las pruebas, vimos como no obtuvimos buenos resultados. En la tabla 2 se muestran las configuraciones de los mejores puntuaciones que pudimos obtener con este algoritmo, junto a su correspondiente coeficiente de Silhouette. En ellos se aprecia que con este algoritmo, o al menos con los parámetros que se probaron, solo conseguimos un coeficiente Silhouette cercano al 0, lo cual implica que las muestras están muy cerca del límite de decisión entre dos clusters vecinos.

Tabla 2. Resultados obtenidos de las pruebas con DBSCAN

Min Samples	eps	Nº Clusters	Silhouette Coefficient
2	2	1597	-0,022
2	2,5	1513	-0,026
2	1,5	1524	-0,032
2	2,1	1525	-0,036
2	2,8	1507	-0,037
2	3	1470	-0,041
2	0,9	1449	-0,057
2	0,8	1451	-0,058
3	2,5	623	-0,118

Fuente: Elaboración propia

Por lo tanto, tras estudiar otros algoritmos, nos decantamos por **HDBSCAN**, el cual ha sido desarrollado por algunas de las personas que escribieron el paper original de DBSCAN, puliendo en este algunos de los problemas que se encontraban en DBSCAN.

En la tabla 3 podemos observar algunas de las pruebas que se realizaron con HDBSCAN.

Para las pruebas utilizamos dos conjuntos de testeo (uno de goodware y otro de malware) que el sistema no ha visto antes.

Tabla 3. Resultados obtenidos de las pruebas con HDBSCAN, entrenado con goodwares

Min Cluster Size	Min Samples	Metric	N Clusters	N Outliers	Silhouette Coefficient	Acc Detectando Goodware (conjunto de goodware)	Acc Detectando Malware (conjunto de malware)	Acc Total
20	5	euclidean	171	6233	0,001	59.5%	92%	75,75%
10	1	euclidean	386	5542	0,058	62%	91.5%	76.75%
3	2	euclidean	236	5576	0,033	52.5%	93%	72.75%

Fuente: Elaboración propia

Tras hacer pruebas el sistema de clusters mejor posicionado, según el coeficiente Silhouette, vemos que detecta un 62% de las muestras como pertenecientes a alguno de los clusters, en el conjunto de goodware, mientras que en el conjunto de malware, detecta un 91.5% como outliers (puntos que no pertenecen a ningún cluster).

Tras ver estos resultados, se decidió crear un sistema de cluster en paralelo, entrenado con malware en lugar de goodware, en la tabla 4 se ven los parámetros y resultados del ganador:

Tabla 4. Resultados obtenidos de las pruebas con HDBSCAN, entrenado con malwares

Min Cluster Size	Min Samples	Metric	N Clusters	N Outliers	Silhouette Coefficient	Acc Detectando Goodware (conjunto de goodware)	Acc Detectando Malware (conjunto de malware)	Acc Total
10	1	euclidean	345	2359	0,314	84.5%	79.5	82%

Fuente: Elaboración propia.

Al igual que en el sistema anterior, entrenado con goodware, en este realizamos las pruebas con conjuntos de testeo, tanto de goodware como de malware. Obteniendo que si se prueba con el conjunto de goodware, nos da un 84.5% de outliers, mientras que si usamos el conjunto de malware, obtenemos un 79.5% de puntos asignados a algún cluster del sistema.

Nuestro sistema de clusters, finalmente está compuesto por dos sistemas, el sistema entrenado con goodware y el sistema entrenado con malware.

Tras estudiar varias opciones acerca de cómo ambos sistemas debían cooperar, nos decantamos por la siguiente opción:

Cuando una muestra llega al sistema, esta es estudiada por ambos sistemas de la siguiente forma:

1. Si el sistema de malware la detecta como perteneciente a alguno de sus clusters, y el sistema de goodware la detecta como un outilier, la muestra es marcada como malware.

2. En caso contrario, la muestra será calificada como diga el sistema de goodware, que puede ser goodware u outlier.

3.2.2. Desarrollando la red neuronal

Para desarrollar la red neuronal, se optó la librería Keras, ya que ya había trabajado con ella y permite que el desarrollador se pueda centrar en la red neuronal y las configuraciones en lugar de en cómo implementar esta.

La red neuronal se compone de una capa de entrada con 414 neuronas, tres capas ocultas, de 500, 500 y 300 neuronas, que usan como la función relu como activación y una capa de salida con una única neurona activada por la función sigmoid.

En cada capa oculta se aplica una configuración de Dropout, la cual consiste en que, aleatoriamente (configurado por el parámetro “rate”) a algunas neuronas les llegue 0 como valor de entrada. Esto ayuda a evitar el overfitting (que se sobreajuste al conjunto y no sea capaz de generalizar para detectar ejemplos nuevos).

A la hora de compilarla usamos la binary_crossentropy como función de pérdida, accuracy como función para juzgar el modelo, y usaremos validación cruzada (cv) de 10 pliegues.

Una vez elegidos esta configuración, procedemos a elegir los hiper-parámetros (parámetros de configuración que se usan tanto en la creación de la red neuronal como en el entrenamiento). Este proceso se conoce como Hyperparameter Tuning.

Los parámetros a optimizar son:

- **optimizer** (adam, rmsprop): Optimizador que usaremos a la hora de compilar.
- **learn_rate** (0.001, 0.01): Tasa de aprendizaje que usará nuestro optimizador.
- **rate** (0.2, 0.3): Tasa que se usará en la configuración Dropout.
- **batch_size** (10, 50, 100): Tamaño del lote que se usará para entrenar.
- **epochs** (10, 25, 50): Número de épocas que durará el entrenamiento.

El proceso de optimización se basa en coger todos estos parámetros, entrenar todas las combinaciones, y devolver la ganadora. Este es un proceso muy lento, y tardará más, cuantas más parámetros y/u opciones se quieran probar.

A continuación, en la tabla 5 se muestran algunos de los resultados obtenidos de este proceso, utilizando para entrenar el subconjunto X_train. Se han seleccionado algunos de los mejores y algunos de los peores resultados para que se pueda apreciar el contraste.

Tabla 5. Resultados obtenidos para las distintas configuraciones de la red neuronal

Batch Size	Epochs	Learn Rate	Optimizer	Dropout Rate	Mean Score	Std Score
100	10	0.001	'adam'	0.3	0.924567	0.003995
100	50	0.001	'adam'	0.3	0.924491	0.003019
50	50	0.001	'rmsprop'	0.2	0.924377	0.003110
50	10	0.001	'adam'	0.3	0.924149	0.004227
100	10	0.001	'rmsprop'	0.2	0.923845	0.003577
50	25	0.001	'adam'	0.3	0.923769	0.004010
50	10	0.001	'adam'	0.2	0.923731	0.003531
10	50	0.001	'adam'	0.3	0.923617	0.003461
100	50	0.01	'rmsprop'	0.3	0.572280	0.007842
10	50	0.01	'rmsprop'	0.2	0.560962	0.117935
100	10	0.01	'rmsprop'	0.2	0.560544	0.040254
100	10	0.01	'rmsprop'	0.3	0.559480	0.041810
50	50	0.01	'rmsprop'	0.3	0.556746	0.045452
100	25	0.01	'rmsprop'	0.3	0.556138	0.046201
50	10	0.01	'rmsprop'	0.3	0.545313	0.056857
50	10	0.01	'rmsprop'	0.2	0.539122	0.061282
10	10	0.01	'rmsprop'	0.2	0.530158	0.066155

Fuente: Elaboración propia

Podemos ver que, entre las mejores combinaciones, hay parámetros que se repiten. Estos son: batch_size: 100, learn_rate: 0.001, optimizer: adam y dropout_rate: 0.3. De hecho, si nos fijamos en el parámetro learn_rate, nos damos cuenta que en la parte superior de la tabla está fijado a 0.001 y en la parte inferior a 0.01. Lo cual nos da a entender que es un parámetro decisivo.

Mientras que en la parte alta de la tabla, vemos precisiones cercanas al 93%, en la parte baja, podemos ver otras cercanas al 50%, lo que sería igual a usar una

moneda para decidir si una muestra es malware o goodware. Esto pone de manifiesto la importancia de elegir correctamente los hiper-parámetros, que para un mismo dataset, pueden darnos resultados tan dispares.

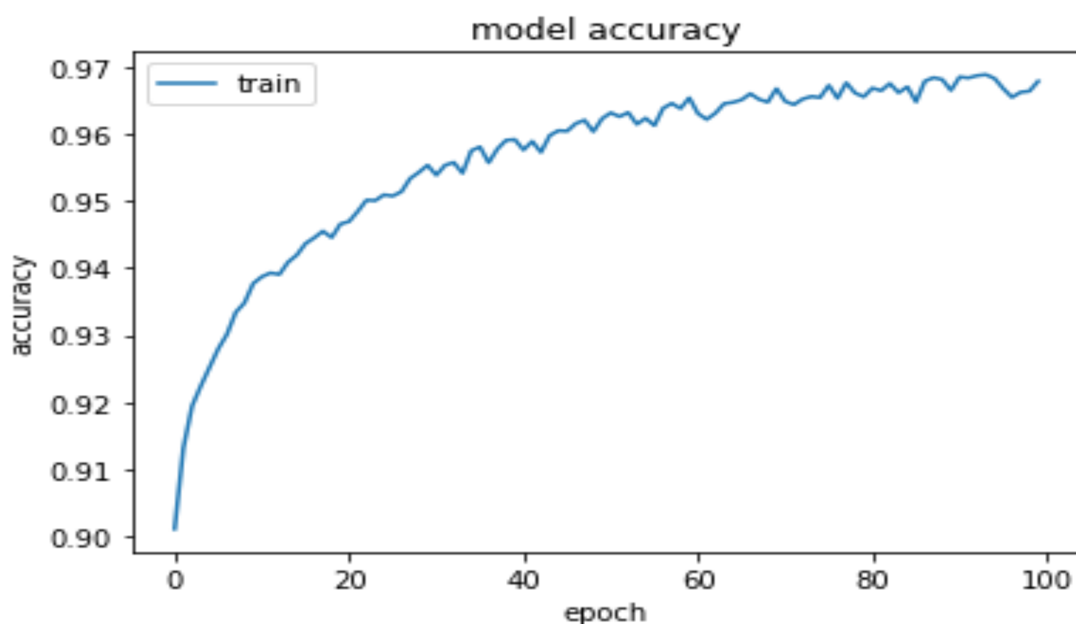
Ahora se reentrena el modelo, con los parámetros obtenidos del proceso de tuning, y se realizan las predicciones para el conjunto de testeo (previamente separado, y por lo tanto la red nunca lo vio) dándonos una precisión de 94.3%, lo cual quiere decir que ha generalizado bien la red. Al ver este resultado, recordamos que antes de empezar la búsqueda de hiper-parámetros, se realizó una configuración básica para poder trabajar en paralelo, y que la precisión de dicha red, con el conjunto de testeo, era superior a la recién obtenida.

Se procede a estudiar las diferencias, y encontramos que el batch_size es 20 en lugar de 100, y las epochs 100 en lugar de las 10 de nuestra ganadora. Al comprobar, vemos que, precisamente estas opciones no estaban entre los parámetros que se les dio al algoritmo de tuning.

Al realizar a nuestra red ganadora estos cambios y reentrenarla, obtenemos un 97.49% de precisión, superando así a nuestra ganadora y a nuestra red básica.

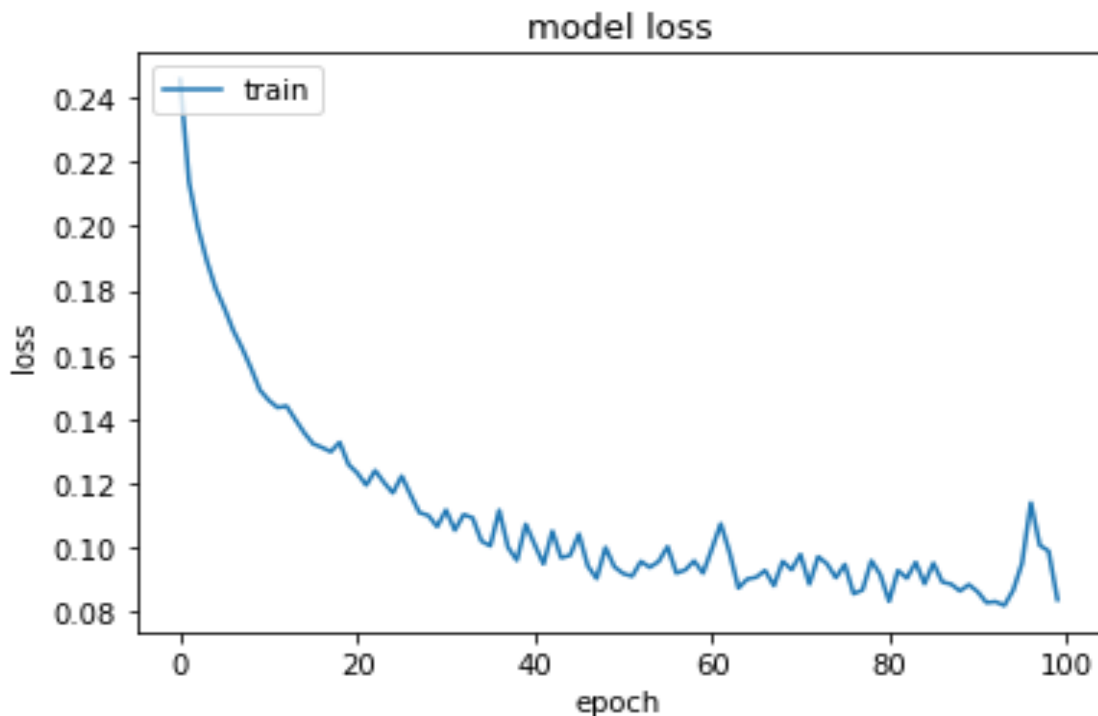
En las figuras 17 y 18 podemos ver tanto la evolución de la precisión como de la pérdida durante el entrenamiento de nuestra red.

Grafico 17. Evolución de la precisión durante el entrenamiento de la red neuronal



Fuente: Elaboración propia

Grafico 18. Evolución de la pérdida durante el entrenamiento de la red neuronal



Fuente: Elaboración propia

Una vez obtenidos nuestros mejores parámetros, hicimos una prueba añadiendo una capa extra, la cual empeoró los resultados.

Hay que tener en cuenta que el resultado de la precisión es haciendo el corte en 0.5, es decir, si realizamos una predicción de una muestra, esto nos devuelve un resultado entre 0 y 1, dando como malware si este resultado es mayor a 0.5.

Como nosotros queremos un sistema más agresivo, podemos variar este límite (threshold). Para nosotros, un resultado será positivo (goodware) cuanto más se acerque al 0 y negativo (malware) cuanto más se acerque cercano al 1.

Para estudiar cómo afectan estos cambios, estudiaremos sus matrices de confusión, figura 19.

Figura 19. Descripción de los componentes de la matriz de confusión

		Predicted class	
		P	*N*
Actual Class	*P*	True Positives (TP)	False Negatives (FN)
	N	False Positives (FP)	True Negatives (TN)

Fuente: MLEXTEND, *Confusion Matrix.*

A continuación, en las figuras 20, 21 y 22 se muestra la matriz de confusión para distintos valores de este límite.

Para evaluar las matrices de confusión utilizaremos la **precisión** o “accuracy” (ACC), la **exactitud** o “Precision” (PPV) y **sensibilidad** o “recall”, también conocida como Tasa de Verdaderos Positivos (TPR), siguiendo el articulo Confusion Matrix (2018),

La **precisión** es la proporción del número total de predicciones que fueron clasificadas correctamente:

$$ACC = \frac{TP + TN}{TP + FN + FP + TN}$$

La **exactitud** se refiere a la proporción de clasificaciones positivas que fueron correctamente identificadas como tal.

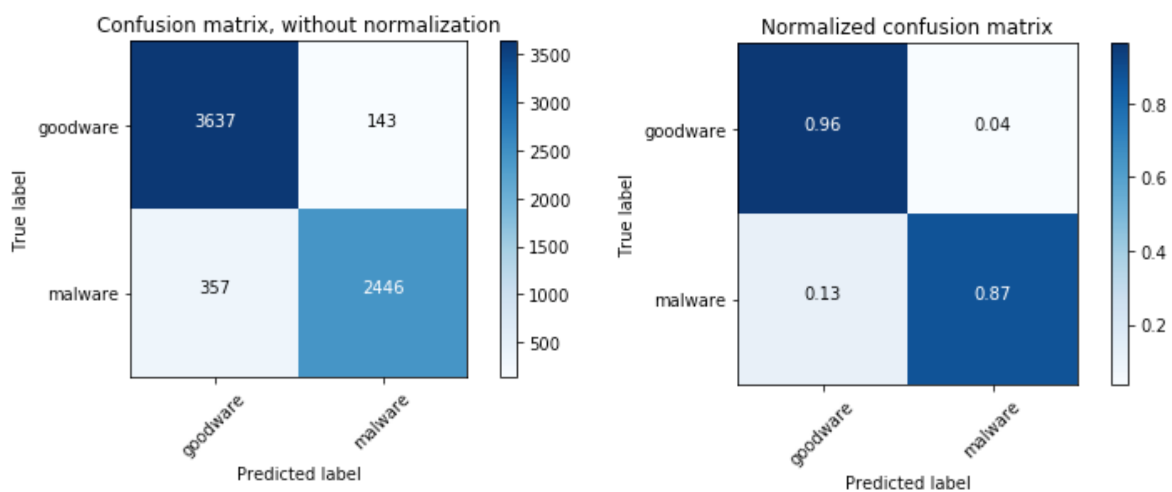
$$PPV = \frac{TP}{TP + FP}$$

La **sensibilidad** es la proporción de casos positivos que fueron correctamente clasificados.

$$TPR = \frac{TP}{TP + FN}$$

En la figura 20 podemos ver la matriz de confusión (CM) estableciendo el corte en 0.5:

Figura 20. Matriz de confusión para la red neuronal, estableciendo el límite en 0.5



Fuente: Elaboración propia

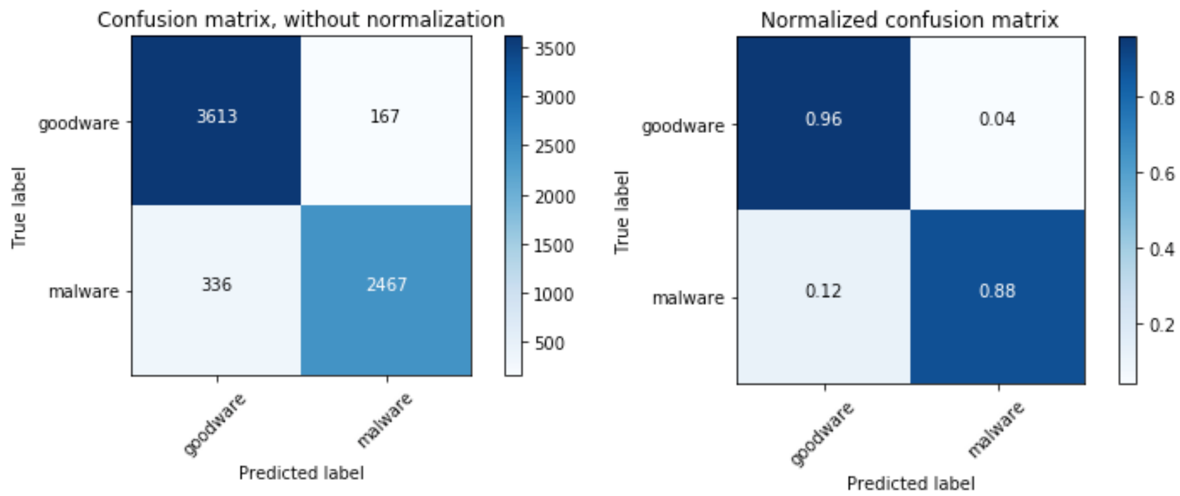
$$ACC = \frac{3637 + 2446}{3637 + 143 + 357 + 2446} = 0.924 = 92.4\%$$

$$PPV = \frac{3637}{3637 + 357} = 0.9106 = 91.1\%$$

$$TPR = \frac{3637}{3637 + 143} = 0.9621 = 96.22\%$$

En la figura 21 podemos ver la matriz de confusión (CM) estableciendo el corte en 0.4:

Figura 21. Matriz de confusión para la red neuronal, estableciendo el límite en 0.4



Fuente: Elaboración propia

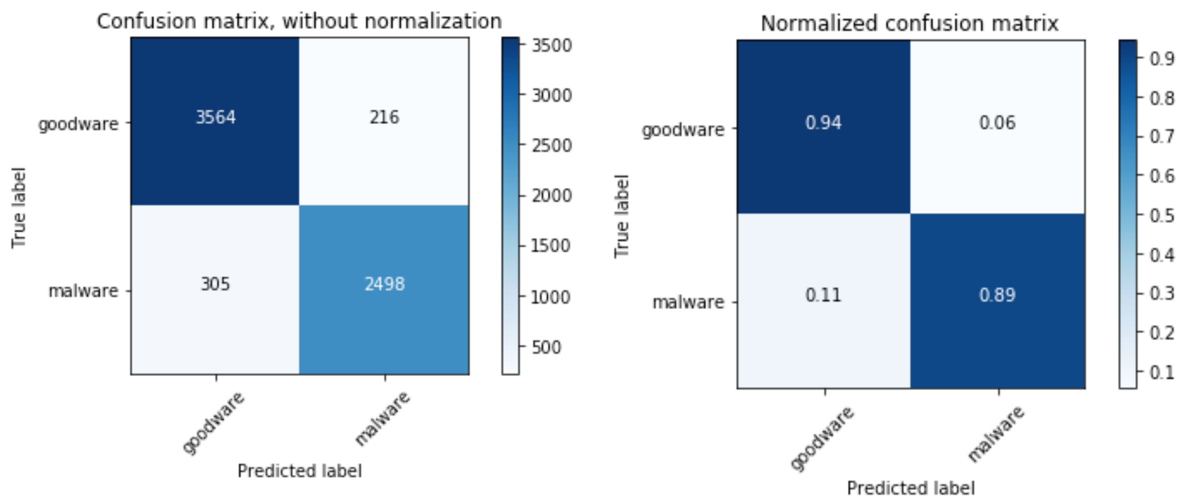
$$ACC = \frac{3613 + 2467}{3613 + 167 + 336 + 2467} = 0.9236 = 92.36\%$$

$$PPV = \frac{3613}{3613 + 336} = 0.9149 = 91.5\%$$

$$TPR = \frac{3613}{3613 + 167} = 0.9558 = 95.6\%$$

En la figura 22 podemos ver la matriz de confusión (CM) estableciendo el corte en 0.3:

Figura 22. Matriz de confusión para la red neuronal, estableciendo el límite en 0.3



Fuente: Elaboración propia

$$ACC = \frac{3564 + 2498}{3564 + 216 + 305 + 2498} = 0.9208 = 92.1\%$$

$$PPV = \frac{3564}{3564 + 305} = 0.9211 = 92.12\%$$

$$TPR = \frac{3564}{3564 + 216} = 0.9428 = 94.3\%$$

Como podemos observar, según donde vamos estableciendo el corte, los valores de la precisión (ACC), la exactitud (PPV) y la sensibilidad (TPR) van cambiando. Con un límite conservador, 0.5, la precisión está en 92.4%, la exactitud está en 94.5% y la sensibilidad en 87.26%, mientras que si establecemos un límite más agresivo, como en 0.3, la precisión se reduce hasta 92.1%, la exactitud hasta 92% y la sensibilidad aumenta hasta el 94.3%.

Esto se debe a que al reducir el límite estamos haciendo que aquellas muestras que la red neuronal había clasificado cerca del límite, entre positivos y

negativos, se queden a un lado u otro. Al establecerlo a 0.3, todas las muestras entre 0.3 y 0.5, antes clasificadas como goodware, pasan a ser clasificadas como malware.

Esta forma de actuar se realiza cuando se quiere que la red neuronal detecte el máximo, y no importa que clasifique ciertas muestras que se encuentran en el límite como positivos. Ya que en este caso es preferible que algún goodware se detecta como malware a decirle al usuario que un malware es goodware y este lo instale con la confianza de que no hay peligro.

3.3. Evaluación de resultados

Al desarrollar el sistema, se quería que este fuera capaz de adaptarse a los cambios en las tendencias del malware. Por eso, en un primer momento, se pensó en desarrollar un sistema tanto de clusters como de redes neuronales mediante entrenamiento continuo, de modo que la red no tuviera que entrenarse desde 0.

Al ir desarrollando, y ver el proceso de normalización que experimentan los datos, en el cual se adaptan al dataset proporcionado, comprendimos que si se normaliza teniendo esa premisa en mente, perdemos precisión sobre las muestras actuales, pensando en un posible futuro. Por tanto, se llegó a la conclusión de que es preferible un reentrenamiento de los sistemas cuando se tenga cierto volumen de muestras nuevas. Esto quedó respaldado tras ver que el proceso de entrenamiento, una vez se tienen los hiper-parámetros correctos, no es tan costoso, y por tanto el sistema siempre está adaptado al contexto actual del malware.

El sistema de clusters goodware, ha sido desarrollado con la idea de que detecte esas posibles muestras que se escapan a la media de muestras calificadas como goodware, el problema aquí es que el goodware no es tan fácilmente clusterizable como el malware, el cual responde más a familias con técnicas similares, o son simplemente modificaciones de un mismo malware. Aun teniendo un porcentaje bajo, el sistema de clusters de goodware es muy importante para nosotros, ya que es el que nos ayudará a detectar futuras familias de malware distintas a las que ya tengamos dentro de nuestro dataset.

También añadir, que nuestro dataset es de algo menos de 33000 muestras, con un dataset mucho más grande, tendríamos unos sistemas de clusters bastante más funcionales.

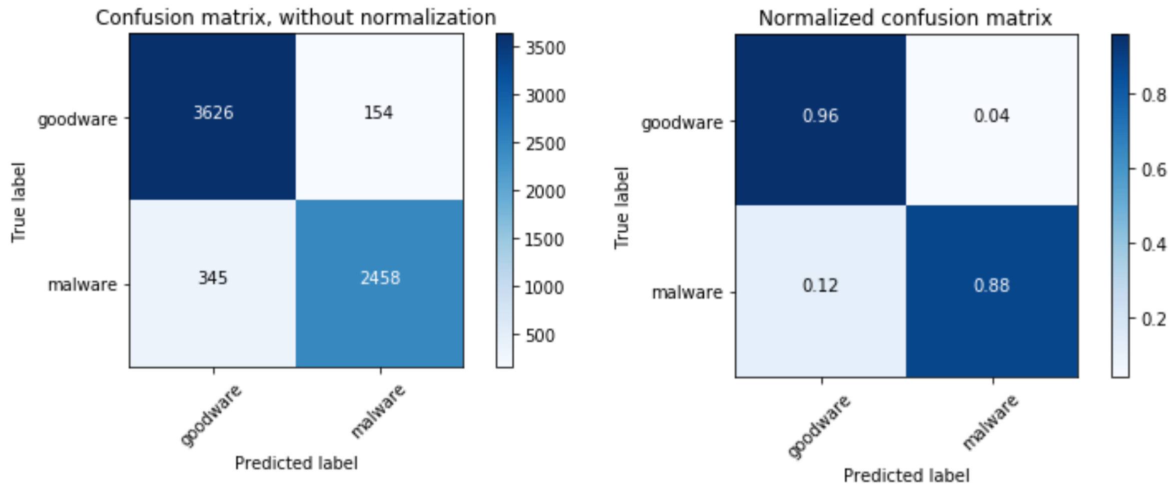
Por otro lado, la red neuronal funciona bastante bien, y, como hemos visto, clasifica bastante bien muestras que no ha visto con anterioridad. Aunque esto es así debido a que las muestras corresponden a muestras subidas durante un determinado periodo de tiempo, por lo tanto es bastante probable que sean muestras de familias similares, en el caso del malware.

Como dijimos anteriormente, es preferible que nuestro sistema clasifique erróneamente una muestra como goodware a que clasifique un malware como goodware, por eso bajamos el límite de 0.5 a 0.3, perdiendo precisión exactitud, y ganando sensibilidad. Por lo tanto en la unión del sistema de clusters con la red neuronal, es normal que sigamos este modelo más "agresivo", y tras probar varias alternativas para acoplar ambos sistemas, optamos porque la que más se adapta a nuestras necesidades es la siguiente:

- La muestra es analizada por nuestra red neuronal.
  - Si resultado > 0.3 (limite), es automáticamente clasificada como malware.
  - Si resultado < 0.3 (limite), es enviada al sistema de cluster.
    - Si los clusters la consiguen clasificar (goodware o malware), se clasifica con este resultado.
    - Si los clusters no la logran clasificar (outlier), se toma como válida la valoración de la red neuronal, que es goodware.

A continuación, en las figuras 23 y 24 mostramos las matrices de confusión de la red neuronal y el sistema completo entrenado y testeado con el mismo conjunto de datos, para que se pueda realizar una comparación real de ambos sistemas:

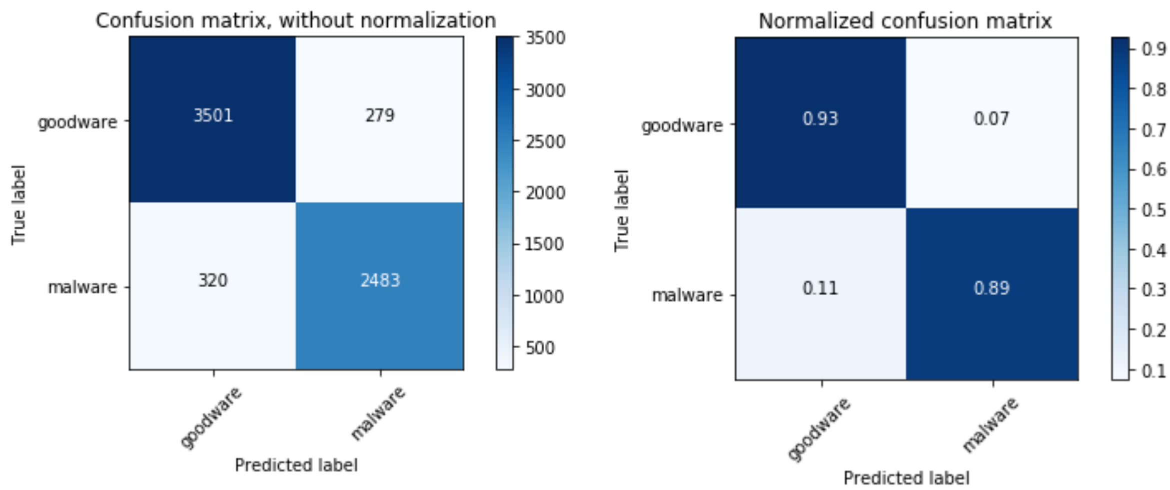
Figura 23. Matriz de confusión para la red neuronal, estableciendo el límite en 0.3



Fuente: Elaboración propia

$$ACC = 92.42\%, PPV = 91.31\%, TPR = 95.9\%$$

Figura 24. Matriz de confusión para el sistema completo (Clusters + ANN)



Fuente: Elaboración propia

$$ACC = 90.9\%, PPV = 91.62\%, TPR = 91.62\%$$

Al ver los resultados, podemos apreciar que empeoró con respecto al resultado obtenido con la red neuronal para el mismo límite. Pero al igual que ocurrió con la

red neuronal, para los distintos límites impuestos en las anteriores pruebas, la sensibilidad ha mejorado, sutilmente.

También, al conocer cómo están integrados ambos sistemas, se entiende que el sistema de clusters está adaptado de forma que lo que busca es disminuir el número de falsos positivos (FP), aunque con ello se empeore el número de verdaderos positivos (TP).

Además, con este sistema se espera que si llega una muestra de malware de una familia desconocida, es más probable que la red no llegue a clasificarlo como malware, y que el sistema de clusters sí.

4. Diseño del detector de malware

4.1. Servidor Web

En la parte del servidor, necesitábamos un sistema que nos proporcionara, por un lado una API RESTFul, que usarán los dispositivos para comunicarse, y por otro lado necesitábamos que fuese capaz de analizar todas las muestras que le lleguen, de forma asíncrona.

Para desarrollar el servidor, nos apoyamos en el framework Django, el cual nos agiliza el trabajo abstrayéndose de problemáticas como la interacción con la base de datos.

En nuestra interfaz definimos las siguientes llamadas:

- **app-result** (GET, POST): Si se ejecuta mediante GET, se encargará de devolver la información sobre la aplicación solicitada (se envía el sha256 para identificar la muestra). En cambio, si se ejecuta con POST, almacenará el resultado del análisis de una determinada aplicación, y notificará al usuario mediante push, si procede, el resultado del análisis.

- **upload-app** (POST): La llamada recibe una aplicación, la almacena y programa la tarea de Celery para realizar su análisis. Adicionalmente, el

dispositivo envía el “fcm_id” (identificador de Firebase Cloud Messaging) necesario para notificar mediante push al dispositivo cuando se tenga el resultado. Así mismo, se incluye el parámetro “notify”, el cual hace de interruptor para notificar o no al usuario.

El sistema lo completa Celery, un gestor de tareas. Celery es el encargado de realizar las tareas que se han ido encolando. En nuestro sistema, la única tarea que tenemos programada es la encargada de analizar una muestra. Primero extrae el reporte en bruto, el cual guarda en disco, después procesa este informe y obtiene las características de la muestra. Estas características son enviadas a nuestro sistema inteligente, el cual nos notificará, si considera, si la muestra es malware o no.

Una vez se tiene esta información, Celery se comunica con el servidor Django mediante la petición POST “app_result” notificando el resultado del análisis, así como aportándole información como el “fcm_id” de quien la envió y si se debe de notificar al usuario.

4.2. Ampliación cliente Android

Para el desarrollo de la aplicación móvil se ha decidido usar las últimas tendencias (por tanto la version minima requerida es Android 7), como trabajar con Android Jetpack, que es una colección de componentes software que facilita el desarrollo de aplicaciones Android. Estos componentes ayudan a seguir las mejores prácticas y liberan al desarrollador de escribir código repetitivo y simplifican tareas complejas para poder enfocarse en el código realmente interesante.

Jetpack está dividido en componentes que puedes usar por independiente o combinar entre ellos.

La aplicación irá desarrollada en Kotlin, lenguaje desarrollado por JetBrains, este está basado en la máquina virtual de java (JVM).

¿Porque elegir Kotlin en lugar de Java?

En el artículo Kotlin: el lenguaje de programación que ha desplazado a Java en Android, podemos ver algunas de las ventajas de ese lenguaje frente a Java:

- **Sencillez, seguro y pragmático:** Kotlin consigue mucho con poco código, lo cual lo hace más legible y nos ayuda a centrarnos en lo que estamos desarrollando, dejando atrás lo verboso que resultaba Java. También elimina los NullPointerException que han sido desde siempre uno de los grandes quebraderos de cabeza para los desarrolladores Java.

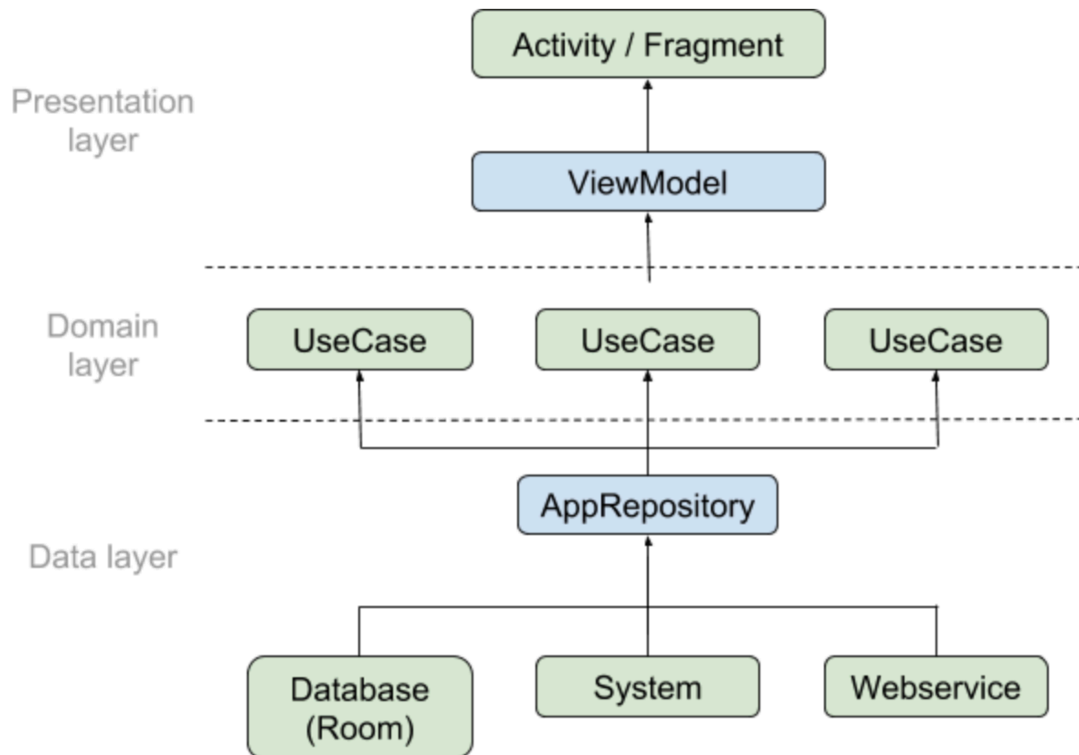
- **Interoperable con Java:** Ambos lenguajes pueden coexistir en un mismo proyecto, por lo que no hay que preocuparse porque las librerías o frameworks estén desarrolladas en Java.

- **Fácil migración:** Para un desarrollador Java, el pasar a Kotlin será muy fácil, y realmente gratificante conforme vaya aprendiendo más de su sintaxis. Por lo tanto, hace que una migración a este lenguaje sea mucho más rápida por parte de los desarrolladores.

- **Evolución asegurada:** Google presentó Kotlin como lenguaje oficial de Android en el Google IO 2017, esto sumado a que los desarrolladores de Kotlin son los mismos que desarrollan el IDE Android Studio (el oficial para desarrollar Android), hace ver que este lenguaje tiene bastante futuro.

Para el desarrollo de la aplicación usaremos el patrón MVVM (en inglés model–view–viewmodel) y la arquitectura de la misma es la siguiente:

Figura 25. Estructura de la aplicación Android



Fuente: Elaboración propia

Esta división nos permite una mayor modularidad de las diferentes partes del proyecto, permitiendo así modificar, añadir o eliminar cualquier parte sin que afecte a las demás.

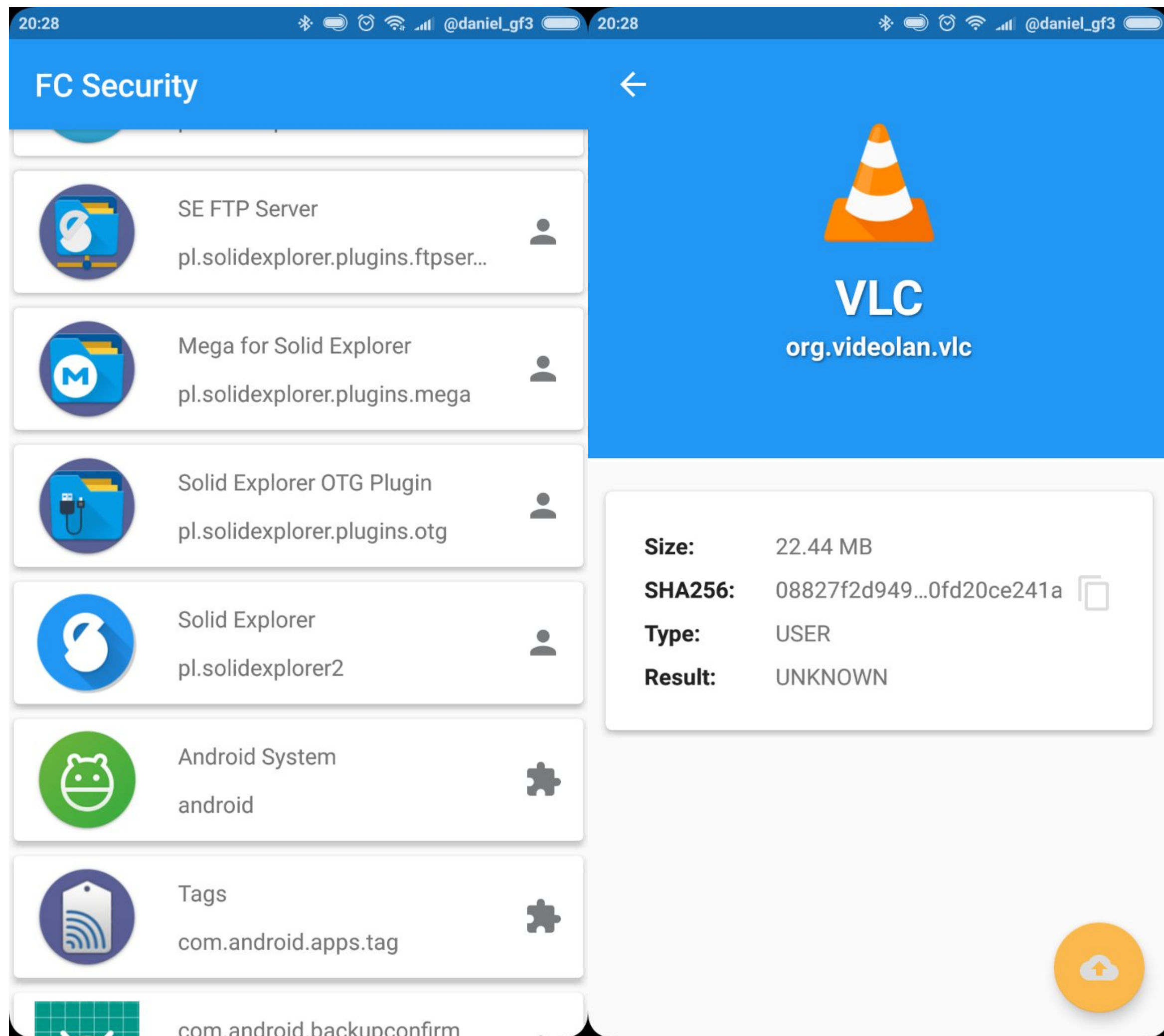
Al usar como base de datos Room y los ViewModels, podemos tener una interfaz siempre actualizada. Cualquier cambio que se registre en la base de datos será automáticamente representado en la interfaz gracias a las propiedades reactivas de este modelo y de los componentes usados.

Para la interfaz, hemos optado por una sencilla compuesta de dos ventanas, en la primera se mostrará el listado de aplicaciones instaladas y algunos datos de estas como el icono, nombre, paquete y si es de usuario (instalada manualmente por el usuario) o sistema (aplicación que viene con el SO).

En la segunda ventana se mostrará información más detallada de la aplicación, junto con el resultado del análisis, si dispusiéramos de él (guardado previamente en nuestra base de datos). En caso de no disponer del resultado, estará visible un

botón flotante en la parte inferior derecha que nos dejará enviar la aplicación para su análisis.

Figura 26. Capturas de pantalla de la aplicación Android



Fuente: Elaboración propia

El envío se realiza en background, usando la clase FirebaseJobServices, la cual ejecutará este cuando se cumplan unas condiciones previamente impuestas, y más convenga al sistema. Si el trabajo no se puede realizar, este se pospone para volver a intentarlo tiempo después. Con esto conseguimos que la aplicación sea muy liviana para el sistema.

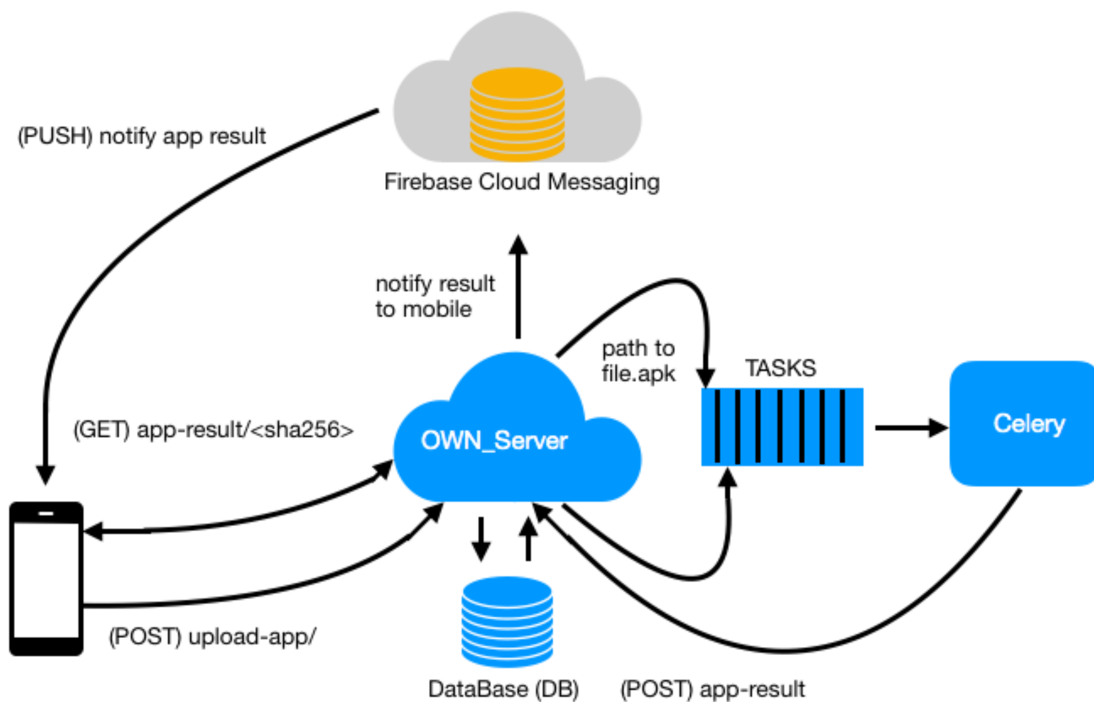
La aplicación está diseñada para ser capaz de obtener toda la información posible de las aplicaciones instaladas haciendo uso del framework de Android.

Para mantenerse actualizada dispone de un BroadcastReceiver que se activará si detecta cualquier cambio en alguna de las aplicaciones (actualizado,

borrado o instalación). Además, para asegurar la consistencia de los datos reales con la base de datos, se procederá a un refresco periódico de la misma.

4.3. Coordinación de los módulos.

Figura 27. Flujo de datos del sistema de detección de malware completo.



Fuente: Elaboración propia

Nuestro sistema está compuesto de dos partes bien diferenciadas, que son los dispositivos móviles y el servidor.

En primer lugar, el dispositivo móvil, como se comentó, escanea todas las aplicaciones instaladas, y comprueba contra el servidor (GET app-result/<sha256>) cuales de ellas se tiene resultado. Si el servidor no tiene, se devolverá un error 404 (NOT FOUND), en cambio, si tiene el resultado, lo devolverá y se almacenará en una base de datos interna del dispositivo móvil.

Cuando una app no está analizada, el usuario tiene la posibilidad de enviarla para el análisis (POST upload-app/) desde el móvil. Si lo hace, además del fichero apk, también enviará el identificador FCM (identifica al dispositivo en el servidor de Firebase), y si se requiere notificación una vez se tenga resultado.

Una vez el fichero llega al servidor, este genera una tarea y la encola. Celery se encarga de ir realizando las tareas encoladas.

Esta tarea se basa en obtener el reporte en bruto y guardarlo en disco, tras lo cual se obtiene un reporte más específico y de este las características necesarias para nuestro sistema inteligente. Ya con las características, se procede a analizar de la forma que se comentó en el punto correspondiente.

Cuando tenemos el resultado, la tarea, como último paso, realiza una petición POST (app-result) al servidor. El servidor al recibir el resultado, lo almacena en la base de datos, para futuras consultas, y realiza, si el parámetro notify es afirmativo, una petición POST a los servidores de Firebase, para que este envíe una notificación PUSH al dispositivo móvil, notificando así el resultado del análisis.

5. Conclusiones y posibles mejoras futuras

Durante el desarrollo del proyecto nos hemos encontrado muchos problemas referentes al tiempo de procesado:

- Descarga de un número considerable de muestras.
- Análisis inicial de las muestras para extracción de características.
- Entrenamiento de la red neuronal.
- Entrenamiento de los clusters.

Esto ha provocado que se invierta mucho tiempo en estas partes, que igual con un equipo más potente no habría sido tanto.

Por culpa de esto mismo, nos tuvimos que contentar con un dataset de cerca de 33000 muestras.

La red neuronal ha aprendido bastante bien, pero estimamos que con un número considerablemente superior de muestras, esta habría dado resultados mucho más precisos.

En el caso de los clusters, esto habría sido mucho más decisivo, puesto que, como ya se comentó, es mucho más complejo hacer clusters de goodware debido a las grandes diferencias que hay entre ellos, no como en el caso del malware. Por tanto, con un dataset mucho más grande, es muy probable que el sistema hubiera encontrado un número mayor de clusters y por tanto, clasificará mejor las muestras.

La extracción de características, fue otro de los puntos críticos con respecto al tiempo, más concretamente la parte de la que se encarga AndroGuard, el cual requiere bastante tiempo a la hora de realizar el decompilado de las aplicaciones. Este proceso es casi inmediato en aplicaciones de unos pocos MB, pero, en el caso de algunas aplicaciones, que pueden llegar a ocupar los 100 MB, se nota mucho. Por tanto, uno de los puntos a mejorar sería realizar un estudio de AndroGuard en profundidad, buscando opciones que agilicen el proceso, o en caso de que no se pueda, intentar reprogramar alguno de sus módulos para optimizarlo para nuestros intereses.

El tiempo de entrenamiento de la red neuronal, una vez se tienen los hiper-parámetros, es aceptable, pero si se podría acortar enormemente si se utiliza un equipo con GPU ya que TensorFlow funciona mucho más rápido con GPU que CPU. De igual modo ocurre si se tuvieran que buscar nuevamente los hiper-parámetros.

Otra de las posibles mejoras a futuro, sería enviar más información desde el dispositivo, y no solo enviar el fichero. Desde el dispositivo se puede extraer, entre otras cosas, el instalador de una aplicación en concreto. Por tanto, si sabemos que una app procede de un instalador conocido y con cierta credibilidad, como el caso de Google Play, esta aplicación se podría analizar con una cierta confianza añadida.

El problema que surge de aquí, es que para poder hacer esto, necesitamos un dataset con esta información, o hacer una suposición muy grande, como por ejemplo, suponer que todas las aplicaciones marcadas como goodware por VirusTotal, son descargadas de Google Play.

De otra forma, no tendríamos esa información a la hora de entrenar la red y por tanto luego no sabría cómo usarla.

Siguiendo en el dispositivo, una mejora muy interesante sería estudiar en profundidad formas de extraer información en el propio dispositivo. Ahora mismo, tras estudios realizados, conseguimos sacar bastante información, pero no se conseguía acercar a AndroGuard. Pero si vemos que con las que obtenemos del dispositivo se puede clasificar satisfactoriamente la muestra, podríamos embeber una red neuronal, entrenada en servidor y actualizada cada cierto tiempo, y que fuera el propio dispositivo el que de forma offline escanee sus aplicaciones. De esta forma, en un uso real, se reduciría enormemente la carga del servidor, que, casi únicamente se encargaría de recibir aplicaciones y cuando tuviera un número nuevo considerable, realizar un entrenamiento y notificar a los dispositivos que la nueva red neuronal está disponible.

6. Referencias bibliográficas

Estadísticas oficiales de Google (2018). Consultado en https://developer.android.com/about/dashboards/?hl=es-419 el 15/09/2018

Hamilton, H. (2018). *Confusion Matrix*. Computer Science 831. Consultado en http://www2.cs.uregina.ca/~dbd/cs831/notes/confusion_matrix/confusion_matrix.html

Ina F. (2010). Steve Jobs: Let the post-PC era begin (live blog). Consultado en https://www.cnet.com/news/steve-jobs-let-the-post-pc-era-begin-live-blog/ el 10/08/2018

MLEXTEND, *Confusion Matrix*. Consultado en https://rasbt.github.io/mlxtend/user_guide/evaluate/confusion_matrix/ el 13/09/2018

Security for Android team (2016*). Android Security 2016 Year In Review*. Consultado en https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf el 13/08/2018

Security for Android team (2017*). Android Security 2017 Year In Review*. Consultado en https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf el 13/08/2018

Sophos (2017). *SophosLabs 2018 Malware Forecast*.
Consultado en https://www.sophos.com/en-us/en-us/medialibrary/PDFs/technical-papers/malware-forecast-2018.pdf?la=en el 02/09/2018

StatCounter. Consultado en http://gs.statcounter.com/os-market-share#monthly-201209-201809 el 10/09/2018

StatCounter. Consultado en http://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201209-201809 el 10/09/2018

Symantec (2016). *Mazar BOT malware invades and erases Android devices*. Consultado en https://us.norton.com/internetsecurity-emerging-threats-mazar-bot-malware-invades-and-erases-android-devices.html el 05/09/2018

Unuchek, R. (2018). *Virología móvil 2017*. Boletín de seguridad de Kaspersky. Consultado en https://securelist.lat/mobile-malware-review-2017/86322/ el 02/09/2018

Vegas, A. (2017). *Kotlin: el lenguaje de programación que ha desplazado a Java en Android.* Consultado en https://www.beeva.com/beeva-view/desarrollo/kotlin-el-lenguaje-de-programacion-que-ha-desplazado-a-java-en-android/ el 20/08/2018

VirusTotal (2018). *Reporte de la muestra Mazar BOT*. Consultado en https://www.virustotal.com/es/file/73c9bf90cb8573db9139d028fa4872e93a528284c02616457749d40878af8cf8/analysis/ el 05/09/2018

Weka. Consultado en https://www.cs.waikato.ac.nz/ml/weka/ el 10/09/2018

Wikipedia (s.f.). *Silhouette (Clustering)*. Consultado en https://en.wikipedia.org/wiki/Silhouette_(clustering) el 12/08/2018