Escuela Politécnica Superior

Dpto. de Ingeniería Informática

Doctorado en Ingeniería Informática y Telecomunicación

Doctoral Thesis

Machine learning techniques for Android malware detection and classification

Author

ALEJANDRO MARTÍN GARCÍA

Supervisors

Dr. D. DAVID CAMACHO FERNÁNDEZ
Dr. D. RAÚL LARA CABRERA

March 2019

Department: Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid (UAM)
SPAIN

Thesis title: "Machine learning techniques for
Android malware detection and classification"

Author: **Alejandro Martín García**
Máster en Ciencia y Tecnología Informática
Universidad Carlos III de Madrid, Spain

Supervisors: **David Camacho Fernández**
Doctor Ingeniero en Informática
Universidad Autónoma de Madrid, SPAIN

Raúl Lara Cabrera
Doctor Ingeniero en Informática
Universidad Politécnica de Madrid, SPAIN

Year: 2019

Committee: President:

Secretary:

Vocal 1:

Vocal 2:

Vocal 3:

To

all that

have supported me

and that have encouraged me all the way.

A

todos aquellos

que me han apoyado

y me han animado hasta el final.

Agradecimientos

Me gustaría empezar estas líneas agradeciendo el haber podido estudiar y trabajar en temas que me apasionan, lo que me ha motivado e inspirado día tras día. A pesar de muchas horas de sueño perdidas, estos años han sido increíbles. Experiencias, viajes, nuevos conocimientos, nuevos amigos y muchas anécdotas que se quedarán conmigo para siempre.

Debo dirigirme en primer lugar a mis directores. A David por la oportunidad que me ofreció para comenzar este trabajo, por sus palabras y consejos, así como por su ayuda para acabar este camino con la mejor preparación posible. A Raúl por sus sabios consejos, por nuestras charlas y cafés y por haberme ayudado a llegar hasta aquí. Gracias de verdad. A Julio debo agradecer haberme recibido en Canterbury. Fue una estancia en la que aprendí y disfruté. Gracias.

A todo el grupo AIDA, Ángel, Antonio, Cristian, Gema, Javi, Raquel y Víctor, por su compañía, ayuda, ideas, experiencias y viajes. Gracias a todos por haber formado parte de este camino.

Debo dedicar también unas palabras a Raúl y Alfonso, dos grandes amigos de verdad. Gracias por vuestro apoyo, por vuestra preocupación y por los buenos ratos que pasamos.

A María me gustaría agradecer sus palabras de ánimo, el saber hacerme sonreír cuando más lo he necesitado, su cariño, su inestimable ayuda y por acompañarme de la mano en este camino. Te quiero mucho.

A mis padres debo agradecerles los valores de educación, dedicación y esfuerzo que me han inculcado, por haberme ayudado a llegar hasta aquí y por apoyarme hasta el final. A mi hermana Vanesa, por preocuparse como lo hace por mí y por estar siempre ahí. Gracias de todo corazón. Y a mis abuelos, por cuidarme desde pequeño, por sus consejos y por haber estado siempre que los he necesitado.

Finalmente, la realización de esta tesis no habría sido posible sin la financiación aportada por el proyecto CIBERDINE: Cybersecurity, Data and Risks (S2013/ICE3095) concedido por la Comunidad de Madrid.

Resumen, conclusiones y trabajo futuro

Resumen

Android representa uno de los sistemas operativos que más sufre la creación de aplicaciones maliciosas. Cada día, miles de nuevas muestras de *malware* tratan de sortear las medidas de seguridad desplegadas por las diversas tiendas de aplicaciones para la plataforma Android, con el objetivo principal de infectar nuevos dispositivos. Para atacar este problema, es necesario investigar y desarrollar mecanismos capaces de filtrar automáticamente grandes conjuntos de muestras sospechosas, detectando aquellas que contienen una carga maliciosa.

Esta tesis estudia y aborda la aplicación de técnicas de aprendizaje automático para el desarrollo de métodos de detección de *malware* para Android desde diferentes perspectivas. Además, también se aborda la clasificación de *malware* en familias. Por otro lado, se ha realizado un profundo análisis de la familia de *malware* Jisut que ha permitido revelar algunas de las prácticas empleadas más importantes por sus desarrolladores y que deben ser consideradas al afrontar esta tarea.

En primer lugar, se utilizan técnicas de aprendizaje automático para construir métodos de detección de *malware* para Android destinados a determinar con gran precisión si una aplicación es *malware* o *benignware*. Con este objetivo, el comportamiento de cada aplicación es descrito mediante grupos de características estáticas y dinámicas, las cuales son modeladas mediante una representación basada en cadenas de Markov. Después se aplican conjuntos de clasificadores, mostrando que las características estáticas permiten obtener mejores resultados que las dinámicas. También se describe un enfoque de fusión de ambos tipos de características que obtiene mejores resultados en comparación con el uso de un único grupo de características.

En segundo lugar, se afronta el problema de la clasificación de aplicaciones maliciosas en familias de *malware*. Este problema compone una tarea esencial que trata de minimizar los daños causados por el *malware* al mismo tiempo que busca identificar de forma apropiada los diferentes grupos de *malware* existentes. Para este proceso se han empleado arquitecturas de aprendizaje profundo, algoritmos clásicos de aprendizaje automático y diferentes técnicas para manejar datos desequilibrados. Los resultados muestran que estas técnicas permiten desarrollar métodos precisos de clasificación por familias. La resistencia de estos métodos frente a ataques de adversarios también ha sido analizada. Para ello, se ha implementado un ataque dirigido hacia un clasificador propuesto en la literatura, demostrando que es posible forzar a un clasificador a asignar muestras de forma incorrecta a nuevas familias aleatorias de *malware*, o incluso modificar una muestra para hacerla pasar por una nueva familia de malware concreta.

Finalmente, se presenta una herramienta de código abierto y que ha sido denominada AndroPyTool. Esta integra diversas herramientas de análisis de *malware* con el objetivo principal de ofrecer a la comunidad científica una herramienta integrada para la extracción de un amplio conjunto de características estáticas y dinámicas de aplicaciones Android. Mediante el uso de esta herramienta ha sido posible construir y

ofrecer públicamente un amplio conjunto de datos llamado OmniDroid, que contiene características estáticas y dinámicas extraídas de aplicaciones benignas y maliciosas.

Conclusiones y Trabajos Futuros

En este capítulo, se exponen las diferentes conclusiones obtenidas en el trascurso de la investigación realizada relativa a la presente Tesis Doctoral. Al mismo tiempo, se responde a las diferentes Preguntas de Investigación planteadas. El objetivo principal es ofrecer una serie de detalles y comentarios que puedan ser útiles a futuros investigadores interesados en los problemas de detección y clasificación de *malware* diseñado para el sistema operativo Android. Finalmente, también se identifican distintas líneas de trabajo futuro para extender este trabajo.

Conclusiones

A lo largo de los diferentes capítulos de esta Tesis, los problemas relativos a la detección y clasificación de *malware* en Android han sido estudiados desde diferentes perspectivas, todas ellas dentro del marco del uso de técnicas de aprendizaje automático.

En el Capítulo 2, se ha presentado una familia concreta de *malware* denominada Jisut. Se trata de un ransomware, un programa que persigue un beneficio económico mediante el bloqueo del terminal o la encriptación de los datos que contiene. El estudio de las diferentes variantes de esta familia ha permitido observar importantes patrones de comportamiento. Por ejemplo, se ha podido ver cómo diferentes variantes son creadas a lo largo del tiempo, aplicando cambios a muestras de variantes anteriores. Este importante detalle recalca que la detección temprana de las primeras aplicaciones que dan lugar a una nueva familia puede ayudar a detectar nuevas muestras de la misma en el futuro.

El análisis también ha permitido detectar el empleo de técnicas que tratan de dificultar el análisis en las muestras más modernas. Por ejemplo, algunas variantes ocultan la carga maliciosa en archivos que son desencriptados en tiempo de ejecución, un hecho que disminuye de forma considerable la efectividad de las técnicas de análisis estático, no siempre capaces de detectar este tipo de contenido.

Después del análisis de la familia Jisut, se proponen diferentes mecanismos para detectar y clasificar *malware* en el Capítulo 3. Comenzando con el problema de detección de *malware*, se han probado diferentes combinaciones de características y algoritmos de clasificación. Las características estáticas han resultado ser un poderoso instrumento para representar el comportamiento de la aplicación. Esta representación se ha utilizado para entrenar distintos conjuntos de clasificadores o *ensemble classifiers* que han arrojado altos valores de precisión.

Una vez estudiado el uso de características estáticas, también se han analizado aquellas extraídas dinámicamente. En este caso, se ha seguido una representación basada en cadenas de Markov para transformar los resultados obtenidos con DroidBox en información estructura que permita entrenar un algoritmo de aprendizaje automático. Primero se genera una secuencia de eventos por cada aplicación. Después, una matriz de probabilidades de transición entre estados (constituidos por los eventos originales) es construida en consonancia con las transiciones observadas en la secuencia.

La importancia de cada evento o estado también se tiene en cuenta mediante el cálculo de su frecuencia sobre la secuencia completa. Finalmente, la matriz de probabilidades

de transición se transforma en un vector al que se concatena la frecuencia de cada estado para crear un vector de características por aplicación. Esta representación también ha sido probada usando distintos conjuntos de clasificadores, mostrando valores de precisión inferiores en comparación con los obtenidos con características estáticas.

El mismo capítulo también describe un novedoso enfoque de fusión de características estáticas y dinámicas que tiene como objetivo principal mejorar los resultados obtenidos por estos dos grupos de características individualmente. Mediante el uso de un clasificador por votación, se combina el estimador que mostró mejores valores en la clasificación con características estáticas con el que mostró el mejor dato con características dinámicas. Los resultados de este nuevo enfoque de fusión muestran una leve mejora respecto a los conseguidos previamente.

Por otro lado, la clasificación de *malware* en Android por familias también ha sido tratada. Para ello, se ha utilizado un conjunto de aplicaciones de distintas familias de malware, de las que se han extraído características dinámicas. Para su representación, se ha seguido el mismo modelo previamente utilizado basado en cadenas de Markov. Los resultados, obtenidos tras distintos experimentos en los que se utilizan técnicas de aprendizaje profundo, algoritmos clásicos de aprendizaje automático y también métodos destinados a tratar el problema de datos desequilibrados, demuestran que todas estas técnicas permiten crear herramientas precisas de clasificación de *malware* por familias.

El Capítulo 4 se ha centrado por su parte en analizar el uso de técnicas de aprendizaje automático desde una perspectiva de protección frente a ataques que tratan de obstruir su correcto funcionamiento. Para ello, se ha diseñado e implementado un ataque contra un clasificador propuesto en la literatura, denominado RevealDroid, demostrando que efectivamente los métodos basados en aprendizaje automático son vulnerables. El ataque consiste principalmente en modificar el vector de características de una muestra concreta con cambios incrementales que no afectan a su semántica. También se analiza una contramedida basada en el uso de conjuntos de clasificadores, los cuales permiten distribuir la toma de decisión sobre la categoría de la muestra entre diferentes estimadores y, por tanto, son capaces de contrarrestar un ataque contra un clasificador concreto.

Finalmente, en el capítulo 5 se ha descrito el *framework* AndroPyTool, que ha sido desarrollado durante el transcurso de este trabajo. Esta herramienta permite obtener automáticamente un gran número de características estáticas y dinámicas extraídas de aplicaciones Android. AndroPyTool puede ser utilizado para generar *datasets* de vectores que representan características estáticas y dinámicas de muestras benignas y malignas. En este trabajo, la herramienta ha permitido construir un *dataset* denominado OmniDroid, que contiene un conjunto equilibrado de muestras de ambas clases representadas por un amplio número de características ya extraídas.

Respuesta a las preguntas de investigación

Este apartado se da respuesta a las distintas preguntas de investigación planteadas:

- **RQ1:** *¿Cuáles son las prácticas más importantes usadas en el malware diseñado para Android y que deben ser consideradas al diseñar herramientas de detección?*

El estudio realizado sobre un gran número de muestras de la familia Jisut de *ransomware* para Android [MHCC18], y que ha sido descrito en la Sección 2.3, ha permitido conocer diversos detalles relevantes sobre su implementación. También se han analizado los patrones estructurales compartidos entre las distintas muestras. Toda esta información permite entender mejor esta familia y también conocer los mecanismos más comunes empleados por el *malware* que afecta a esta plataforma.

Mediante un análisis de las similitudes que existen entre las diversas muestras, se ha podido demostrar cómo esta familia de ransomware ha originado diferentes variantes que implementan pequeños cambios. A un mayor nivel de abstracción, se puede observar una evolución temporal en la que ciertas variantes son tomadas como punto de partida para desarrollar nuevos grupos de aplicaciones maliciosas, haciendo uso de cambios incrementales. Esto es un hecho interesante, ya que implica que es posible encontrar fuertes relaciones entre muestras de la misma familia aun perteneciendo a diferentes variantes. Al mismo tiempo, esta tendencia resalta la importancia de la clasificación de *malware* por familias, una tarea esencial para detectar nuevas variantes, pero también nuevas familias desconocidas hasta el momento. Este conocimiento puede ser más tarde utilizado para construir herramientas de detección que cubran un mayor número de muestras.

El análisis realizado también ha remarcado la importancia de la presencia de operaciones criptográficas en familias de *ransomware*. Por ejemplo, algunas de las muestras de esta familia encriptan todos los archivos del usuario, forzándolo a pagar el rescate. Esta operación origina un inusual incremento en el número de llamadas realizadas a la API relacionadas con funciones criptográficas. En este sentido, un control sobre las llamadas a la API realizadas debe formar parte de los mecanismos de detección de malware, ya que pueden relevar patrones específicos de muestras de *ransomware*.

Otra práctica interesante es el uso de nuevas técnicas de ocultamiento observadas en las variantes más nuevas. Estas técnicas se basan en esconder la carga maliciosa en archivos o librerías separadas, en algunos casos encriptadas, con el objetivo de dificultar su análisis. Debido a esto, un enfoque dinámico resulta más conveniente para afrontar este problema. El uso de un entorno restringido de pruebas donde la aplicación sospechosa es ejecutada, y todas las interacciones realizadas por la misma monitorizadas, permite capturar detalles de bajo nivel que no podrían ser extraídos en caso de utilizar un análisis estático.

- **RQ2:** *¿Es posible encontrar grandes conjuntos de características ya extraídas de aplicaciones Android benignas y malignas?*

  Si bien existe una gran cantidad de literatura centrada en el problema de la detección de *malware* en Android, al mismo tiempo existe una falta de conjuntos de datos conteniendo características ya extraídas de muestras. Estos datos resultan imprescindibles para entrenar y comparar los algoritmos de aprendizaje automático utilizados en la construcción de herramientas de detección y clasificación. En la mayoría de los casos, los autores utilizan conjuntos de ejecutables de los cuales ellos mismos extraen el conjunto deseado de características. Este

proceso se hace más complicado cuando son múltiples las características a extraer y son varias las herramientas necesarias. Los conjuntos de datos existentes que ofrecen información del comportamiento son limitados. Como se ha discutido en la Sección 3.1, ofrecen un número reducido de características, extraídas únicamente de muestras maliciosas [RF16, RFB17].

Con el objetivo de facilitar el proceso de desarrollo y prueba de métodos de detección y clasificación de malware, se decidió construir un nuevo *dataset* que cubriera un gran número de características extraídas estáticamente y dinámicamente. Para ello, se desarrolló el *framework* AndroPyTool, que integra diferentes herramientas de análisis de *malware*. Esta herramienta permite extraer de forma eficiente y automática las características más utilizadas en la literatura, evitando tener que utilizar distintas herramientas específicas. AndroPyTool se encuentra disponible como un proyecto de código abierto [MCC18].

Finalmente, con el fin de construir un dataset mediante la herramienta AndroPyTool, se recolectó un gran número de muestras benignas y maliciosas desde el repositorio Koodous[1] y desde AndroZoo[2]. Todas ellas fueron analizadas con AndroPyTool, lo que permitió generar el *dataset* OmniDroid, el cual incluye información preestática, estática y dinámica de 11.000 aplicaciones benignas y 11.000 aplicaciones maliciosas. Estos datos se encuentran disponibles públicamente y pueden ser descargados desde *AIDA Datasets Repository*[3]. Con este *dataset* se pretende ofrecer a la comunidad un conjunto de datos de referencia para construir o probar herramientas de detección de *malware*.

- **RQ3:** *¿Se pueden utilizar métodos de aprendizaje automático con características estáticas para detectar malware en Android de forma precisa?*

  La Sección 3.1 del presente documento describe nuevos métodos para detectar y clasificar *malware* en Android. Estos se basan en una representación de características estáticas con las que se han entrenado varios conjuntos de clasificadores principalmente integrados por árboles de decisión. Los resultados muestran que esta combinación permite crear herramientas de detección y clasificación precisas. Debido al uso de características numéricas y binarias, este tipo de clasificador resulta el más adecuado para esta tarea.

  De forma más específica, un listado de llamadas al sistema, permisos declarados, *opcodes*, *intent-filters*, servicios o comandos del sistema han sido utilizados en los experimentos. De estos se pueden extraer diferentes conclusiones. Por ejemplo, el uso de diferentes combinaciones de características estáticas ha demostrado que las llamadas a la API son la mejor forma de representar el comportamiento de cada aplicación, ayudando a distinguir de forma precisa entre *malware* y *benignware*. Incluso cuando se combinan con otras características como flujos de información, los resultados no mejoran los del uso de llamadas al sistema de forma individual.

  En general, las técnicas de análisis estático han resultado ser eficientes y rigurosas para entrenar clasificadores basados en técnicas de aprendizaje automático. En comparación con información dinámica, este enfoque no requiere de la ejecu-

[1] `https://koodous.com`
[2] `https://androzoo.uni.lu`
[3] `https://aida.ii.uam.es/datasets/`

ción de cada muestra durante un período de tiempo concreto, sino simplemente descomprimir la muestra y extraer datos de sus diferentes archivos y recursos. Esto permite detectar muestras maliciosas con cerca del 90% de precisión mediante un clasificador de tipo Random Forest.

- **RQ4:** *¿Es posible aplicar modelos de aprendizaje automático sobre trazas dinámicas para detectar malware en Android de forma precisa?*

  En este trabajo se ha comprobado que las características estáticas permiten generar una rápida y eficiente descripción del comportamiento de cada aplicación. Sin embargo, estas pueden fallar al intentar detectar secciones de código malicioso. Por ejemplo, las técnicas de ofuscación modernas que encapsulan la carga maliciosa en archivos ocultos pueden provocar que una descripción estática resulte ineficaz. De este modo, puede resultar necesario ejecutar la muestra para que esta acceda a los archivos que contienen el *malware* y este sea ejecutado, pudiendo así capturar la carga dinámica de las secciones de código malicioso. En base a esto, el uso de características dinámicas ha sido estudiado en la Sección 3.2.

  Al contrario de lo que se podía esperar, los experimentos realizados con métodos basados en conjuntos de clasificadores muestran una reducción en el número de muestras correctamente asignadas a su categoría en comparación a los que se basan en características estáticas. Son varias las posibles causas de este resultado. Por un lado, el conjunto de eventos que es controlado por DroidBox puede ser insuficiente para detectar pequeños patrones de comportamiento. Por otro lado, un enfoque dinámico podría no cubrir por completo las operaciones realizadas por la muestra maliciosa (por ejemplo, si la carga maliciosa es activada únicamente cuando el usuario accede a una sección en particular de la aplicación).

  Por esta razón, el funcionamiento original de MonkeyRunner (el servicio que controla la ejecución de la aplicación en el emulador) fue modificado con el objetivo de enviar un mayor número de interacciones a la pantalla y los botones. Los resultados apuntan también a la necesidad de combinan características estáticas y dinámicas, lo que puede ser crucial para determinar con precisión la naturaleza de la muestra. Por ello, es necesario estudiar si una combinación de ambos tipos de características permite construir mejores clasificadores.

- **RQ5:** *¿Es posible combinar características estáticas y dinámicas para construir mecanismos de detección de malware más efectivos?*

  En la Sección 3.3 se ha propuesto un modelo para la fusión de características estáticas y dinámicas. En este modelo se combinan, mediante un clasificador por votación, los algoritmos que mejores resultados dieron al clasificar cada tipo de características de forma individual. De este modo, ambos tipos de características aportan a la decisión final. Para medir el grado de contribución de cada uno de ellos a esta decisión, se utilizan dos pesos calculados mediante una búsqueda en *grid*. Los pesos resultantes fueron de 0,7 para el clasificador que recibe como entrada las características estáticas y de 0,3 para el que recibe las dinámicas. Como se puede ver en estos dos valores, la información estática toma especial relevancia en la clasificación final.

El enfoque de fusión propuesto mejora muy levemente los resultados obtenidos por los dos grupos de características de forma individual, desde una exactitud del 89,3% al 89,7%. Aunque la diferencia es pequeña, esta refleja que las características dinámicas pueden ayudar en algunos casos a una mejor detección de patrones maliciosos. Además, la combinación de ambos tipos de fuentes de información también es importante en términos de robustez frente a ataques. Como se vio en el Capítulo 4, los clasificadores por conjuntos en combinación con un amplio número de características permiten reducir el éxito de estos ataques, ya que el espacio exploratorio se vuelve más grande y complejo.

El uso de mejores herramientas de análisis dinámico también puede ayudar a mejorar los resultados en los enfoques de fusión de características estáticas y dinámicas. En este sentido, resulta necesario estudiar nuevos mecanismos de análisis dinámico, utilizando emuladores más avanzados e indetectables que permitan capturar un mayor rango de interacciones de la muestra con el sistema operativo. Esto ayudará a mejorar el rendimiento de los actuales métodos de clasificación por familias y de detección.

- **RQ6:** *¿Es posible atacar clasificadores basados en aprendizaje automático para producir errores de clasificación?*

  El ataque implementado y probado descrito en el Capítulo 4 ha demostrado que los clasificadores de *malware* para Android basados en aprendizaje automático pueden forzados para producir errores en la clasificación. Mediante el uso de una búsqueda heurística guiada por un algoritmo genético, el vector original de características de una muestra puede ser modificado añadiendo cambios incrementales. Esto puede llevar a asignar la muestra a una nueva familia aleatoria o a una previamente prefijada. Los cambios se pueden introducir en la muestra usando predicados opacos que nunca son ejecutados, evitando así modificar la semántica de la aplicación.

  El ataque realizado evidencia que los métodos basados en aprendizaje automático son vulnerables. Específicamente, los clasificadores que confían en un número reducido de características para entregar una etiqueta pueden ser más fácilmente atacados. Para contrarrestar este problema, se ha planteado e implementado una contramedida, basada en reemplazar el clasificador original por un conjunto de clasificadores. Cada uno de ellos uno se encarga de clasificar la muestra en base a un grupo de características independientes. Esto conlleva una complicación en la implementación de estos ataques, ya que la clasificación ahora depende de la decisión de varios estimadores.

Trabajo futuro

Aunque todo el trabajo realizado ha tratado de analizar de forma exhaustiva el problema de la detección y clasificación de *malware* en Android desde diferentes perspectivas, también se han observado puntos donde se podría continuar desarrollando esta investigación:

- El análisis de la familia Jisut ha manifestado la existencia de patrones de comportamiento que deben ser tomados en consideración. Se necesitan estudios similares de otras familias para mostrar detalles de implementación que pueden ayudar a diseñar y construir mejores herramientas de detección y clasificación. Además, estos análisis pueden ser útiles para mejorar métodos de clasificación existentes, ayudando a descubrir nuevas variantes de familias ya conocidas o agrupando diferentes muestras bajo la misma familia. A un nivel inferior, este tipo de estudios permite analizar los métodos de encriptación y desencriptación y los métodos utilizados para bloquear los dispositivos, entre otras particularidades.

- En general, aunque los resultados obtenidos han mostrado altas tasas de precisión y exactitud, es necesario estudiar el uso de nuevas características, técnicas de representación y preprocesado, y también de nuevos métodos de aprendizaje. Particularmente, se deben estudiar características más avanzadas, que cubran a un mayor número de archivos con el objetivo de obtener información de librerías compiladas y de la presencia de funciones para la carga dinámica de código. También se deben estudiar nuevas combinaciones de características con el objetivo de mejorar los resultados obtenidos, donde las llamadas al sistema han aportado los mejores resultados.

- Los experimentos en los que se ha evaluado el uso de características dinámicas evidencian también que se debe investigar más este tipo de información. Si bien estas características pueden ayudar indudablemente a mejorar los métodos existentes, se requieren emuladores que no puedan ser detectados por la muestra analizada. También hace falta investigación en relación a la estimulación de la muestra mediante interacciones más realistas, monitorizando un amplio conjunto de eventos y analizando nuevos procedimientos para combinar esta información con estática.

- Los clasificadores de *malware* son un potente instrumento para asignar aplicaciones a su correspondiente familia, manteniendo así un mejor registro de las diferentes familias existentes, mejorando la detección de *malware* desconocido o que se basan en ataques de día cero y detectando nuevas variantes de familias ya conocidas. Este triaje es una tarea esencial para lidiar con el riesgo que supone el *malware*, especialmente para mitigar los posibles daños causados si logra infectar el dispositivo. Al conocer la familia a la que pertenece un programa malicioso, será mucho más fácil aplicar los pasos más convenientes para evitar su propagación y para mitigar sus efectos. Por lo tanto, es necesario ampliar la investigación con el objetivo de mejorar los métodos actuales de clasificación.

- El ataque diseñado, implementado y probado, descrito en el Capítulo 4, ha

permitido demostrar que los métodos de clasificación basados en técnicas de aprendizaje automático pueden ser eludidos. Esto conlleva una serie de riesgos que deben ser tenidos en cuenta. Una ampliación de esta investigación podría centrarse en estudiar la fortaleza de estas herramientas y en proponer nuevas contramedidas para prevenir este tipo de ataques.

- Finalmente, la herramienta AndroPyTool y el *dataset* OmniDroid presentados en el Capítulo 5 también pueden ser mejorados. La primera puede ser extendida integrando nuevas herramientas de análisis y de ingeniería inversa, con el objetivo de permitir la extracción de un mayor número de características. Esto puede ayudar a desarrollar mecanismos de clasificación y detección más avanzados y también a construir nuevos *datasets* más completos. En este sentido, OmniDroid puede mejorarse de distintas formas: incrementando el número de muestras, mejorando la descripción de las familias de *malware* contenidas para poder ser utilizado en tareas de clasificación, o incluyendo un mayor número de características.

Abstract

Android has been intently picked as the main target by many malware creators for designing new malicious applications. Every day, thousands of new malware samples try to circumvent the security measures implemented by Android applications stores, aiming to infect new devices. In order to tackle this problem, it is required to research and develop mechanisms able to classify large amounts of suspicious samples automatically, detecting those that contain a malicious payload.

This thesis studies and addresses the application of machine learning techniques for the construction of Android malware detection mechanisms taking into account different perspectives. Furthermore, the classification of Android malware into families is also addressed. A preliminary in-depth study of the Jisut family of Android malware has allowed to reveal some of the most important practices employed and which must be considered when facing these two tasks.

In the first place, machine learning techniques are applied as the core element to build Android malware detection methods aimed at deciding accurately whether an application is malware or benignware. For that purpose, the behaviour of each application is described through groups of static and dynamic features, which are modelled using a Markov chains based representation. Then, ensemble classifiers are applied, showing how static features provide better results in comparison to dynamically extracted features. A fusion approach of both categories of features is also proposed, showing improved performance in comparison to models relying on a particular set of features.

In the second place, the classification of Android malicious applications into malware families is also tackled in this dissertation, an essential task which seeks to minimise the damages caused and to properly identify groups of malware. Deep learning architectures, classic machine learning algorithms, and different techniques for dealing with imbalanced data are tested in this case. The results evidence that these techniques allow to develop accurate family classification methods. The resilience of these methods against adversarial attacks is also analysed. A targeted attack against a state-of-the-art classifier is proposed, showing that it is possible to force the classifier to allocate samples to a fictitious, random, and new malware family or even to a previously selected destination family.

Finally, an open source framework called AndroPyTool is presented. It integrates different state-of-the-art malware analysis tools with the main goal of providing the research community with an integrated tool for the extraction of a wide set of static and dynamic features. Using this tool, the OmniDroid dataset is built and publicly released, containing both static and dynamic features extracted from benign and malicious Android applications.

Contents

List of Figures

List of Tables

Part I

Report

Chapter 1

Introduction

"If knowledge can create problems,
it is not through ignorance
that we can solve them."

- Isaac Asimov, *Asimov's New Guide To Science*

This chapter heads this dissertation to present the motivation behind this work, providing the necessary context to understand the underlying problem and the reasons which compel research to provide knowledge and solutions. The first two sections of this chapter aim at this purpose: Section 1.1 outlines the motivation of this work, whereas Section 1.2 traces the problem presented. Then, Section 1.3 proposes six Research Questions that this work tries to answer. Section 1.4 presents the structure of this thesis and finally Section 1.5 and Section 1.6 summarise the main contributions and publications associated to this research.

1.1 Context and Motivation

Cyber attacks are currently one of the most critical and important issues which modern society is facing. These attacks are an "attempt to destroy, expose, alter, disable, steal or gain unauthorized access to or make unauthorized use of anything that has value to the organization" according to ISO/IEC 27000:2009 [iso12]. The efforts dedicated to tackle these attacks have entailed enormous costs, reaching $86.4 billions in 2017 [gar17]. While the shapes in which these cyber attacks are presented and perpetrated are varied, there is also a large number of mechanisms and techniques deployed to deal with them.

This work focuses on a particular kind of attacks, those performed through executable files containing a malicious payload, also known as **malware**, software whose purpose is to cause damages in computers, trying to disrupt their normal operation [cam18]. These attacks have a long trajectory, from the emergence of the first viruses in the 70's. Since then, they have evolved to different shapes, implementing complex mechanisms; trying to circumvent antivirus and to reach the victim; or making use of advanced obfuscation techniques designed to avoid their detection.

The major problem that current malware involves is wide and complex. Malware has proved to be a powerful tool to perform large scale attacks, targeting to a large number of users and

pointing to critical infrastructures. A recent example of this kind of attacks is the WannaCry ransomware [Ehr17], which affected thousands of not properly updated computers encrypting data and demanding a ransom. Taking this as an example, there are two major tasks when facing this kind of attacks: to build malware detectors able to filter and classify suspicious samples which integrate malicious pieces of code and, in second place, to mitigate the damages caused when they have succeeded in circumventing the detection mechanism. This thesis focuses on the former problem, as an offline task in which suspicious samples are analysed in order to make a decision on the malicious or harmless nature.

Furthermore, malware is not only present in personal computers, but in almost every smart device present in our lives. Therefore, problems associated with the existence of malware gain importance when taking into consideration the massive amount of devices around us and their key role in our daily live. Our private and sensitive data become compromised. The most important exponent of these devices are smartphones, in which we store photos, messages and many other personal information along with bank, medical and other applications. The protection of these devices against malware constitutes a major task.

Regarding these devices, Android is both the mobile operating system representing the highest market share worldwide[1], close to the 80%, and the most targeted platform to create malware [Cor17], receiving 99% of all mobile malware. The possibility of installing applications from non official and different market stores, or the huge amount of new applications found every day are some of the causes of this enormous interest in Android. Nevertheless, here too lies one of the most critical barriers against the propagation of malware and to avoid the infection of users' devices: the implementation of filters which successfully discard those applications that contain malicious pieces of code before being published in the stores.

Thus, Android malware has adopted different shapes, such as scareware or ransomware, and has originated different families of malware. Grouping malicious applications into sets which share common behavioural patterns and intentions is an essential task to analyse and understand Android malware. Families such as BaseBridge, Plankton, Jisut or FakeRun have resulted in thousands of different applications, presenting both slight and significant differences, which have successfully infected devices all over the world. Android malware is a serious, huge and hard-to-solve problem.

Efforts dedicated to counteract malware focus on the design and implementation of filters which can decide accurately if a suspicious sample can be considered as benign or as malicious. Through the extraction of a series of behavioural markers, it is possible to evaluate the range of actions that the application can take and to adopt a decision. The importance of this task is beyond question, however, the large amount of new applications found every day make it mandatory the employment of tools able to deal with large amounts of samples automatically. Thus, it is necessary to conduct research towards the study of mechanisms which can automatise this task, avoiding malware samples to reach the users.

In this scenario, machine learning techniques emerge as a powerful solution to tackle this problem. They can be leveraged as an instrument to deploy malware detectors which can manage huge numbers of applications and which can provide a categorization of malicious or benignware based on a previous training process from already labelled samples.

[1] `http://gs.statcounter.com/os-market-share/mobile/worldwide`

1.2 Problem statement

This work aims to tackle the Android malware detection problem from different perspectives through the application of machine learning techniques. Thus, this research is intended to study the feasibility of these techniques when applied to solve the highlighted problem, providing the necessary instruments to undertake this combination and designing novel mechanisms geared towards detecting and classifying malware accurately.

More specifically, this research focuses on machine learning classifiers, those supervised models which after a training process from labelled examples are used to predict the class of new unlabelled examples in the future [HPK11]. These models are employed in this work to solve two different tasks. On the one hand, machine learning classifiers can be used as malware **detection** models able to characterise suspicious samples into two different categories: malware or benignware. On the other hand, they have also been used as family **classification** models, those aimed at determining the malware family of samples already labelled as malicious samples. Subsequently, the terms *benign samples*, *benignware* and *goodware* will be used interchangeably to refer to harmless applications, while *malware* and *malicious samples* will be used when referring to applications with malicious intentions.

While both tasks are faced through the use of machine learning classifiers, the term *detection* is commonly used in the literature to refer to a two-class categorisation (where *malware* and *benignware* are the possible classes), the term family *classification* refers to a multi-label classification process. Both tasks are equally important in the fight against malware. While the detection of malware represents the first and major barrier to defeat malware, the triage process according to which the family of suspicious samples is decided remains as a major task, since it permits to mitigate the damages and to limit the propagation of malware properly.

Whether for detection or malware family classification, building these tools with machine learning models entails a series of steps, from the collection of a representative set of samples to the validation and testing of the trained models. Thus, a procedure similar to a classic data mining process involving the following steps can be identified [Sax18]:

1. **Samples collection**: A representative collection of both benignware and malware is required. In case of family classification, samples from different malware families have to be gathered.
2. **Feature extraction**: Through malware analysis and reverse engineering tools, a series of features able to describe the behaviour of each application and to make differences between malicious and benign traces are extracted.
3. **Training:** The selected machine learning algorithms are trained with varied samples represented as vectors containing the features extracted.
4. **Testing:** The last step involves testing the models trained to evaluate their validity.

In the first place, the **collection** of samples plays a key role in developing machine learning aided Android malware detectors. A good selection allows to avoid bias and to properly train and evaluate the models designed. Regarding the **feature extraction** step, it involves a series

of techniques which allow to extract varied features [SH12]. Here it is possible to follow two main approaches, related to the use of static and dynamic analysis procedures.

Features obtained using static analysis are those that can be extracted from the different resources contained in the executable file. In contrast, dynamic analysis makes reference to a process in which the application is executed in a sandbox and features are extracted in runtime. In the **training** step, a plethora of algorithms can be used, such as decision trees, bayesian networks or deep learning models. The election mainly depends on the type of malware analysis approach followed, the selection of features and of their representation. Finally, the **testing** step allows to check the performance of the model using fresh samples.

All this process conforms a complex task where many and varied specific techniques have to be used at each stage. In order to study the use of machine learning techniques to solve the two raised tasks, malware detection and family identification, this research has been structured around the following objectives, all of them aimed at extending current research:

1. To study the most important android malware behavioural patterns which must be taken into consideration when designing Android malware detection tools.
2. To develop the necessary tools for the automated extraction of hybrid features.
3. To generate a comprehensive dataset of features extracted from labelled samples to train machine learning aided tools.
4. To study, develop and evaluate machine learning models for Android malware detection.
5. To study, develop and evaluate machine learning models for Android malware family classification.
6. To assess the protection of machine learning models against adversarial attacks.

First of all, it is necessary to study the most important behavioural patterns exhibited by malware and which need to be taken into account when designing malware detection and classification tools. By analysing samples of Android malware families, rich information can be identified regarding the most important malicious practices implemented. This is primarily important when determining the most appropriate set of features and malware analysis approach to perform this task. Besides, this process requires automated tools able to extract the set of features based on the malware analysis methods selected.

Once defined the space of characteristics where the behaviour of each application is represented, feature vectors are extracted from large sets of samples in order to train the machine learning models selected. In this case, representative batches of both goodware and malware are required for designing malware detection tools. When focusing on family classification, labelled malware from different families will be needed.

A plethora of existent machine learning models can be used to face the problem at hand. However, given its importance, the most appropriate methods in terms of performance must be pursued trying to minimise as much as possible the number of instances incorrectly classified. While current research has already faced this problem, this work tries to take a step further on this issue and to provide knowledge for building stronger models.

Finally, the security of the detection mechanism itself has also been considered. Recent research [XQE16, CKOF09, GPM+16], and specially in the case of Android [MXM+16], evidences that machine learning aided detection and classification models can be defeated by using adversarial learning techniques that are able to lead the classifier to deliver misclassifications. In this work, the security of these mechanisms is studied by implementing an attack against a state-of-the-art malware classifier.

1.3 Research questions

The previous problem statement presented represents a series of issues that this dissertation aims to tackle, with the ultimate goal of helping to develop accurate and secure malware detection and classification mechanisms. It composes a series of goals and questions which need to be answered. For that purpose, the different objectives of this research have been articulated in the form of six Research Questions to be addressed and answered:

- **RQ1:** Which are the most important malicious practices among Android malware samples to be considered when designing detection tools?
- **RQ2:** Is it possible to find large labelled dataset of features extracted from Android malware and benign samples?
- **RQ3:** Can machine Learning classification methods be combined with static features to detect Android malware accurately?
- **RQ4:** Is it possible to apply machine Learning over dynamic traces to detect Android malware accurately?
- **RQ5:** Is it possible to combine static and dynamic features in order to build more effective detection mechanisms?
- **RQ6:** Is it feasible to attack machine learning classifiers to produce family misclassifications?

1.4 Structure of the thesis

This thesis is presented as a compendium of publications and structured into **Part I** and **Part II**. The former exposes the general lines of this research, describing the necessary context, synthesising the main results obtained, and presenting a series of conclusions. The second part contains four papers published in international journals which define the core of this dissertation, and on which Part I is based. The first part is composed by the following chapters:

- **Chapter 1: Introduction.** This is the present chapter. It includes the motivation and the definition of the problem addressed in this research. Then, a series of Research Questions are proposed, followed by the structure of this thesis and finally listing the publications which shape this dissertation.

- **Chapter 2: Android malware detection and classification from a machine learning perspective.** In this second chapter, the context and details required before facing the Android malware detection problem from the use of machine learning techniques are presented. The Android platform is also described taking into account the architecture and its security properties. Then, malware designed for Android is presented, with a special focus on a specific family of Android ransomware [MHCC18]. Then, current malware analysis mechanisms are discussed and finally the use of machine learning for Android malware detection and classification is analysed from a state-of-the-art research perspective.

- **Chapter 3: Applying machine learning techniques for Android malware detection and classification.** The Android malware detection and classification problems are studied in this chapter, extending existing research with novel mechanisms to perform accurately these two tasks [MRFC18, MLCC18]. Different section are devoted to analyse the use of different types of features and machine learning models.

- **Chapter 4: Adversarial machine learning in the Android malware domain.** The previous use of machine learning is evaluated when facing adversarial attacks trying to disrupt the correct detection or classification operation [CMM+18]. An attack is proposed and succesfully implemented. The final section of this chapter raises a countermeasure which can succesfully deal with this kind of attacks.

- **Chapter 5: AndroPyTool and OmniDroid.** This chapter focuses on describing in detail two contributions: a tool for the automated extraction of hybrid characteristics from Android applications, and a large and comprehensive dataset publicly available for training and testing machine learning detection algorithms such as the ones previously presented in Chapter 3.

- **Chapter 6: Conclusions and future work.** The final chapter aims to present a series of useful conclusions based on all the results and findings obtained in the course of this research. In the second place, different possible lines of future work are suggested.

1.5 Publications of the compendium and Contributions

This section presents the list of articles on which this thesis as a compendium of publications is based. For each of them, quality indices and contributions of the PhD candidate are described:

(IJ-1) **Martín, Alejandro**; Julio Hernández-Castro & David Camacho: "An in-depth study of the Jisut family of Android ransomware." *IEEE Access*, Vol. 6, pp. 57205-57218, 2018, DOI: `10.1109/ACCESS.2018.2873583`
Impact factor = 3.557 (JCR, 2017) [Q1, 24/148, Computer Science, Information Systems].

  - **Overall contributions:** This work performs a detailed study of a particular family of Android ransomware, as described in Chapter 2. This allows to obtain a general picture of the most important patterns implemented by Android malware and it is useful in the design of machine learning methods for Android malware detection presented in Chapter 3.
  - **Contributions of the PhD candidate:**

  - First author of the article.
  - Contributions made in the conception of the presented idea.
  - Analysis and application of reverse engineering techniques over a large number of varied samples of the Jisut family.
  - Analysis of the common implementation and behavioural patterns among samples.
  - Co-author of the interpretation and discussion of results provided.
  - Co-author of the manuscript, figures and tables presented.

(IJ-2) **Martín, Alejandro**; Raúl Lara-Cabrera & David Camacho: "Android malware detection through hybrid features fusion and ensemble classifiers: the AndroPyTool framework and the OmniDroid dataset." *Information Fusion*, Ed. By Elsevier. Accepted. DOI: `10.1016/j.inffus.2018.12.006`
Impact factor = 6.639 (JCR, 2017) [Q1, 4/103 Computer Science, Theory & Methods].

- **Overall contributions:** This article presents a tool for the analysis of Android applications and a large dataset which are described in Chapter 5. It also studies the use of machine learning models, specifically ensemble classifiers, for building Android malware detection mechanisms, as presented in Chapter 3.
- **Contributions of the PhD candidate:**
  - First author of the article.
  - Contributions made in the conception of the presented idea.
  - Contributions made in the design of the tool and dataset presented.
  - Implementation of the tool presented.
  - Generation of the dataset proposed.
  - Design and execution of the experiments.
  - Co-author of the interpretation and discussion of results provided.
  - Co-author of the manuscript, figures and tables presented.

(IJ-3) **Martín, Alejandro**; Víctor Rodríguez-Fernández & David Camacho: "CANDYMAN: Classifying Android malware families by modelling dynamic traces with Markov chains." *Engineering Applications of Artificial Intelligence*, Volume 74, 2018, Pages 121-133, DOI: `10.1016/j.engappai.2018.06.006`
Impact factor = 2.819 (JCR, 2017) [Q1, 32/132, Computer Science, Artificial Intelligence].

- **Overall contributions:** This article proposes novel mechanisms for the classification of Android malware into families using dynamically extracted features, which are described in Chapter 3. Different machine learning techniques are used in combination with a representation of features based on Markov Chains.
- **Contributions of the PhD candidate:**
  - First author of the article.
  - Contributions made in the conception of the presented idea.
  - Contributions made in the design of the classification mechanism presented.
  - Implementation of the experiments using machine learning techniques.
  - Contributions made in the design and execution of the experiments.
  - Co-author of the interpretation and discussion of results provided.

  - Co-author of the manuscript, figures and tables presented.

(IJ-4) Calleja, Alejandro; **Alejandro Martín**, Héctor D. Menéndez, Juan Tapiador & David Clark: "Picking on the family: Disrupting android malware triage by forcing misclassification." *Expert Systems with Applications*, 95 (2018): 113-126, DOI: 10.1016/j.eswa.2017.11.032
Impact factor = 3.768 (JCR, 2017) [Q1, 20/132, Computer Science, Artificial Intelligence].

- **Overall contributions:** The aim of this work is to analyse the protection of machine learning aided Android malware detection tools against adversarial attacks, which try to force a classifier to produce misclassification. This attack, which is described in Chapter 4, also presents a countermeasure to tackle these attacks.
- **Contributions of the PhD candidate:**
  - Second author of the article.
  - Contributions made in the conception of the presented ideas.
  - Contributions made in the design and implementation of the genetic algorithm exposed.
  - Implementation of the countermeasure.
  - Co-author of the interpretation and discussion of results provided.
  - Co-author of the manuscript, figures and tables presented.

1.6 Other publications and Contributions

This section presents other related publications and conference articles which have also been published in the course of this dissertation and that present methods and results related to this work:

1.6.1 International Journals

(IJ-5) **Martín, Alejandro**; Héctor D. Menéndez & David Camacho: "MOCDroid: multi-objective evolutionary classifier for Android malware detection." *Soft Computing*, 21.24 (2017): 7405-7415, DOI: 10.1007/s00500-016-2283-y
Impact factor = 2.367 (JCR, 2017) [Q2, 45/132, Computer Science, Artificial Intelligence].

- **Overall contributions:** The contribution of this work are related to Chapter 3, describing a novel method for the detection of Android malware in this case through the use of a multi-objective evolutionary classifier.

(IJ-6) **Martín, Alejandro**; Raúl Lara-Cabrera; Félix Fuertes-Hurtado; Valery Naranjo & David Camacho: "EvoDeep: A new evolutionary approach for automatic Deep Neural Networks parametrisation." *Journal of Parallel and Distributed Computing*, 117 (2018): 180-191, DOI: 10.1016/j.jpdc.2017.09.006
Impact factor = 1.815 (JCR, 2017) [Q2, 33/103, Computer Science, Theory & Methods].

- **Overall contributions:** This articles propose a new method to deal with the parametrisation problem of Deep Neural Networks and it is related to the contents described

in Chapter 3, specifically to Section 3.4.1, where Deep Learning models are used for Android malware family classification.

1.6.2 Conferences

(IC-1) **Martín, Alejandro**; Raúl Lara-Cabrera & David Camacho: "A new tool for static and dynamic Android malware analysis." *In the 13th International FLINS conference on Data Science and Knowledge Engineering for Sensing Decision Support*, pp. 509-516 (2018). DOI: 10.1142/9789813273238_0066.
Rank B (2018), CORE ERA, Artificial Intelligence and Image Processing.

- **Overall contributions:** This conference article presented an overview of the framework AndroPyTool, and it is related to Section 5.1, where this tool is presented.

(IC-2) **Martín, Alejandro**; Félix Fuentes-Hurtado, Valery Naranjo & David Camacho: "Evolving deep neural networks architectures for Android malware classification." *2017 IEEE Congress on Evolutionary Computation (CEC)*, San Sebastián, Spain, 2017, pp. 1659-1666. DOI: 10.1109/CEC.2017.7969501.
Rank B (2017), CORE ERA, Artificial Intelligence and Image Processing.

- **Overall contributions:** This article is focused on the use of deep learning architectures for Android malware classification and is related to Section 3.4.1.

(IC-3) **Martín, Alejandro**; Héctor D. Menéndez & David Camacho: "Genetic boosting classification for malware detection." *2016 IEEE Congress on Evolutionary Computation (CEC)*, Vancouver, BC, 2016, pp. 1030-1037. DOI: 10.1109/CEC.2016.7743902.
Rank B (2017), CORE ERA, Artificial Intelligence and Image Processing.

- **Overall contributions:** This work presents a mechanism for building Windows malware detection mechanisms based on a set of independent classifiers trained for different regions in the space of samples and guided by a genetic algorithm. It is related to Chapter 3.

(IC-4) **Martín, Alejandro**; Alejandro Calleja, Héctor D. Menéndez, Juan Tapiador & David Camacho: "ADROIT: Android malware detection using meta-information." *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, Athens, Greece, 2016, pp. 1-8. DOI: 10.1109/SSCI.2016.7849904.
Rank C (2017), CORE ERA, Artificial Intelligence and Image Processing.

- **Overall contributions:** This work focuses on metainformation to train machine learning classifiers for Android malware detection. It is related to Chapter 3.

(IC-5) **Martín, Alejandro**; Héctor D. Menéndez & David Camacho: "String-based Malware Detection for Android Environments." *10th International Symposium on Intelligent Distributed Computing (IDC 2016)*, París, France, Studies in Computational Intelligence, vol 678. Springer, Cham, DOI: https://doi.org/10.1007/978-3-319-48829-5_10

- **Overall contributions:** A new approach for detecting malware based on a representation of strings is proposed in this article and related to the contents of Chapter 3.

(IC-6) **Martín, Alejandro**; Héctor D. Menéndez & David Camacho: "Studying the Influence of Static API Calls for Hiding Malware." *17th Conference of the Spanish Association for Artificial Intelligence*, CAEPIA 2016, Salamanca, Spain. Lecture Notes in Computer Science, vol 9868. Springer, Cham. DOI: 10.1007/978-3-319-44636-3_34.

- **Overall contributions:** The study performed in this article is related to the features employed in Chapter 3 for Android malware detection and classification.

Chapter 2

Android malware detection and classification from a machine learning perspective

"People think of education as something they can finish."

- Isaac Asimov

The detection of malware executables conforms an essential task in many scenarios. From preventing users to get infected with different kinds of malware, trying to steal private information, or to stop network intrusions in large corporations. The malicious payload included in these malicious executables can be defined as "any code added, changed, or removed from a software system in order to intentionally cause harm or subvert the intended function of the system." [MM00, p. 33]. This problem, typically tackled using manual procedures, has adopted new dimensions which involve the use of new instruments able to automatise this process with large amounts of suspicious samples. Among these, machine learning techniques represent a promising solution.

Machine learning has positioned itself as a powerful mechanism to solve diverse, large and complex problems of different nature. This concept is classified as a subfield of Artificial Intelligence and it is a fundamental component of many Data Mining processes, those related to the extraction of knowledge from large amounts of data [HPK11]. In particular, to define the term "machine learning", Kevin P. Murphy's definition, in his book *Machine Learning: A Probabilistic Perspective*, is one of the most comprehensive and precise: "a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty" [Rob14].

This work is focused on the application of this set of techniques to detect malware designed for the Android operating system. The present chapter focuses on providing the necessary context to face this problem. First, the Android platform is introduced, presenting details of the different layers composing its architecture. Then, malware targeting this platform is studied, paying special attention to a particular family of malware called Jisut. Then, two possible approaches to analyse and extract descriptive features from Android applications are presented. Finally, literature studying and proposing mechanisms to build malware detectors and family classifiers based on machine learning models is reviewed.

2.1 An introduction to the Android operating system

Android is one of the most important mobile operating systems or platforms at present, included in a large number of users' smartphones, tablets or smart TVs, among others. The first version was released in November 2007 [RLMM09]. Since then, it has experienced an incredible growth, and it has become the most used and extended mobile operating system. In September 2018, Android represented the 76,61% of market share worldwide, whereas its first competitor, iOS, represented the 20,66%. This operating system is characterised by its open source architecture, its cross-platform approach and its Linux kernel.

The wide range of possibilities that this operating system offers to manufacturers, who can use it for their own built devices, the possibility of running applications developed in different platforms, or the possibility of developing new software for free, are some of the reasons explaining the high market share of Android. The next subsections will present the different layers composing the Android architecture, and some of the most important technical details regarding its security components.

2.1.1 Architecture of the Android operating system

The Android architecture[1] is structured in a series of layers (see Fig. 2.1) offering different components and functionalities at different levels of abstraction, from hardware to system applications. The bottom layer is composed by the **Linux Kernel**. This is the core system of Android, on top of which all the components and layers are deployed. It also manages some of the most important security related policies. The next layer, in ascending order, is the **Hardware Abstraction Layer (HAL)**, in charge of allowing access to the hardware from components in upper layers. It contains independent modules for each hardware component.

Two layers are placed above. The first one, the **Native C/C++ Libraries**, used by developers who build their applications in any of these programming languages or also employed by the Java API. The second one is the **Android Runtime (ART)**, which has replaced Dalvik. This environment allows to run multiple virtual machines where applications developed in Java execute in isolation.

On top of the two previously described parallel layers, it is possible to find the **Java API framework**. Through this API, it is possible to access all functionalities offered by the Android Operating system. The main objective of this layer is to facilitate the process of developing new applications or reusing code in a rich environment designed to help the developer. Finally, the top layer contains a set of system applications. They provide basic functionalities to the user, such as internet browser, phone, email or calendar, but they can be replaced by other applications installed by the user.

In Android, applications are distributed in files with extension *.apk*, which stands for Android PacKage (APK). These compressed files in *zip* format contain the necessary code, data, and resources required to execute the application. Section 2.4 describes the structure of an APK.

[1] https://developer.android.com/guide/platform/

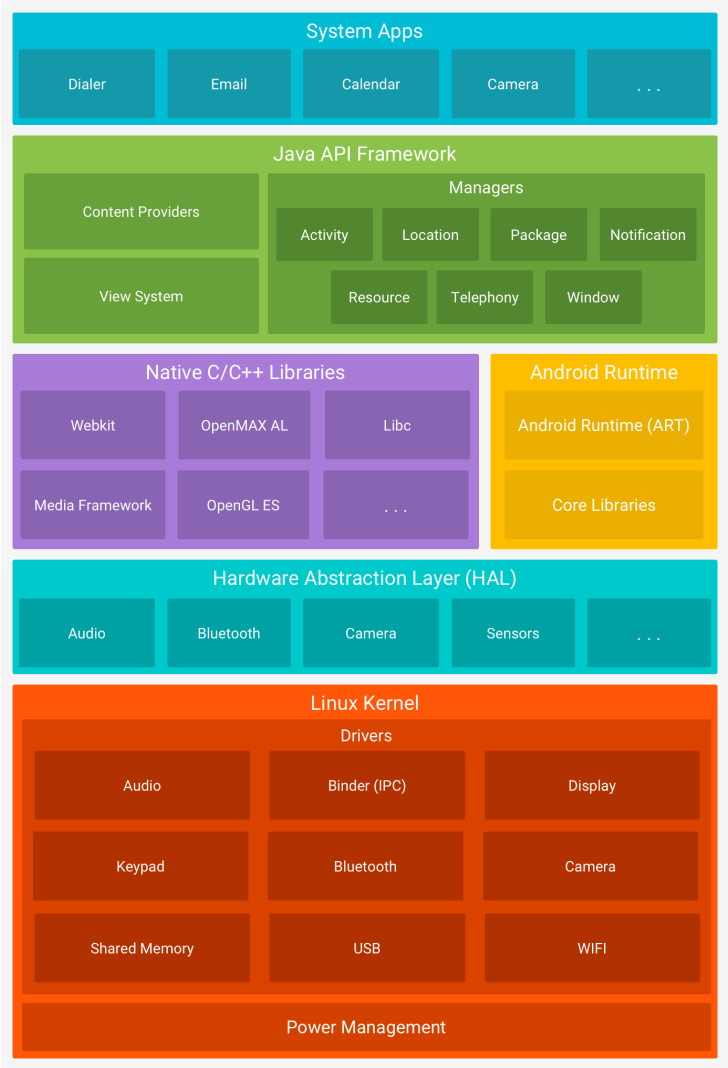


Figure 2.1: Diagram showing the different layers that compose the Android operating system architecture, from the module which access to hardware functions to the system applications level[1].

2.1.2 Android security properties

The Android platform merges a variety of security policies and instruments in order to ensure the reliability and integrity of the system. The different security mechanisms implemented allow to restrict the access of new applications to functionalities and resources provided by the device[2]. One of the most basic security policies in Android is based on the sandboxing approach to execute applications. In this sense, each program runs in an isolated environment, composed by an individual virtual machine. At the Linux level, each sandbox is associated with an unique user ID, in order to guarantee the complete encapsulation.

Another important security mechanism is implemented through the Android Manifest. This file, which is included in all Android applications, contains not only different metadata, such as package name or a definition of activities, but also a declaration of the functionalities required

[2] `https://source.android.com/security/overview/app-security`

to be executed. For instance, a list of permissions is mandatory in order to define which system components the application requires (i.e. access to the camera or to location information).

Those system calls requiring the declaration of the according permissions fall under the scope of the so-called sensitive APIs. The user must grant access to all demanded permissions when installing the application. However, from Android 6.0, permissions are asked in runtime when needed. This new policy aims to ensure that the user explicitly grants permission to access specific functionalities, instead of approving a large set of permissions without actually considering if the applications should require them.

There are other components which are specially protected. In the case of the SIM card, only the operating system is able to manage the information contained. Access to device metadata, including system logs or the phone number, is also particularly protected, requiring the user to explicitly approve the action. Another important component aiming to enhance the security component in Android lies in the Interprocess Communication mechanism, which allows to safely share data among applications.

Finally, there are two particularities of the Android ecosystem to highlight. They present important security barriers in the form of policies to control the access to very sensitive operations and which, if granted, represent a very serious risk. On the one hand, to gain *root* privileges entails a number of problems. A malware using these concessions is able to take full control of the device. On the other hand, the Device Administrator API enables applications to perform a wide set of high level operations, such as removing all user's data or locking the device.

2.2 Malware designed for Android

Android has been attacked by malware since its appeareance in 2008. Shortly after this date, the first malware specifically designed for this platform, particularly a Trojan [DHQ+14], was found. From then, attackers have repeatedly pointed this platform as the main target for their attacks, mainly due to different facts such as its large market share. According to G DATA analysts [Lue18], 3,002,482 new malware samples for Android were discovered in 2017. Throughout all these years, different families of malware have been discovered with different intentions and implementations.

This section introduces some of the most important and spread families of Android malware, aiming to provide a general overview of the intentions, implementation details and risks that they present. These families shape different types of malware, such as Trojans, scareware[3] or ransomware, and perform certain malicious actions such as dialling premium phone numbers or stealing personal information.

Below is presented a brief description of some of the most important Android malware families [DHQ+14]:

- **FakePlayer:** acts as a fake movie player which actually sends SMS messages to premium numbers.

[3] A malware which tries to scare the victim.

- **Geimini:** is a repacking of a benign application whose main purpose is to steal user's private data and which is able to receive instructions through the network. This family is also able to display a web page, to send SMS messages or dialling premium phone numbers.
- **DroidDream:** represents one of the most spread malware families in Android and one of the first which reached the Google Play Store. The main objective of this Trojan is to control the device at the Linux level.
- **Jisut:** a ransomware which lock user's devices and in some cases encrypts user's data. The application also shows a message informing that a ransom must be paid in order to unlock the device or to decrypt user's data.
- **BaseBridge:** performs similar actions to the ones shown by *Geimini*, sending premium SMS messages. The malicious payload is stored in a hidden file installed when root access is achieved.
- **DroidKungFu:** this Trojan shows a more complex implementation. It is able to delete user's files, steal information or avoid its detection.
- **Plankton:** it is also presented as a repacking of other applications. It receives orders from a remote server and also a *jar* file which contains malicious code.
- **GingerMaster:** it repackages some applications of the Chinese market and collects and sends system information to a server. It is also known as GinMaster [Yu13].
- **FakeRun:** is an example of malware which instead of stealing private data, tries to obtain money by the use of advertising. In fact, it acts as a disguised ad blocking application.

From a general point of view, Android malware families are varied and perform diverse actions. They steal private information, lock user's devices or demand money (ransomware). In order to provide implementation details at a lower level, a further analysis was performed in this dissertation. For that purpose, a specific malware family called Jisut was used, where the different variants created over time are clearly visible.

2.3 An inspection of the Jisut family of Android malware

With the goal of shedding light on how Android malware is developed, to observe some of the most important artefacts employed by attackers or to know how the malicious payload is triggered, a thorough analysis of a specific malware family was done in this work [MHCC18]. This novel study traces in detail different implementations of a particular ransomware with the objective of drawing conclusions that could help to design better detection tools.

One of the main goals of this study is to analyse behavioural patterns in evolutionary terms, that is to say, to inspect the changes instrumentalised by malware samples over time in order to become more complex. Literature so far has focused on building detection methods [MMC17] or classification tools with a particular emphasis on obfuscation techniques [STDA+17]. However, research focused on studying a particular family of Android malware is limited to an analysis of the GinMaster family [Yu13]. In this work, a ransomware called Jisut was chosen for this

purpose. This malware has been mainly distributed in the Chinese market, where it started spreading in 2014. Jisut shapes a ransomware which demands a ransom in order to decrypt user's data or to unlock the device, after having deployed the malicious payload. By analysing samples of this family, it is possible to discern subsets that include certain modifications while sharing common structural patterns.

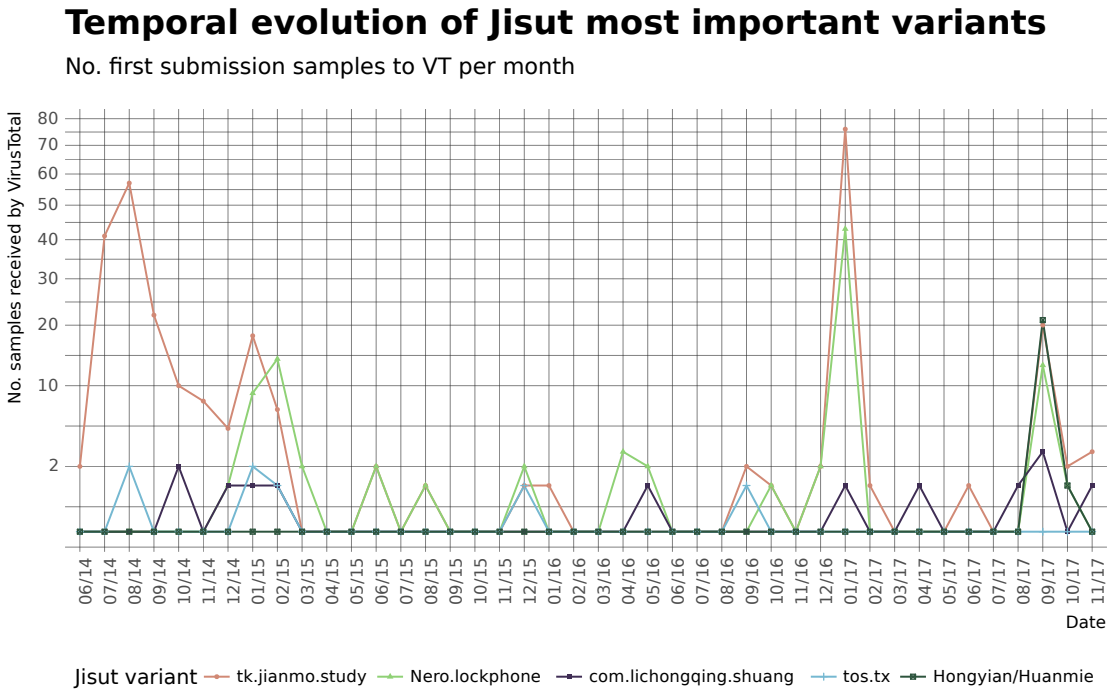


Figure 2.2: Evolution over time in the number of submissions of new Jisut ransomware samples.

In order to collect samples of the Jisut ransomware, the VirusTotal Intelligence portal[4] was used. A large number of queries was required with the goal of retrieving varied samples, searching for the term *Jisut* in the antivirus results. Due to the known existing problem when naming malware samples [MBSZ11], under which antivirus employ different labels to name the same family of malware, manual searches were necessary to obtain a large set of samples of this family.

Figure 2.2 shows the evolution of the most important variants of Jisut according to the number of samples submitted to VirusTotal per month. As of October 2017, the total number of samples labelled as Jisut is 4,693. In all these submissions it can be observed a seasonal component, with different frequency peaks. Furthermore, it is also remarkable that the *Nero.lockphone* variant, which was found for the first time at the beginning of 2015, shows a very similar trend to the one shown by the original family, *tk.jianmo.study* over time. This leads to affirm that different attackers could be working cooperatively.

(a) **(b)** **(c)** **(d)** **(e)**

Figure 2.3: Screenshots of the five variants analysed of the Jisut ransomware. Each picture has been captured once the malicious application started, informing the user of the actions to unlock the device. a) Variant *tk.jianmo.study* (2014). b) Variant *lichongqing_shuang* (2014). c) Variant *Nero.lockphone* (2015). d) Variant *qqmagic* (2016). e) Variant *Hongyan - Huanmie* (2017).

2.3.1 Description of the most relevant Jisut variants

Through the following subsections, the most important variants found of the Jisut family are analysed. Fig. 2.3 shows a screenshot of each of these variants. Furthermore, a web page was created (`https://aida.ii.uam.es/jisutnoransom/`) providing more information, mainly related to the deactivation mechanisms employed by this malware.

The *Jianmo* variant

Jianmo can be considered the first broadly spread variant of the Jisut family of Android ransomware. It is represented by the package name *tk.jianmo.study*. When this piece of malware is executed, it displays a screen informing the user that his/her device has been infected by a Trojan (see Fig. 2.3a). A second text states that a message has to be sent through the QQ messaging service in the next 24 hours in order to receive instructions to get the unlock code. The malware acts as a lock-screen malware, configuring this screen to be permanently shown and overriding any closing action. The remaining time is shown on the screen. The malware circumvents the system clock by writing the remaining time to a file, avoiding the user to save time by turning the clock back. However, in many samples of this variant, the threat is not implemented, so user's files are never removed.

It can also be seen that the application includes functionality to detect if it has been closed or if the device was rebooted. When these events are detected, the malware restarts and the lock screen emerges again. In other piece of code, it can be noticed that the malware implements a keylogger module, in charge of detecting a series of key sequences. These two events highlight the importance of the *receivers* declared by the suspicious applications, since they reveal events monitored by the application.

Different samples of this family have been analysed in order to reveal general implementation

[4] `https://www.virustotal.com/intelligence/`

patterns. For instance, it is possible to identify almost similar samples but which include different package names. This could be aimed at avoiding detection by signature. Furthermore, there are also samples featuring different unlock key sequences or that allow to introduce an alphanumeric deactivation code.

The *lichongqing.shuang* variant

The first samples of this second slightly different variant of the original Jisut family were found in 2014. In this case, it is presented in the form of scareware. When this malware is launched, it shows a screen (see Fig. 2.3b) with a picture attempting to scare the victim. At the same time, a sound of a person screaming is played at maximum volume. Aiming to avoid the user to decrease the volume, the malicious application continuously invokes the necessary system call to revert this action in a loop. In this case, a control of the system calls invoked by the malware can help to detect this kind of actions. The scareware also monitors when the user taps a specific section of the screen, moment in which a text box appear. If the user introduces a specific numerical code in this field, this lock screen malware is deactivated.

The *nero.lockphone* variant

This new variant conforms again a very similar implementation if compared to the first variant. It started spreading in 2014 but was notoriously more important in 2015. Although the visual aspect (see Fig. 2.3c) differs from the one exhibited by the *Jianmo* variant, behaviour, code, or class names are almost the same. The main view of this malware directly shows a text box to introduce the unlock code.

The *qqmagic* variant

The *qqmagic* variant of Jisut represents a significant change in the overall family evolution. By embedding a more complex deactivation mechanism, the attacker/s try to build a more hazardous malware. This new variant (a screenshot is provided in Fig. 2.3d) does not longer save the unlock code in plain text. Instead, it shares different messages with the attacker in order to generate the final unlock code.

There are also two important details which it is important to remark in this variant. On the one hand, this variant implements the necessary code to remove user's files if the ransom has not been paid, so samples of this family can be considered as quite dangerous. On the other hand, there can be found traces revealing the use of Ijiami[5], a tool which allows to apply different obfuscation techniques.

Furthermore, the malware includes a string defining a command to be executed at the Linux level. As it can be seen in Figure 2.4, a sample[6] of this family remounts the `/system/app/` partition to grant read and write permissions. Then, a file is copied from the `/sdcard/` partition to the system applications path. Finally, read and write permissions are given to this file and

[5] `http://www.ijiami.cn`

[6] Identified by SHA-256:`506f668438477b7476674957d14407d207de1f576e5c9de2852490b43a6a013b`

```
1   void rootShell(){
2   execCommand(new String[] {
3       "mount -o rw,remount /system", "mount -o rw,remount /system/app",
4       "cp /sdcard/zihao.l /system/app/", "chmod 777 /system/app/zihao.l",
5       "mv /system/app/zihao.l /system/app/zihao.apk",
6       "chmod 644 /system/app/zihao.apk", "reboot" }, true);
7   }
8
```

Figure 2.4: Installation of a new hidden application in the qqmagic variant. The system applications folder is remounted with read and write permissions. Then, the malicious application is copied into this folder and the device is rebooted.

the reboot command is invoked. Through this process, an external application, which could be downloaded in runtime, is integrated as a new system application. However, root permissions are required to perform these operations.

The above process, which entails loading new code in runtime, conforms a major issue, since it highlights the weaknesses of static analysis techniques. Although this process reveals the existence of the necessary calls to import load libraries, this single fact cannot be attributed to a malicious behaviour, as benign applications also employ this technique.

The *Hongyan* and *Huanmie* variant

This recent variant shows a similar aspect to the SLocker family, also well known by different antivirus engines. When the main activity of this malware is launched, a screen informs the user that the device configuration is being checked. Meanwhile, a background process is encrypting all user's files. After a while, or if the application is closed and restarted, a new message claims that all user's files have been encrypted and a large number is displayed at the bottom (see Fig. 2.3e).

To recover all files, the user must contact the author of the malware through the QQ platform. Then, the user will be asked to provide the number shown in the screen and to pay a ransom of 20 yuans. In order to check if user's files are actually encrypted, a set of decoy files were saved with different extensions. Once the malware is installed and launched, all files are encrypted and a new extension added: .文件已被幻灭劫持.

Another interesting detail found is related to the large number shown in the screen. This value is calculated after applying a SHA-256 over the MD5 hash function of the IMEI number (the identifier of the device). The unlock code is generated following the same basic procedure again: the SHA-256 over the MD5 of the number shown in the screen. In summary, this code is calculated as:

Deactivation code calculation in the Hongyan-Huanmie variant
$\text{UNLOCK\_CODE} = \text{SHA-256}(\text{MD5}(\text{SHA-256}(\text{MD5}(\text{IMEI}))))$

Furthermore, this family of malware presents a favourable scenario to study how a malware evolves, integrates new modifications and gives place to new variants. After analysing a significant number of samples of the Hongyan variant, these modifications start to take shape and to build a series of ramifications which allow to extract several considerations. Fig. 2.5 represents the relations found among these samples and the changes implemented. To identify each sample, the first 8 hex characters of the SHA-256 hash are shown. Below is indicated the date of first submission to the VirusTotal portal.

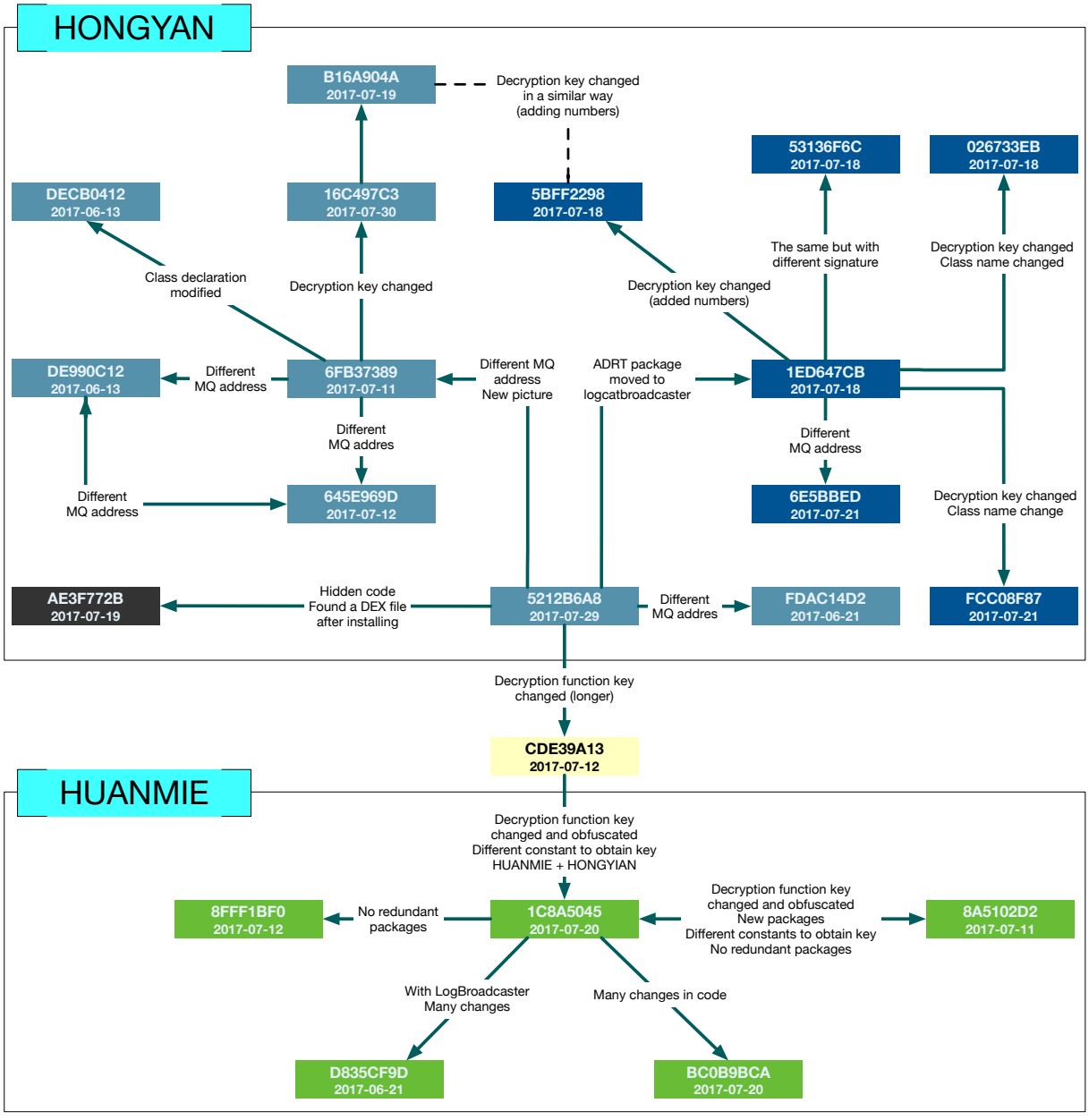


Figure 2.5: Diagram showing different samples of the Hongyan and Huanmie variants and the changes implemented.

The Hongyan variant, represented in the upper box of the figure implements small changes to build new APKs. The root of this family can be located in the sample identified by *5212B6A8*. From this one, new samples are created integrating a series of modifications. For instance,

descendants of this malware adopt different simple changes, such as a different address related to the instant messaging service used, or to include a new resource. At the left, from the sample with signature *6FB37389*, other variations are created, implementing a different encryption key or a change in a class declaration. Variants at the right include a modification in the packages structure, creating a new one called *logcatbroadcaster*. From this one, new samples include again small changes, such as new encryption keys or different signatures.

It is remarkable the process through which a new variant is created. From the above mentioned sample *5212B6A8*, a new one, represented by *CDE39A13*, transforms the decryption key function to include a more sophisticated one. This anticipates a series of samples involving a number of notorious changes and shaping a new variant, starting from *1C8A5048*. In this new malware, a separated package contains the code related to the previous Hongyan variant, in combination with a new package named `com.a.a.android.admin.huanmie`. This represents a clear step towards hampering the analysis of the sample, by deploying new changes. At the same time, it is a clear example of the evolution of malware, when old samples are used as a template to create new variants more sophisticated.

The *com.bll.apkin* variant

This is the most recent variant analysed, reported in 2017 [F-S]. The most striking characteristic of this variant lies in that the malware talks to the victim in order to demand the ransom. Thus, the malware uses a spoken message which follows the same procedure seen in previous variants: to inform the user that the QQ service must be used to contact the attacker. It requires administration privileges. This sample contains a hidden APK which is extracted in runtime. Finally, it has been found that the Jiagu 360[7] is used as obfuscation software. As it can been seen, this malware has not stopped growing and evolving to more complex shapes.

2.3.2 Conclusions extracted from the analysis

The Jisut family poses an interesting scenario where different important behavioural patterns can be identified and where important conclusions arise. In the first place, this family implements a series of incremental changes directly aimed at making the different key components more complex. For instance, hampering the extraction of the unlock code, using obfuscation techniques or hiding the malicious payload behind encrypted files, whether downloaded or already existing in the APK package, which are decrypted and loaded in runtime. Furthermore, the use of cryptographic operations represents a potential characteristic to which special attention has to be paid. They imply a key indicator in ransomware, since they are the essential mechanism to realize the threat and thus to force the user to pay the ransom.

Besides, the coding style and different artefacts implemented suggest that the developers of this malware have limited experience. Nevertheless, some of the techniques employed have a strong importance when dealing with Android malware. From a malware analysis perspective, the use of obfuscation and dynamic code loading techniques constitutes a very important step towards counteracting static based malware detection.

[7] `http://jiagu.360.cn`

2.4 Malware analysis

The term malware analysis refers to a series of techniques and processes aiming to describe the set of actions that a suspicious executable file can take. This allows to obtain the necessary information to detect the malicious payload and to contain the damages [SH12]. There are two approaches in which malware analysis techniques can be organised: **static** and **dynamic analysis**, but it is also possible a combination of both, named **hybrid analysis**. Each of these techniques present different procedures in order to extract relevant pieces of information able to describe the behaviour of the sample. Fig. 2.6 summarises the features which can be extracted depending on the static or dynamic approach followed.

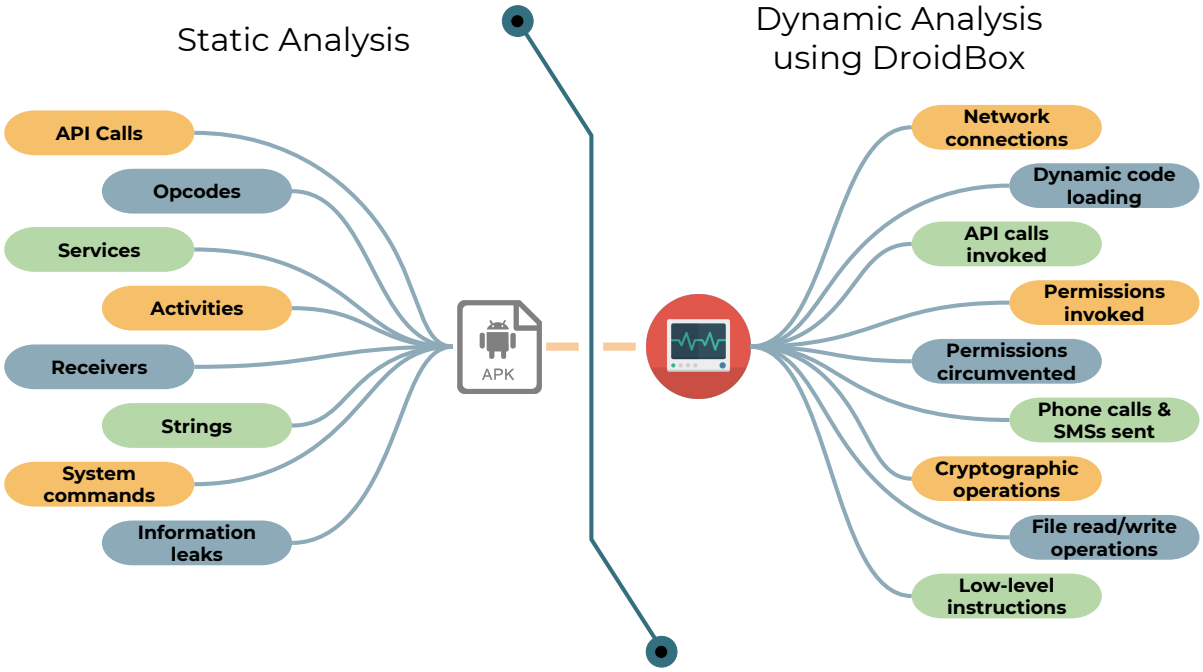


Figure 2.6: Diagram showing the different features that can be extracted following a static and a dynamic analysis approach. In case of those dynamically extracted, the DroidBox tool is considered.

2.4.1 Static analysis

Techniques under the scope of static analysis refer to the extraction of descriptive information from the executable file and which do not entail to execute the suspicious sample. This allows to build efficient and effective models to detect malware. However, obfuscation techniques pose an important obstacle against the effectiveness of this approach. Static analysis includes the use of reverse engineering techniques in order to access the set of instructions that define the application operation. Furthermore, and focused on the Android platform, a large variety of characteristics can be revealed using this kind of analysis. Information gathered from the Android Manifest or from the resources are included into this category.

Fig. 2.7 presents an overview of the different files and folders obtained after unzipping an APK. However, since some of these files contain encrypted information, it is required to use

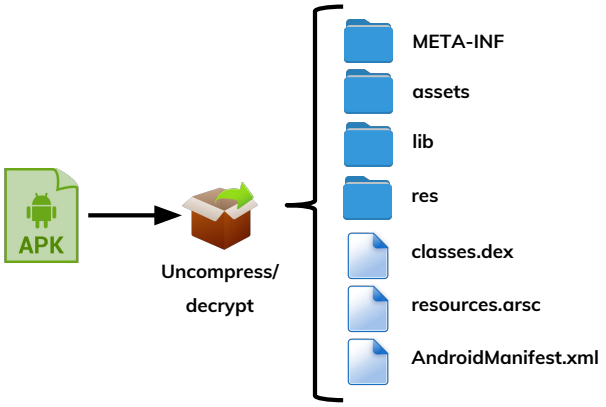


Figure 2.7: Structure of files and folders after uncompressing an APK.

specific tools such as AndroGuard [DG13] or Apktool [Win12] to extract the human readable version of each file. The files contained in the different folders provide varied information which can be used to categorise the behaviour of the sample. For instance, `/META-INF/` includes certificates, developer information or information to run the *jar* file. `/assets/`, `resources.arsc` and `/res/` are related to different mechanisms to import resources. The `/lib/` folder stores compiled libraries. Finally, two files provide the most relevant features when facing a malware analysis task and which are shown in Fig. 2.6. `classes.dex` defines the code of the application in the form of Dalvik bytecode. From here, a list of API calls, system commands or receivers defined can be retrieved. The second important file is the `AndroidManifest.xml`, which declares a list of permissions, the package name or a relation of intent filters.

2.4.2 Dynamic analysis

A dynamic analysis entails a process in which the sample is executed in a controlled environment and a monitoring service is in charge of capturing each event or action which takes place during the execution. While this kind of analysis can provide deep information never revealed by a static analysis process (mainly due to the use of dynamic code loading techniques), it is much more expensive and also less efficient. However, there is one more drawback, since existing literature evidences how it is possible to detect when a piece of code is running in a virtual environment [PVA+14].

DroidBox [Lan15, MCC17] was selected to conduct the dynamic analysis process. This open source tool implemented in Python leverages the official Android emulator to monitor a wide set of events while the suspicious sample is executed. It uses a modified version of the Linux kernel and of the system image in order to monitor every API call invoked and also to detect data leaks. In order to extract these events from outside of the emulator, all events captured are sent to the Android system log and read using the Logcat tool. In a final step, DroidBox organises all the information retrieved in different categories, where each event is associated to a timestamp indicating the moment in which it was captured. The analysis contains the following categories of events (as shown at the right part of Fig. 2.6):

- **Accessed files:** Every file accessed is included in this category. For each event, the path to the file and the timestamp are defined.

- **Closed connections:** Related to the sockets that are closed during the application execution.
- **Cryptographic API usage:** A relation of the cryptographic operations performed, including the specific algorithm, key and the operation performed.
- **Data leaks:** Flows of private information which reach an output connection, such as network or SMS calls.
- **DEX classes loaded:** A list of new packages dynamically loaded during the execution. This is a source of information which reflects important malicious patterns. Each event of this type includes a field reporting the path to the new file being loaded.
- **Enforced permissions:** The set of permissions declared and enforced by the application.
- **Files operations:** Read and write operations, including the path, performed over any file during the execution.
- **Hashes:** The MD5, SHA-1 and SHA-256 of the application.
- **Opened network connections:** In this case, the destination host and port and also the corresponding file descriptor are associated.
- **Phone calls:** A list of phone calls that could not be even noticed by the user.
- **Data received from network:** A list of data received from a particular socket, including the source, the port and the specific data transmitted.
- **Receivers:** Information related to the different Broadcast receivers declared.
- **Network data sent:** Information of outgoing connections sending data.
- **Sent SMS:** A relation of messages sent in the form of SMS.
- **Services started:** Services started by the application.

2.4.3 Hybrid analysis

Finally, it is also possible an approach in which both static and dynamic analysis techniques are combined (all the features represented in Fig. 2.6). The advantages that each type of analysis offers can be used to build stronger detection models in comparison to models selecting a single perspective [YLWX14]. A hybrid analysis poses the most effective approach to follow, however, the computational cost can be considered as high. In many cases, a two steps analysis processes is the most appropriate solution. Thus, a first layer involving static features determines the nature of the sample, but, in those cases in which a categorisation is not reached with a certain degree of precision, a dynamic analysis is performed [MGF+15].

2.5 Machine learning in the Android malware domain

Varied literature has focused on the use of machine learning techniques to implement malware detection and classification mechanisms. This work focuses on the use of supervised models, particularly those under the field of classification. In this line, diverse features, models and algorithms are applied to tackle these two problems. The process in which these models are trained and tested follows different steps. First, it is required to extract a series of features able to describe the behaviour of a specific labelled (as benign or malicious) sample. These features, which are gathered using any of the approaches presented in the previous section, are modelled using different alternatives, such as vectors of binary and counter variables or as graphs.

The next sections present some of the most important research lines in the literature from two different points of view: the malware detection problem, in which a classification task aims to discern between malicious and benign samples and the malware family classification problem, where the family of malicious samples is decided.

2.5.1 Machine learning for Android malware detection

Most of the existing literature focuses on the detection of malicious executables. Moreover, this problem can be faced from two perspectives, related to the static and dynamic analysis approaches. The easy extraction of static features and also the wide description that they provide of the application operation and intentions, are some of the reasons of the important amount of research focused on this line. Information related to the package, code or metadata can be used as input data to train these models. The following paragraphs describe how different features are combined with varied machine learning techniques to develop malware detection tools.

API calls compose one of the most important instruments to provide detailed information of the operations that a suspicious sample could invoke. A significant portion of the existing literature exploits system calls to nurture machine learning classifiers. DroidMat [WMW+12] is a model where API calls are used based on the component to which they are related in the execution. Information related to Intent actions, permissions or Inter-Component Communications (ICC) is also considered. Clustering algorithms allow to improve the modelling of malware behaviours while naive Bayes and k-NN lead the learning process.

DroidMiner [YXG+14] or DroidAPIMiner [ADY13] are other examples of literature proposing API calls as the main descriptive characteristic to train malware detectors. The first one generates Component Behaviour Graphs (CBG) to represent the existing links connecting API resources and permissions with the actions performed. Then, Support Vector Machines (SVMs), naive Bayes, Decision Tree and Random Forest algorithms are trained. Other existing literature also focuses on API calls graphs [GYAR13]. In DroidAPIMiner, special attention is paid to dangerous calls while the ID5, C4.5, k-NN and SVM algorithms are trained.

In DroidSIFT [ZDYZ14], dependency graphs of API calls and similarity metrics to detect zero-day malware enable to train a naive Bayes classifier. In combination with permissions and other system events [ZYZ+18], system calls feed a rotation forest model. The authors prove that this decision tree based clasiffier delivers better results in comparison to SVMs. A different approach called MOCDroid builds a malware detector with an evolutionary approach [MMC17].

Other more complex approach involves the combination of API calls with permissions to train a Multifeature Collaborative Decision Fusion (MCFG) [SAN15] approach where SVMs, Decision Trees, naive Bayes, IBk and JRip form the set of classifiers.

The Android Manifest yields a series of key details to understand the range of action that the application could take. With a definition of **permissions** required, one can draw a general picture of the operation of the suspicious sample and to extract general patterns able to make a distinction between malware and goodware. In general, permissions related features are combined with other characteristics. Support Vector Machines (SVM), Decision tree and Bagging are trained with a dataset of API calls and permissions [PZ13]. With other features such as Intent actions, network data or hardware components, Support Vector Machines are trained in Drebin [ASH+14]. A similar combination of features is employed by DroidSieve [STDA+17], with special attention to obfuscation techniques and using Extra Trees.

Intents also pose a profitable information source to model the behaviour of Android applications in terms of operations defined and invoked in runtime (i.e. to start a background service). This is one of the features which Andro-dympsys [JKW+16] uses in order to perform malware detection with similarity measures and naive Bayes models. An study analyses the effectiveness of Intents for detecting Android malware with Bayesian Networks [FAS+17], showing that they provide a better description mechanisms than permissions.

Existing research also adopts more generic features, such as those related to **metainformation** extracted from the APK. ADROIT [MCM+16] applies Natural Language Processing techniques to the description of the application and a set of machine learning classifiers, such as Random Forest, Bagging or naive Bayes are trained with this information and also details from the Android Manifest. Manilyzer also concentrates on metainformation extracted from the Android Manifest to train k-NN, C4.5, SVM and naive Bayes [FSW14].

Opcodes conforms another approach to perform a deep description to feed a malware detector. In this line, this research studies and evaluates the detection of malware using opcodes [CDLM+15]. A set of classifiers, including Decision Tree, Simple Logistic, naive Bayes, PART and RIDOR learn in parallel with a set of features which include **system commands** [YSM14]. This feature is also valid to train Bayesian models [YSMM13]. Inter-Component Communications (ICC) is other feature which also focuses literature on this topic, for example, training Support Vector Machines [XLD16].

Other recent feature extraction technique is based on **taint analysis**, which allows to discover information leaks able to describe some important malicious patterns. In Android, the FlowDroid [ARF+14] tool allows to perform this kind of analysis by modelling the entire lifecycle of an application. Information flows are used in DeepFlow to train deep learning models [ZJY+17].

More recent approaches build complex architectures of classifiers and features. DroidFusion [YS18] trains a set of classifiers at a low level, including Random Tree, REPTree, J48 and Voted Perceptron but also ensemble classifiers, such as Random Forest, AdaBoost or Random Subspace. Then, four different ranking algorithms allow to decide the combination strategy of these base classifiers, which receive data related to permissions, Intents, API calls, Linux commands and of the presence of files containing new code such as those with extension *.jar* or *.so*. An ensemble of 11 different classifiers was also used aiming to accurately detect Android malware [WLW+18]. In this case, SVM, k-NN, CART, naive Bayes or Random Forest allow to learn from data describing the use of Intents, restricted and suspicious API calls, permissions or

hardware related features.

While previous presented literature leverages static features to build machine learning based Android malware detectors, dynamically extracted information also offers a powerful instrument to build strong classifiers. For instance, Andromaly [SKE+12] collects metrics of the runtime behaviour and feeds Bayesian Networks or a Logistic Regression classifier.

2.5.2 Machine learning for Android malware classification

The classification of malware into families, also known as malware triage, it is also present in varied research due to its importance when mitigating the effects of malware. In general, there can be found family classification methods which follow similar approaches to the ones shown in literature focused on detection of malware. For instance, Dendroid [STTPLB14] implements a text mining process over code structures represented as Control Flow Graphs (CFG) to build a feature space in which a 1-NN classifier learns to discern between malware families. DroidScribe [DSTK+16] opts for Support Vector Machines with dynamically extracted API calls. DroidSieve [STDA+17], focused on malware detection, also aims to identify the malware family.

2.5.3 State-of-the-art machine learning algorithms for Android malware detection and classification

This section summarises the most used machine learning classification algorithms in the literature related to Android malware detection and family classification.

Decision trees

Decision trees are one of the former machine learning methods for regression and classification. They provide a useful mechanism based on a set of splitting rules [JWHT13]. These models make a prediction based on the most common class in the region of the example. Typically, decision trees are generated through consecutive binary splitting while an error function is minimised. One of the strenghts of these models lies in that they allow an easy interpretation and visualisation. In contrast, they have several drawbacks, such as classification problems when presenting data with small changes. An example of decision tree algorithm is ID3 [Qui86], which makes divisions trying to maximise the information gain.

Support Vector Machines

This is a powerful algorithm used for both classification and regression problems. It is based on representing each example in a n-dimensional space, where a hyperplane is created pursuing the best separation between classes [CV95]. For building this hyperplane, a portion of the training data called *support vectors* is employed. These models have been widely used in the literature for building malware detection tools.

Naive Bayes

The naive Bayes classifier is a probabilistic model which fall under the scope of bayesian learning [Mit97]. It is based on the Bayes theorem and pursues to calculate the hypothesis with higher probability of a space defined by training data. These models are characterised by their simplicity but also by their proven ability to deal with varied problems.

K-nearest neighbours

The K-nearest neighbours is a non-parametric classifier, meaning that this model grows in parallel to the size of the training data [Rob14]. Basically, it uses a distance metric to provide a prediction based on the majority class among the K nearest point to the example. While these models have proved to be powerful in varied problems, they are not indicated for high-dimensional spaces.

Deep learning architectures

Deep learning models are an evolution of the neural networks models proposed several decades ago [GBCB16]. They are complex models composed of multiple layers with different functions. This is the case of Convolutional Neural Networks (CNNs), which have shown excellent results in image classification and recognition problems. Deep learning models are currently receiving great attention due their ability to cope with problems of varied nature and complexity.

Ensemble classifiers

Ensemble classifiers are based on a weighted combinations of different models [VM02]. A plethora of algorithms have been proposed following this idea [OM99], such as Bagging or Boosting, which apply resampling processes over training data in order to build better classification mechanisms. They can be combined with different individual estimators, although typically decision trees are used. A particular classifier widely used by malware detection tools is Random Forest, which trains a set of decision trees on different subsets of the data [Rob14].

Chapter 3

Applying machine learning techniques for Android malware detection and classification

"Never let your sense of morals
prevent you from doing what is right."

- Isaac Asimov, *Foundation*

The ability of machine learning to deal with complex and varied problems is beyond question. In particular, machine learning classifiers are powerful methods that, based on a training process where knowledge is acquired from a set of labelled examples, are able to make predictions on instances given in the future [HPK11]. Furthermore, they can be constantly trained with more examples, thus continuously improving their understanding of the problem.

A plethora of algorithms under the scope of machine learning has already been applied to the detection of malicious executables [Mal05]. The basic idea of this combination lies in analysing the source code of a suspicious sample looking for known patterns, revealing the true intentions of the sample. In case of malicious code, aiming to compromise the target computer system, to destroy or remove user's information, to steal private information or even to use the affected device to distribute illegal content.

Malware in the Android domain poses an important challenge since its origin. Attackers have firmly targeted this platform in order to lock user's phones, to steal data, to demand money or to hamper the normal use of the device attacked. The huge number of samples containing a malicious payload implies the use of techniques able to manage large amounts of data in order to avoid these applications to reach the user through market stores. Machine learning provides powerful mechanisms to help in this task, which would be impossible to face using manual procedures.

This section deals with the Android malware detection problem from a machine learning aided detection perspective. More in particular, the paradigm followed throughout all this work revolves around an offline process where the goal is to design and implement instruments that, receiving the executable file of a suspicious sample as input, will take a decision categorising the sample as malware (it has evil intentions) or benignware/goodware (its operation does not incurs

in any damage to the user or his/her device). This model has also been extended to consider the classification of malicious samples into their corresponding malware family.

The ultimate aim of this chapter is to study and evaluate the use of machine learning techniques to detect and classify Android malware. In the different sections of this chapter, the application of these techniques is tested, comparing the different perspectives which can be followed, and providing different remarks which can help to better understand how these models can be successfully applied.

On the following sections, the problem stated is faced from different points of view related to the different malware analysis techniques available, that is to say, by representing instances with static, dynamic and hybrid features. In each of these sections, a representation based on the according type of features is first proposed and described. Then, the use of a particular set of machine learning models is evaluated. The final section focuses on tackling the Android malware classification process.

3.1 Android malware detection through static features

Static features focus large part of the state-of-the-art literature related to Android malware detection. The easy and quick extraction of this kind of features make them suitable to be used to build malware detection tools. In this section, a wide set of machine learning models are leveraged to perform a study on the appropriateness of statically extracted features for the profile of Android malware and goodware.

Independently of the procedure followed to extract static features, they must be processed in order to build vector based behavioural profiles to feed machine learning classifiers. This procedure tries to describe the specific behaviour of each sample in terms of absence or existence of a set of characteristics previously determined. The underlying hypothesis under this assumption is that the existence of a particular subset of characteristics will reveal a behaviour depiction which can be unequivocally attributed to malicious or to harmless intentions. Among those features which can be retrieved performing static extraction, numerical and binary variables can be identified. While the former represent measurable indicators (i.e. the number of times an API call is invoked), the latter denotes the existence of a certain feature (i.e. the declaration of a permission). Thus, a dual counter and binary based representation was followed.

Once extracted the desired set of characteristics, such as permissions requested, API calls invoked or information flows, all this information is processed to build a representative feature vector for each application. This process is shown in Fig. 3.1. On the one hand, M measurable characteristics $C_M = \{c_1, c_2, \ldots, c_M\}$ are assigned a value counting the number of occurrences of that feature within the application. On the other hand, K binary features $D_K = \{d_1, d_2, \ldots, d_K\}$ point the use or the existence of a particular attribute. The bundling of both features compose the static description of the application behaviour.

With the previous representation in mind, it is required to vectorise the resulting features extracted from descriptive datasets of samples in order to train and test new classification models. And this is one of the most crucial points when testing supervised machine learning based tools, to gather a large collection of representative examples, in this case composed of both malware and benignware samples, in order to fairly train and evaluate these tools.

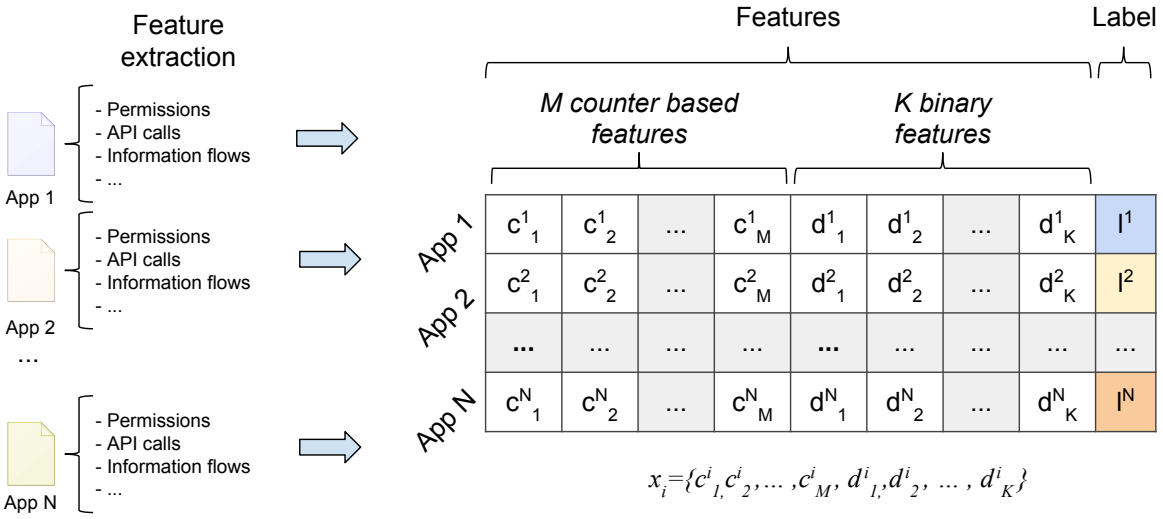


Figure 3.1: Extraction and representation of static information into representative feature vectors.

Currently, there are different datasets which can be used for this purpose, although they present some limitations. For instance, some datasets provide sets of executable files such as the AndroZoo project [ABKT16] or the Android Malware dataset [WLR+17]. Alternatively, some public datasets offer logs of information extracted from samples such as the DroidCat dataset [RF16, RFB17]. Nevertheless, these datasets are not suitable for the above mentioned task, either because they provide raw samples instead of vectors of already extracted features or because the number of instances is insufficient and the selection of features is unsatisfactory.

Based on the reasons underlined in the previous paragraph, it was decided to build a new hybrid, labelled, comprehensive and representative dataset of malware and benignware samples. With this premise, a new automated feature extraction tool named AndroPyTool was implemented, aiming to make it easier the process of extracting varied static and dynamic characteristics from large collections of Android applications. In second place, this tool allowed to generate the OmniDroid dataset. A description of both contributions is provided in Chapter 5.

While AndroPyTool is able to extract static and dynamic features, this section focuses on the former ones. By using Android malware analysis tools such as AndroGuard or by analysing the source code of each sample, AndroPyTool extracts a representative set of characteristics based on state-of-the-art Android malware detection approaches. These features include API calls, main activity name, opcodes, package name, permissions, intent receivers, intents services, intent activities, strings found, system commands or information flows extracted with FlowDroid [GHP+15]. Samples gathered from AndroZoo [ABKT16] and Koodous[1] were scanned by AndroPyTool to conform a dataset of features containing instances from 11,000 benign applications and 11,000 malicious applications[2].

[1] `https://koodous.com/`

[2] Considering as goodware those applications with zero positive detections among all antivirus integrated in VirusTotal.

3.1.1 Malware detection using ensemble classifiers and static features

The main aim of this section is to assess the use of static features when involved in the characterisation of Android malware and goodware samples used to train detection tools. Thus, through the use of several machine learning classifiers, it is studied if this combination is feasible and, if so, to assess its performance when applied to build a machine learning aided Android malware detection tools.

In general, the use of statically extracted characteristics allows to build a meticulous portrayal of the actions that the application can take and that are expected to be invoked once executed. The process previously shown in Fig 3.1 was followed with all the samples retrieved in order to generate a new representation of the OmniDroid dataset. This new descriptive rendering, where each vector outlines the behaviour of a particular sample (i.e. each position depicts the existence or the number of occurrences that a particular feature is involved in the application operation) serves as entry point to the training process of the different machine learning classification algorithms tested. The file containing all these vectors and used in the experiments is publicly available[3].

With the corresponding batch of feature vectors extracted from benign and malicious applications, a set of ensemble classifiers was used. The reasons under the selection of this particular kind of machine learning algorithms lies in their proven performance when integrating diverse classification and learning methods, taking advantage from their joint advantages and overcoming the possible individual drawbacks [MMC16a, YS18]. Furthermore, the use of ensemble classifiers composed of decision tree based estimators brings the best approach to follow. Given the space of features, characterised by independent features without a spatial relation, some of them binary and the rest formed by variables defined within the $\mathbb{Z}^+$ space, a linear approach through decision rules appears to be the most recommendable approach.

The selected ensemble classifiers were run with the *Scikit-learn* library for Python [PVG+11]. They include AdaBoost [HRZZ09], an implementation of Bagging (with Random Forest estimators) which combines different works [Bre99, Bre96], ExtraTrees [GEW06], Gradient Boosting [Fri01], Random Forest [Bre01] (with 100 internal estimators), and a Voting classifier combining Random Forest, K-NN [Alt92] and a simple decision tree classifier, each one with the same weight.

All these ensemble classifiers were tested with the previously mentioned dataset through a 10-folds cross validation. For a better evaluation of the capacity of these algorithms to extract and generalise conclusions, several experiments were conducted in order to judge the individual and joint performance of the different features involved. The results are shown in Table 3.1.

In contrast to what it could be expected, the use of an individual group of features allows to obtain the best results among all the combinations of features tested. In particular, API calls compose the most appropriate representation to build a machine learning classification tool, exhibiting a 89,3% accuracy and precision with a Random Forest classifier. In general, this algorithm obtains the best results overall, independently of the combination of features. It is remarkable the fact that a combination of API calls with other also competitive features, such as opcodes or receivers, which have proved to be powerful at building detection mechanisms [MMC16b], does not lead to better values.

[3] `https://aida.ii.uam.es/datasets/`

Features set	Metric	AdaBoost	Bagging	ExtraTrees	Gradient Boosting	Random Forest	Voting
Activities	**Acc**	0.506 ± 0.002	0.506 ± 0.002	0.506 ± 0.002	0.506 ± 0.002	0.506 ± 0.002	0.506 ± 0.002
	Prec	0.602 ± 0.036	0.602 ± 0.036	0.602 ± 0.036	0.602 ± 0.036	0.602 ± 0.036	0.602 ± 0.036
API calls	**Acc**	0.859 ± 0.008	0.891 ± 0.007	0.89 ± 0.006	0.871 ± 0.009	**0.893 ± 0.006**	0.886 ± 0.006
	Prec	0.859 ± 0.008	0.892 ± 0.007	0.89 ± 0.006	0.871 ± 0.009	**0.893 ± 0.006**	0.887 ± 0.006
API Packages	**Acc**	0.5 ± 0	0.5 ± 0	0.5 ± 0	0.5 ± 0	0.5 ± 0	0.5 ± 0
	Prec	0.25 ± 0	0.25 ± 0	0.25 ± 0	0.25 ± 0	0.25 ± 0	0.25 ± 0
FlowDroid	**Acc**	0.677 ± 0.009	0.706 ± 0.008	0.708 ± 0.008	0.681 ± 0.01	0.708 ± 0.008	0.704 ± 0.009
	Prec	0.72 ± 0.013	0.744 ± 0.009	0.748 ± 0.009	0.723 ± 0.013	0.746 ± 0.009	0.744 ± 0.01
FlowDroid, API calls	**Acc**	0.86 ± 0.01	0.891 ± 0.007	0.889 ± 0.006	0.872 ± 0.007	0.892 ± 0.007	0.886 ± 0.006
	Prec	0.86 ± 0.01	0.892 ± 0.007	0.89 ± 0.006	0.872 ± 0.007	0.892 ± 0.007	0.886 ± 0.006
FlowDroid, API Packages	**Acc**	0.677 ± 0.009	0.708 ± 0.008	0.709 ± 0.008	0.681 ± 0.01	0.707 ± 0.01	0.704 ± 0.008
	Prec	0.72 ± 0.013	0.745 ± 0.009	0.749 ± 0.009	0.722 ± 0.013	0.745 ± 0.011	0.745 ± 0.01
Opcodes	**Acc**	0.833 ± 0.011	0.873 ± 0.01	0.869 ± 0.007	0.846 ± 0.009	0.874 ± 0.012	0.868 ± 0.009
	Prec	0.833 ± 0.011	0.873 ± 0.009	0.869 ± 0.007	0.846 ± 0.009	0.874 ± 0.011	0.868 ± 0.009
Permissions	**Acc**	0.781 ± 0.01	0.824 ± 0.006	0.824 ± 0.006	0.792 ± 0.008	0.825 ± 0.007	0.821 ± 0.008
	Prec	0.781 ± 0.01	0.826 ± 0.006	0.826 ± 0.006	0.792 ± 0.008	0.827 ± 0.006	0.823 ± 0.008
Receivers	**Acc**	0.824 ± 0.005	0.876 ± 0.01	0.877 ± 0.009	0.84 ± 0.005	0.877 ± 0.01	0.875 ± 0.009
	Prec	0.825 ± 0.006	0.876 ± 0.01	0.877 ± 0.009	0.84 ± 0.005	0.877 ± 0.01	0.875 ± 0.009
Receivers, API calls	**Acc**	0.858 ± 0.01	0.889 ± 0.006	0.89 ± 0.008	0.875 ± 0.007	**0.892 ± 0.008**	0.885 ± 0.007
	Prec	0.858 ± 0.01	0.889 ± 0.006	0.891 ± 0.008	0.875 ± 0.007	**0.892 ± 0.008**	0.885 ± 0.007
Receivers, API calls, Opcodes, Permissions	**Acc**	0.862 ± 0.009	0.89 ± 0.008	**0.891 ± 0.008**	0.88 ± 0.008	**0.891 ± 0.007**	0.884 ± 0.008
	Prec	0.862 ± 0.009	0.89 ± 0.008	**0.891 ± 0.008**	0.88 ± 0.008	**0.892 ± 0.007**	0.884 ± 0.008
Receivers, API calls, Opcodes, Permissions, FlowDroid	**Acc**	0.865 ± 0.008	0.889 ± 0.007	**0.892 ± 0.009**	0.879 ± 0.008	**0.891 ± 0.008**	0.883 ± 0.007
	Prec	0.865 ± 0.008	0.89 ± 0.007	**0.893 ± 0.009**	0.88 ± 0.008	**0.892 ± 0.008**	0.883 ± 0.007
Receivers, Services, Activities	**Acc**	0.825 ± 0.005	0.875 ± 0.008	0.877 ± 0.007	0.843 ± 0.008	0.876 ± 0.008	0.876 ± 0.008
	Prec	0.826 ± 0.005	0.875 ± 0.008	0.878 ± 0.007	0.843 ± 0.008	0.876 ± 0.008	0.876 ± 0.008
Receivers, Services, Activities, API calls	**Acc**	0.858 ± 0.01	0.889 ± 0.007	0.888 ± 0.006	0.874 ± 0.008	0.889 ± 0.007	0.884 ± 0.007
	Prec	0.858 ± 0.01	0.889 ± 0.007	0.889 ± 0.006	0.874 ± 0.008	0.89 ± 0.007	0.884 ± 0.007
Services	**Acc**	0.515 ± 0.003	0.516 ± 0.002	0.516 ± 0.002	0.515 ± 0.003	0.516 ± 0.002	0.516 ± 0.003
	Prec	0.749 ± 0.013	0.741 ± 0.015	0.743 ± 0.015	0.75 ± 0.012	0.741 ± 0.015	0.742 ± 0.015
System commands	**Acc**	0.761 ± 0.009	0.827 ± 0.007	0.827 ± 0.007	0.776 ± 0.007	0.826 ± 0.006	0.82 ± 0.008
	Prec	0.763 ± 0.009	0.828 ± 0.007	0.828 ± 0.007	0.777 ± 0.007	0.827 ± 0.006	0.821 ± 0.008

Table 3.1: Results from the different ensembles classifiers used, where different combinations of static features are tested.

This apparently contradictory situation reflects a lack of ability to generalise patterns. Thus, it is possible to relate the use of complementary information with an overfitting effect. This is supported by the fact that the accuracy actually improves when evaluating the training set, from a 99,27% with only API calls to a 99,76% with the most comprehensive set of features, including Receivers, API calls, opcodes, Permissions and information flows. However, the difference found in terms of accuracy in the test set for API calls (89,3% ± 0,6) against the use of the complete set of features (89,1% ± 0,8) is too small to extract more specific conclusions.

Other conclusions can be made after the analysis of these results. In the case of API packages, services or activities, it can be seen how these features do not allow to discern the nature of the sample, since they provide very high level information unable to describe specific behaviours. Information flows, which provide relevant information related to malicious patterns, slightly improve the results of the previous mentioned characteristics, but are far from the best values obtained. For the rest of components, the results remain very similar. Regarding the differences among ensemble classifiers, all of them perform very close. The only method where it can be observed how accuracy and precision slightly decrease is the AdaBoost algorithm.

Finally, some conclusions can be made on the basis of the results obtained. In general, static features allow to reach an appealing almost 90% accuracy and precision. What it is more remarkable is that the description of a single component of the Android platform describing the

use of API calls enables to build the most representative feature space, that is to say, the space where instances of malware and goodware can be better differentiated.

3.2 Android malware detection through dynamic features

Static analysis techniques face an important barrier when the suspicious sample makes use of obfuscation techniques [MMC16c]. For instance, dynamic code loading can make statically extracted features useless, as shown in the analysis of the Jisut family in Chapter 2. In contrast, a dynamic analysis based approach allows to obtain extensive information about the real actions performed while the target sample is executed in a controlled environment. In this scenario, while the sample is executed, a monitoring agent is in charge of capturing each event or action undertaken by the application in runtime. The main strength of this kind of analysis lies in that it unveils patterns which are only disclosed when the sample is executed and when certain capabilities are given. For instance, the malicious payload could only be triggered when the applications can access to Internet. This can be made through the use of reflection or dynamic code loading techniques. However, it is also necessary to comment that existing literature points that the execution inside a virtual machine can be detected [PVA+14]. If so, the application could hide the malicious payload if it is not running in a real device.

In order to test how machine learning technique deal with the Android malware detection problem through dynamic features, the DroidBox tool [Lan15, Lan11] was selected to run the dynamic analysis. For that purpose, the training set was composed by the same set of samples previously used in previous experiments and present in the OmniDroid dataset. This dynamic analysis tool has also been integrated into the AndroPyTool framework. A further description of this tool is presented in Chapter 3.

In summary, DroidBox monitors a wide set of events, including cryptographic operations, network activity, operations over files or the use of SMS services. When the analysis ends (after a predefined time interval), a report is delivered describing all these actions, linked to a timestamp and organised by the category of each event. All this information, however, requires a special preprocessing task, in order to represent data in a format able to describe the application behaviour but also appropriate to feed a machine learning classifier.

The next subsection describes the representation model used to transform the report generated by DroidBox for each sample analysed. Then, the same set of ensemble classifiers involved in the study when using static features is again tested in combination with dynamically extracted features.

3.2.1 Markov chains based representation

In this section, a Markov chain based modelling of the dynamic traces retrieved with DroidBox is presented [MRFC18], in order to later train different machine learning classification algorithms. The different existent Markov models have been successfully used in a plethora of problems, due to their wide range of possibilities. In general terms, they allow to model and predict sequences of symbols or time series, among others [Fin14], and follow the *Markov property*, where future states in a sequence only depend on the current states, and not on past events. To model the

Dynamic traces delivered by DroidBox, a simple model has been selected, the Discrete-Time Markov Chains. It is a stochastic process, where there is a set of states $S = \{S_1, S_2, \ldots, S_N\}$, and a set of transition probabilities $a_{ij}(t)$ between them. Then, it is possible to build a state transition probability matrix where all these transitions are represented. In this matrix, it is possible to find positions with $a_{ij}(t) = 0$, meaning that this particular transition never occurred during the dynamic analysis.

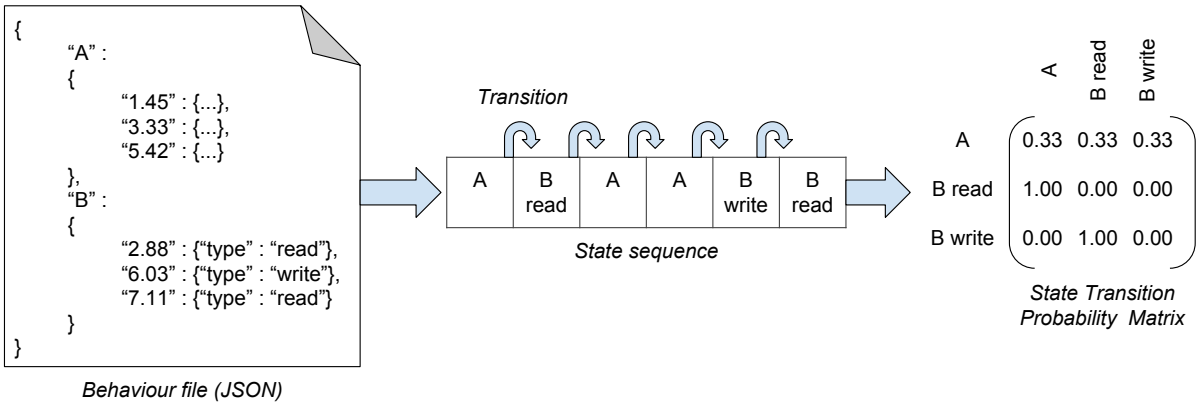


Figure 3.2: Diagram showing the transformation process from the raw information delivered by DroidBox to a state transition probability matrix for each sample.

The process to build this matrix when the DroidBox tool ends its execution is shown in Fig. 3.2. In this moment, a *JSON* file is returned, containing all the events recorded and organised by different sections (file operations, network transmissions, data leaks or services started). Then, it is possible to build a vector where all these events are represented in a single sorted timeline: each event constitutes a state and two sequential states define a transition. Finally, a matrix including all these events as rows and columns and filled with the corresponding transition probabilities is obtained.

The different transition probabilities between states drawn by this matrix is used to represent the behaviour of a particular sample. Following this procedure, with a large set of matrices representing the different behaviour dynamics of a large figure of malicious and benign applications, general behavioural patterns can be extracted in order to build detection and classification mechanisms. When these are machine learning based tools, they need to be trained with labelled vectors of features. These vectors are generated according to the scheme displayed in Fig. 3.3: each matrix is flattened to obtain a representative vector for each APK of size $M \times M$. Besides, a second vector is concatenated to the former, aiming to include other useful information to better describe the behaviour of the sample. In particular, this section is employed to define the frequency of each possible state, with the goal of providing overall information of the importance of the different types of events invoked thorough the execution. Then, the vector size is $M^2 + M$.

Since it is expected that a significant number of transitions will have close to zero probability of existence for most of the samples, a threshold criteria was established. Transitions with less than ϵ non-zero instances performing that transition are removed (a parameter experimentally fixed). Finally, the corresponding label is added at the end of the feature vector of each app, and it can be used to train the desired classification or detection algorithm.

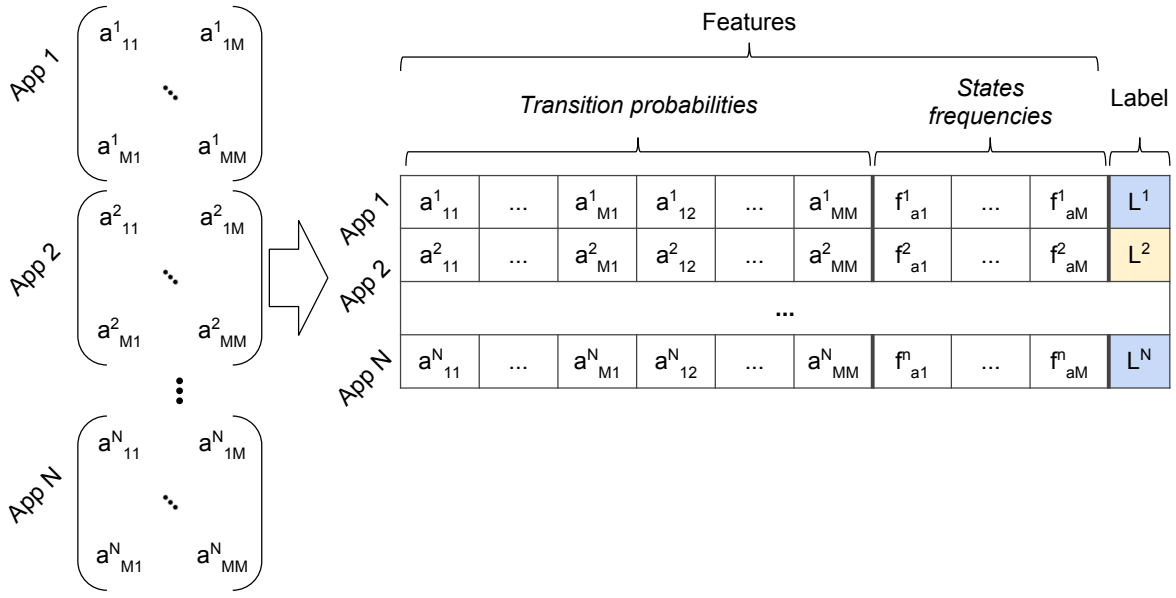


Figure 3.3: Diagram showing the transformation process from each transition probability matrix to a set of vectors describing transition probabilities and state frequencies.

Features set	Metric	AdaBoost	Bagging	ExtraTrees	Gradient Boosting	Random Forest	Voting
Transitions	**Acc**	0.731 ± 0.01	0.776 ± 0.01	0.775 ± 0.012	0.741 ± 0.007	0.775 ± 0.009	0.763 ± 0.009
Transitions	**Prec**	0.731 ± 0.01	0.777 ± 0.01	0.776 ± 0.012	0.741 ± 0.007	0.775 ± 0.009	0.764 ± 0.009
Frequencies	**Acc**	0.739 ± 0.009	0.78 ± 0.012	0.774 ± 0.01	0.743 ± 0.009	0.778 ± 0.011	0.768 ± 0.008
Frequencies	**Prec**	0.74 ± 0.009	0.78 ± 0.012	0.774 ± 0.01	0.743 ± 0.008	0.778 ± 0.011	0.769 ± 0.008
Combination	**Acc**	0.742 ± 0.009	**0.786 ± 0.007**	0.779 ± 0.007	0.751 ± 0.006	**0.785 ± 0.006**	0.771 ± 0.011
Combination	**Prec**	0.743 ± 0.009	**0.786 ± 0.008**	0.78 ± 0.007	0.751 ± 0.006	**0.785 ± 0.006**	0.772 ± 0.011

Table 3.2: Results of the different ensembles classifiers used with different combinations of dynamic features.

3.2.2 Malware detection using ensemble classifiers and dynamic features

The previously presented Markov chains based representation allows to transform the raw sequences of dynamic events extracted in runtime into descriptive information, which characterise the malicious or harmless behaviours of a given set of samples. In this line, while the analysis obtained with DroidBox comprise large sequences of events, many of them showing useless details (i.e. the access to files in specific temporal paths), the transformation process conducted translates these data into organised and descriptive information. The goal of this new representation is to arise behavioural patterns which can help to distinguish between malicious and benign samples.

The dynamic analysis logs provided in the OmniDroid dataset for 11,000 benign and 11,000 malicious samples were transformed in order to obtain information in the form of transition probabilities and state frequencies. For these experiments, the ϵ parameter, which defines the minimum number of samples performing a transition to keep that transition, was configured to remove all transitions in which all instances define zero transition probability $a_{ij}(t) = 0$ with the goal of reducing the search space. In case of events including a path as parameter, a maximum of two levels of depth was allowed.

The experiments were performed using the same pool of ensemble classifiers involved in the experiments with static features. In this scenario, the use of transition probabilities, state frequencies and a combination of them were tested. Results are shown in Table 3.2 in terms of accuracy and precision after a 10-fold cross validation process. When comparing the use of transition probabilities against state frequencies, it can be seen that the second ones arise a small improvement, from 77,6% ± 1 to 78% ± 1.2 using a Bagging classifier. A more significant improvement can be observed when combining both features, reaching 78.6% ± 0.7 in accuracy. Either in the case of individual features or in the combination of both, Random Forest achieves almost similar results.

In comparison with the results achieved in the experiments with static features, which were run in the same conditions (same ensemble classifiers and same dataset), it is remarkable that dynamically extracted features do not allow to reach the same levels of precision and accuracy. From 89,3% using static features to 78,6% accuracy with dynamic features. This considerable difference, of by more than 10%, leads to consider that static features are a more powerful mechanism. Different conclusions can be made at this point. The unexpected worse behaviour of dynamic analysis can be attributed to the analysis tool employed, which could not allow to obtain sufficiently detailed information. Conversely, the use of a Markov chains based representation could result in loosing important details able to make a difference. More in general, although the use of dynamic information allows to model real behavioural data, its scope could be reduced in comparison to static information.

3.3 Android malware detection through hybrid features

On the basis of the two detection models proposed in previous sections, one through the use of static features and the second one using a representation of the dynamic behaviours, in this section it is proposed a new approach as a fusion of both. Aiming to build a more in-depth description of the specific behaviour of each sample and to generate stronger detection models, the approach proposed takes advantage from the combination of static and dynamic features.

The new model combines the ensemble classification model which showed the best performance for each type of features. Then, each one is trained and focuses on the according set of features, either static or dynamic. Based on the results previously provided, a Random Forest classifier is in charge of categorising the suspicious sample taking as input the static description of its behaviour. On the other hand, a Bagging classifier returns a category based on the set of dynamic features.

The fusion of both ensemble based methods is made through a voting classifier, as shown in the diagram provided in Fig. 3.4. Each classifier contributes to the final categorisation with a predefined weight. In order to select the best combination of $\mathbf{W_{RF}}$ and $\mathbf{W_{BG}}$, a grid search was used. The results are provided in Table 3.3 for the different combinations of static and dynamic features after a 10-fold cross validation.

The results evidence again that a combination of information related to transition probabilities and state frequencies allows to reach the best values in terms of both accuracy and precision. Regarding the best pair of weights for each classifier joining the voting scheme, the best configuration proved to be $\mathbf{W_{RF} = 0.7}$ and $\mathbf{W_{BG} = 0.3}$. The inclusion of dynamic information resulted in a slight improvement, reaching 89,7% accuracy and precision.

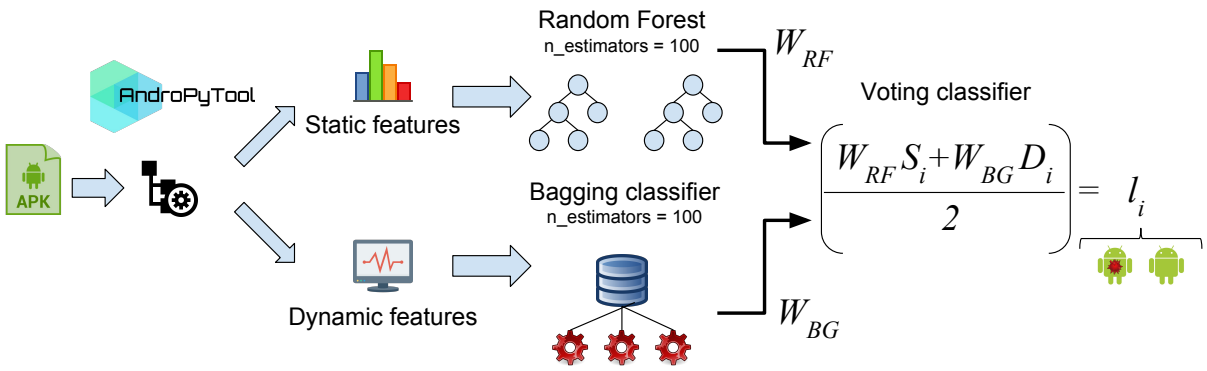


Figure 3.4: Scheme of the fusion approach proposed to combine both static and dynamic features through a voting classifier.

$W_{RandomForest}$	$W_{Bagging}$	Metric	Transitions	Frequencies	Combination
0.1	**0.9**	**Acc**	0.812 ± 0.01	0.81 ± 0.01	0.815 ± 0.011
		Prec	0.813 ± 0.01	0.81 ± 0.01	0.816 ± 0.011
0.2	**0.8**	**Acc**	0.834 ± 0.01	0.83 ± 0.007	0.837 ± 0.009
		Prec	0.835 ± 0.01	0.83 ± 0.007	0.838 ± 0.009
0.3	**0.7**	**Acc**	0.86 ± 0.008	0.854 ± 0.008	0.861 ± 0.007
		Prec	0.86 ± 0.008	0.855 ± 0.008	0.862 ± 0.007
0.4	**0.6**	**Acc**	0.879 ± 0.006	0.873 ± 0.007	0.88 ± 0.007
		Prec	0.879 ± 0.006	0.873 ± 0.007	0.88 ± 0.007
0.5	**0.5**	**Acc**	0.891 ± 0.007	0.887 ± 0.008	0.892 ± 0.007
		Prec	0.891 ± 0.006	0.887 ± 0.007	0.892 ± 0.007
0.6	**0.4**	**Acc**	0.896 ± 0.01	0.895 ± 0.008	0.894 ± 0.01
		Prec	0.896 ± 0.01	0.896 ± 0.008	0.894 ± 0.01
0.7	**0.3**	**Acc**	0.895 ± 0.008	0.896 ± 0.008	**0.897 ± 0.008**
		Prec	0.895 ± 0.008	0.896 ± 0.008	**0.897 ± 0.007**
0.8	**0.2**	**Acc**	0.895 ± 0.008	0.896 ± 0.008	0.895 ± 0.007
		Prec	0.895 ± 0.008	0.896 ± 0.008	0.896 ± 0.007
0.9	**0.1**	**Acc**	0.893 ± 0.008	0.893 ± 0.008	0.894 ± 0.006
		Prec	0.893 ± 0.008	0.894 ± 0.008	0.894 ± 0.006

Table 3.3: Results of the different ensembles classifiers used with different combinations of static and dynamic features.

3.4 Android malware classification

Early detection conforms an essential step in order to prevent the target system to be attacked or breached. In the Android environment, to detect Android malware becomes a very important process in order to avoid the final user to get infected. However, this is a complex and very expensive task. The huge amount of new malware instances found every day makes impossible to detect every threat and, in some cases, they manage to surpass malware detection mechanisms. At this stage, and in order to mitigate the possible damages caused, it is important to identify the kind of threat, evaluating the scope and possible countermeasures to take. The malware triage process focuses on this task [CRTE13], and it is related to the concept of malware family classification. By allocating pieces of malware into families, it is possible to extract common patterns among set of similar samples, thus deploying countermeasures which can be useful when dealing with varied attacks.

Android malware families provide an organised view of the different threats, intentions and damages they can cause but also information related to their origin and evolution. Thus, machine

Family	No. samples	Family	No. samples	Family	No. samples	Family	No. samples
Adrd	79	FakeDoc	132	Glodream	68	MobileTx	68
BaseBridge	311	FakeInstaller	904	Hamob	26	Opfake	597
Boxer	25	FakeRun	60	Iconosys	135	Plankton	478
DroidDream	77	Gappusin	46	Imlog	42	SMSreg	38
DroidKungFu	658	Geinimi	80	Jifake	28	SendPay	58
ExploitLinuxLotoor	67	GinMaster	334	Kmin	95	Yzhc	36

Table 3.4: Number of samples by malware family extracted from the Drebin dataset after the two filtering criteria applied to perform the experiments.

learning can help to build family classifiers able to allocate malware samples into their respective malware family. Once deployed, it can help not only to mitigate the effects of a malware, but also to prevent, understand or detect new variants and shapes of threats.

In this section it is studied the use of machine learning classification algorithms when facing the Android malware classification problem. With the goal of studying the performance of these algorithms, the Drebin dataset [ASH+14] was selected to perform the experiments. It is a collection of 5,560 malicious applications from 179 different malware families gathered from August 2010 to October 2012. Two criteria were fixed in order to obtain the finally used dataset. On the one hand, only those families containing more than 20 samples are kept, in order to ensure that each family is properly represented in the training and test partitions. On the other hand, invalid applications are removed by using the Androguard tool [DG13]. After applying these two filters, a total of 4,442 samples of 24 different malware families were obtained. A description of the final aspect of the data in terms of number of instances per family is shown in Table 3.4.

The number of samples by family evidences a significant imbalance between families, a fact which must be taken into account when facing a machine learning task, since it can provoke an important bias. Some of the most well-known families are included in the dataset. For instance, *BaseBridge* is a Trojan trying to send premium-rate SMS messages to certain numbers. It pursues root privileges in order to deploy a hidden file named *SMSApp.apk* [DHQ+14].

3.4.1 Malware classification using deep learning models

Deep learning models were also tested in this work applied to the Android malware families classification scenario [MRFC18]. Following the Markov chain based representation introduced in Section 3.2.1, categorised samples of varied Android malware families allow to train and test different deep learning architectures. For this purpose, the Drebin dataset [ASH+14] already depicted is used.

In order to extract the dynamic traces of this set of malware samples, the DroidBox tool [Lan15] was launched for each sample to perform a 300 seconds dynamic analysis[4]. Once retrieved the analysis for each sample, each one is processed in order to unify similar events. For instance, a write operation includes a path to the file to be modified. Given that this parameter can take a large number of non significant values, the path is removed and coincident states are consolidated. Besides, the hyperparameter ϵ is set to 10, meaning that each individual transition must be present in at least ϵ samples.

[4]In most of the samples, this time was adequate to capture most of the actions that this kind of analysis can monitor

Deep Learning model	Range of parameters
Fully connected + Dropout	No. Neurons = {50, 100, 300, 500} No. Layers = {2,3,4,6}
CNN	No. Filters = {10, 30, 50} Filter length = {5, 10, 15} Pooling size = {2, 5, 10}
RNN	No. units = {2, 5, 10, 20}
LSTM	No. units = {2, 6, 10}

Table 3.5: Relation of the different range of values used for each parameter in the experiments involving deep learning techniques for Android malware classification.

All the resulting dynamic traces were used to train different deep learning architectures. The Keras framework [C+15] was used in combination with Tensorflow [AAB+15] as backend library. Deep Neural Networks (DNNs) in combination with Dropout layers (aimed at reducing overfitting), Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs) and Long Short-Term Memory Neural Networks (LSTM) were used. These models were tested using a bounded range of values as shown in Table. 3.5. The election the hyperparameters of deep learning architectures was addressed in this dissertation through a heuristic search guided by an evolutionary algorithm [MLCFH+17, MFHNC17]. Other parameters were fixed experimentally: the optimiser was fixed to *Adam* and *sigmoid* was selected as activation function.

In order to test the different architectures, 70% of the data was used for training (10% of this slice allows to monitor the learning curve) and the remaining 30% for testing purposes.

The results obtained are shown in Table. 3.6 presenting five different experiments involving different combinations of features, including the use of transition probabilities and state frequencies when they are used independently, or when a combination of both is used. Furthermore, state frequencies have also been grouped by *superstates*, which include all the events related to a specific category according to DroidBox. In first place, it can be seen that grouping events by superstates leads to non representative information, reaching low accuracy rates.

When comparing transition probabilities and state frequencies, the former allow to achieve better results as expected. The best overall results involve the combination of both features, thus meaning that the inclusion of information related to the importance of each state allows to complement transitions based information. In terms of the different architectures tested, RNN and LSTM networks are not able to produce a proper family space discrimination, since they do not contain a spatial or temporal structure (the Markov chains based representation in matrices results in independent variables). The best results were obtained with a classic fully connected architecture, where 77,8% accuracy is reached.

3.4.2 Malware classification using classic machine learning methods

The problem faced in the previous section has also been studied from the use of classic machine learning methods. In this line, several classifiers widely included in literature related to malware detection have been tested: Random Forest (including 100 internal estimators), Decision Trees, Bagging classifier composed of Random Forest estimators, k-Nearest Neighbours and Support

Experiment	Metric	Deep learning architecture			
		CNN	**Fully connected + Dropout**	**LSTM**	**RNN**
Experiment 1: Transition probabilities	Accuracy	0.76	0.773	0.204	0.204
	F1	0.755	0.767	0.069	0.069
	Precision	0.757	0.769	0.041	0.041
	Recall	0.76	0.773	0.204	0.204
Experiment 2: State frequencies	Accuracy	0.752	0.698	0.204	0.204
	F1	0.739	0.672	0.069	0.069
	Precision	0.744	0.699	0.041	0.041
	Recall	0.752	0.698	0.204	0.204
Experiment 3: State frequencies grouped	Accuracy	0.445	0.512	0.204	0.204
	F1	0.39	0.471	0.069	0.069
	Precision	0.403	0.502	0.041	0.041
	Recall	0.445	0.512	0.204	0.204
Experiment 4: Transition probabilities & state frequencies	Accuracy	0.768	**0.778**	0.204	0.204
	F1	0.764	**0.768**	0.069	0.069
	Precision	0.766	**0.768**	0.041	0.041
	Recall	0.768	**0.778**	0.204	0.204
Experiment 5: Transition probabilities & state frequencies grouped	Accuracy	0.762	0.771	0.204	0.204
	F1	0.758	0.757	0.069	0.069
	Precision	0.761	0.755	0.041	0.041
	Recall	0.762	0.771	0.204	0.204

Table 3.6: Results obtained in the classification of Android malware families with deep learning architectures using different combinations of dynamic features.

Vector Machines (with linear, radial based and sigmoid kernel functions). All the experiments for this approach were run using the *Scikit-learn* library for Python and the average from 10 different executions was obtained.

The results, following the same scheme of previous experiments are shown in Table 3.7. Although the numbers show similar trends when the different experiments are compared, in general lines the new series of algorithms allow to reach better results. A superstates based grouping proves to produce worse values. In contrast to the deep architectures previously tested and when applied to individual features (Experiments 1 and 2), the use of state frequencies allow to build a space where families are better differentiated. In terms of models, SVMs are only competitive with a linear kernel. Finally, the best results are obtained with a combination of state frequencies and state transition probabilities, improving previous results and reaching 81,8% accuracy.

The observable differences between deep learning models and classic models such as ensemble classifiers of decision trees lead to conclude that the second ones draw a better separation among families in the feature space, thus allowing to build more powerful methods. However, a further study of the possibilities to develop a classification tool in this scenario can be screened by paying special attention to the high imbalance between the representative instances of each family. For this reason, the next subsection focuses on exploring the effects of the application of different imbalanced learning algorithms.

Experiment	Metric	ML algorithm						
		Bagging	**Decision tree**	**k-NN**	**Random Forest**	**SVM linear**	**SVM RBF**	**SVM sigmoid**
Experiment 1: Transition probabilities	Accuracy	0.799	0.679	0.739	0.801	0.748	0.512	0.426
	F1	0.779	0.678	0.728	0.784	0.728	0.462	0.359
	Precision	0.787	0.681	0.729	0.792	0.738	0.529	0.516
	Recall	0.799	0.679	0.739	0.801	0.748	0.512	0.426
Experiment 2: State frequencies	Accuracy	0.805	0.741	0.708	0.813	0.454	0.288	0.257
	F1	0.793	0.739	0.697	**0.803**	0.414	0.194	0.148
	Precision	0.797	0.74	0.697	0.804	0.614	0.412	0.178
	Recall	0.805	0.741	0.708	0.813	0.454	0.288	0.257
Experiment 3: State frequencies grouped	Accuracy	0.666	0.626	0.563	0.667	0.269	0.23	0.221
	F1	0.654	0.62	0.541	0.659	0.156	0.114	0.098
	Precision	0.655	0.622	0.557	0.658	0.146	0.162	0.137
	Recall	0.666	0.626	0.563	0.667	0.269	0.23	0.221
Experiment 4: Transition probabilities & state frequencies	Accuracy	0.811	0.723	0.719	**0.818**	0.752	0.489	0.333
	F1	0.792	0.721	0.709	0.802	0.734	0.439	0.255
	Precision	0.801	0.723	0.712	**0.807**	0.745	0.53	0.431
	Recall	0.811	0.723	0.719	**0.818**	0.752	0.489	0.333
Experiment 5: Transition probabilities & state frequencies grouped	Accuracy	0.808	0.713	0.742	0.815	0.748	0.509	0.422
	F1	0.789	0.71	0.729	0.799	0.728	0.458	0.355
	Precision	0.797	0.71	0.734	0.806	0.736	0.529	0.509
	Recall	0.808	0.713	0.742	0.815	0.748	0.509	0.422

Table 3.7: Results obtained in the classification of Android malware families with machine learning classifiers using different combinations of dynamic features are used.

Imbalanced Learning Algorithm	Experiment 1	Experiment 2	Experiment 3	Experiment 4	Experiment 5
ADASYN	0.808	0.803	0.654	0.816	0.806
AllKNN	0.76	0.757	0.6	0.774	0.768
Cluster Centroids	0.77	0.703	0.593	0.788	0.781
Condensed Nearest Neighbour	0.611	0.617	0.453	0.643	0.609
Edited Nearest Neighbours	0.71	0.691	0.544	0.723	0.714
Instance Hardness Threshold	0.74	0.75	0.637	0.761	0.741
Near Miss	0.45	0.454	0.334	0.404	0.459
One Sided Selection	0.711	0.717	0.593	0.742	0.698
Random Over Sampler	0.799	0.796	0.655	0.806	0.804
Random Under Sampler	0.704	0.718	0.553	0.724	0.725
Repeated Edited Nearest Neighbours	0.708	0.683	0.516	0.722	0.715
SMOTE	0.807	0.805	0.658	0.815	0.813
SMOTE borderline 1	0.804	0.797	0.648	0.808	0.809
SMOTE borderline 2	0.797	0.799	0.647	0.815	0.806
SMOTE svm	0.801	0.8	0.654	0.813	0.812
SMOTE ENN	0.808	0.792	0.654	0.816	0.812
SMOTE Tomek	0.809	0.805	0.66	**0.818**	0.813
Tomek Links	0.784	0.796	0.656	0.796	0.803

Table 3.8: Results obtained in the classification of Android malware families with machine learning classifiers in combination with imbalanced learning algorithms using different combinations of dynamic features are used. Values are shown in terms of precision.

3.4.3 Malware classification using learning algorithms for imbalanced data

The absence of balanced datasets in many domains evidences the need for applying specialised methods able to mitigate this effect. Imbalanced learning algorithms encompass a powerful instrument when tackling this kind of problems. Geared towards the same Android malware family classification scenario previously examined, a set of state-of-the-art techniques able to handle imbalanced data have been tested. These techniques include *undersampling* based strategies, where samples belonging to overwhelmed labels are randomly discarded. On the other hand, *oversampling* techniques duplicate samples of under-represented families in order to reach an equilibrium. A total of 18 different techniques were tested in combination with the best classic

classifier previously found, which as shown in Table 3.7, is the Random Forest classifier.

Table 3.8 shows the results in terms of precision after applying a wide set of imbalance learning algorithms. The best methods are those based on SMOTE [CBHK02] in its different variations. In particular, the combination SMOTE + Tomek with a Random Forest classifier arises the best result (81,8% precision), slightly improving the performance of a single Random Forest without applying a resampling technique. This result is obtained in Experiment 4, the same previously proven to provide the best features combination: state frequencies and transition probabilities. Regarding the resampling procedure SMOTE + Tomek, it performs an over-sampling process to create more representative samples of under-represented families.

Chapter 4

Adversarial machine learning in the Android malware domain

"To succeed, planning alone is insufficient.
One must improvise as well."

- Isaac Asimov, *Foundation*

Machine learning classification algorithms have been deployed in varied domains, including those related to cybersecurity issues, performing efficiently diverse tasks. Notwithstanding, these methods are vulnerable to attacks where their effectiveness can be compromised. When used to build Android malware detection tools, as shown in the previous chapter, an attack could lead the classifier to produce flawed outputs. This chapter presents an adversarial learning model which seeks to test the resilience of classification algorithms in the Android malware domain against targeted incursions.

The attack postulated is focused on the triage process of Android malicious applications in which machine learning classifiers are used. While malware detectors play a fundamental role in order to prevent the attack to be enforced, to classify a malicious software into the according malware family poses also a major task. In this respect, a correct labelling can help to pay attention to those families that present the greatest risk [CRTE13]. Besides, the model proposed is easily extended to malware detection approaches. The different experiments prove that the attack can successfully disrupt the classification process, causing samples to be assigned to a wrong family. Consequently, a countermeasure is formulated to deal with this kind of attacks.

4.1 Attack definition

In general terms, the attack proposed, called IagoDroid [CMM+18], tries to insert a series of modifications into a malicious sample whose categorisation by the target classifier is known. These changes are aimed at producing a classification output different from the original one. The goal is to reach a family which presents some advantages, such as one to which less attention is paid.

The scenario where the attack is implemented is as follows. The attacker can access the machine learning based classifier and knows the set of features employed by the classifier, however,

he/she has not details regarding its implementation (the classification algorithm), since it is not a feasible assumption. This classifier takes as input the feature vector of a particular sample and issues a probability of belonging to a specific family. In this model, the attacker is able to extract the same selection of features that the classifier employs, and to submit vectors without any restriction to obtain the corresponding labels and probabilities. Following the same procedure, the attacker is free to modify these vectors and upload them in order to obtain the classification result.

According to a state-of-the-art taxonomy defining different kind of attacks against machine learning [BNS+06], the attack designed is exploratory, due to its focus on provoking misclassifications and not in the training process itself; it is a targeted attack, since the attack can target specific families and it is not an attack seeking to compromise the classifier availability.

Because the classification procedure only receives the feature vector associated with the sample to be classified, once extracted this vector it is possible to easily apply different modifications, which means decreasing or increasing the different vector positions. Thus, the attacker can manually modify these values with the goal of generating a new vector able to deceive the classifier and provoking a misclassification. However, in order to keep the semantic intact, these changes are limited to incremental operations, since decreasing one position would lead to remove a functionality (i.e. an API call).

Furthermore, this process is not supposed to be trivial: there is a large number of features and combinations of them which can be manipulated. For this reason, a heuristic search process was chosen to find new vectors able to reflect a family change. A similar combination has already been implemented for evading PDF malware classifiers [XQE16]. In case of the adversarial learning in the Android domain, researchers disrupt Deep Neural Networks models [GPM+16]. Meng et al. [MXM+16] also use genetic algorithms to auditing anti-malware tools.

In short, the attack decomposes in three steps:

1. Extraction of the feature vector of a given APK whose family is to be camouflaged.
2. To conduct a heuristic search process to generate new feature vectors which keep the semantic intact but that cause a different labelling.
3. Decompressing and disassembling the APK to address the changes pointed by the selected individual and repacking again the sample.

4.2 Attack implementation: IagoDroid

IagoDroid represents an attack whose main component is a heuristic search process guided by a genetic algorithm. In this process, individuals shape candidate feature vectors which evolve with the objective of producing a fictitious family change. The genetic search can be geared towards two different directions: to induce a random family change, different from the original one, or targeting a specific family. In both cases, the genetic operators are restricted in order to introduce incremental modifications which do not imply semantic variations. When finally deploying these changes into the sample, they will be introduced into unreachable sections of code (i.e. a new API call), covered by opaque predicates.

4.2.1 Attack formalisation

Assuming a dataset of n malicious samples belonging to different malware families, each one is represented by its feature vector. Then, the corresponding set of n feature vectors is defined by:

$$X = \{x_1, x_2, ..., x_n\}, \tag{4.1}$$

Given a set of k different features, each sample x_i will be represented by:

$$x_i = \{x_i^1, x_i^2, x_i^3, \ldots, x_i^k\} \tag{4.2}$$

Then, it is possible to represent the target classifier as the function $C(x_i)$, which receives as input a feature vector and deliver the label y_j with the highest probability, noted by $p(y_j)$:

$$C(x_i) = (p(y_j), y_j) \tag{4.3}$$

The attack to be performed against the target classifier consists on a search process which, starting from a feature vector x_i, pursues a vector x_i' whose classification differs from the original family y_j:

$$IagoDroid(x_i, y_j) = x_i',\ C(x_i') = y_j',\ y_j' \neq y_j,\ x_i + \Delta = x_i' \tag{4.4}$$

The new modified vector emerges from the combination of the original vector x_i and a vector of change Δ, so that $x_i + \Delta = x_i'$.

4.2.2 Target classifier

In order to test the proposed attack, a search was made for state-of-the-art Android malware classification methods. Table 4.1 shows the different proposals found, indicating the use of some of the most important descriptive features, if they are tested for malware family classification and if the code is available. Only the source code of three classifiers is publicly available and of these, RevealDroid[1] includes the wider set of features: API calls, intent actions and information flows.

The learning module of RevealDroid is composed by a machine learning classification algorithm. The authors of this tool test two different methods: a C4.5 classifier [Qui14] and the 1-Nearest Neighbour algorithm [Rob14]. Regarding the training process, feature vectors representing the three kind of characteristics shown above are used.

[1] `https://bitbucket.org/joshuaga/revealdroid`

Classifier	Code structures	Permissions	Api Calls	Intent-actions	Flow analysis	Tested for family classification	Freely available to download
RevealDroid [GHP+15]	✗	✗	✓	✓	✓	✓	✓
DroidSIFT [ZDYZ14]	✗	✓	✓	✓	✓	✗	✗
Dendroid [STTPLB14]	✓	✗	✗	✗	✗	✓	✓
Drebin [ASH+14]	✗	✓	✓	✓	✗	✓	✗
DroidMiner [YXG+14]	✗	✗	✓	✓	✗	✓	✗
DroidAPIMiner [ADY13]	✗	✗	✓	✗	✗	✗	✗
VILO [LWMS13]	✓	✗	✗	✗	✗	✓	✗
DroidLegacy [DNL14]	✗	✗	✓	✗	✗	✓	✓
MAST [CRTE13]	✓	✓	✗	✓	✗	✗	✗

Table 4.1: State-of-the-art android malware classifiers based on machine learning algorithms comparison.

4.2.3 Genetic search

The search process for new vectors able to deceive the classifier is guided by a genetic algorithm. Individuals taking part of this evolutionary process reflect a modified feature vector of the sample to be camouflaged. The variations applied to each gene or feature are limited to incremental values which cannot exceed a certain maximum threshold MT. Such a restriction allows to ensure that the semantic remains intact.

Four different operators participate in the genetic algorithm. An elitist **selection** operator picks the n best individuals, which will be sent to the next generation. A standard tournament operator works as the **reproduction** function. An uniform **crossover** operator and a **mutation** operator which randomly modifies certain positions in the individual are also used.

The evaluation of the individuals is performed through a fitness function aiming to produce the family change, that is to say, to move the vector away from the original classification. Based on a feature vector and its original real label, the fitness function seeks to decrease the probability of classifying the sample to the original family. The fitness function can be formulated as:

$$f(x_i, y_j) = \begin{cases} 1 - p(y_j) & if(p(y_j), y_j) = C(x_i) \\ 1 & otherwise \end{cases} \tag{4.5}$$

Note that if the classification output already differs from the real label (it is a misclassified example), there is no need to modify the vector, as described by the second case in the previous equation.

As a direct extension of the fitness function shown in Equation 4.5, it is possible to target specific families. Then, the function seeks to maximise the probability of allocation of the vector to a specific provided target family. This second function is as follows:

$$f'(x_i, y_k) = \begin{cases} p(y_k) & ifC(x_i) \neq (p(y_k), y_k) \\ 1 & otherwise \end{cases} \tag{4.6}$$

When the target family corresponds to the classification result (a misclassified example which

actually belongs to a different origin family), the function gives the maximum rating to the individual.

4.2.4 Implementation of the attack

The first step in order to conduct this attack is to extract the range of features used by the target classifier, which are known by the attacker. To this purpose, different tools can be considered. In the current scenario, the well known AndroGuard [DG13] tool is employed to extract API calls and intent actions, while FlowDroid is run in order to retrieve a listing of the information flows found in the different samples.

Once the features are extracted from the malware samples whose family is to be camouflaged, and the corresponding feature vectors built, it is possible to perform the evolutionary process. When a valid individual is obtained, it will indicate the new feature vector that the original one has to migrate to. For that purpose, a set of functionalities have to be added.

In order to integrate these modifications, it is necesary to decompress the APK and to disassemble the Dalvik Executable (DEX) files. The `smali` and `backsmali` [Gru18] tools allow to assembly and to disassembly DEX code, a process which allows to translate the bytecode to human readable smali code. Here, it is possible to modify the instructions and to introduce the different modifications within opaque predicates. When these changes involve the addition of new information flows, it is possible to use sections of code related to the different callbacks managed by the Android operating system.

4.3 Experimentation

A series of experiments were performed to test the feasibility of the attack. The Drebin dataset [ASH+14], already described in the previous chapter, was used. Due to the large fluctuation in the number of representative examples per family (some of them represented by a single sample), a minimum threshold was fixed. This parameter was set to 10, so families with a lower number of instances are not considered. By using this parameter, the number of families decreased to 54 from the original 179 groups.

From this set of samples, API calls and intents were extracted with Androguard. In the case of information flows, FlowDroid was run setting its parameters to maximise effectiveness and depth of the analysis. However, due to the large amount of time and resources needed by this tool to perform, it was not possible to analyse the whole dataset. Thus, the final dataset used in the experiments is composed of 1,919 samples from 29 different malware families.

In order to replicate RevealDroid, the RWeka package for R was used to train a C4.5 classifier. On this point, the dataset was splitted into two parts: 2/3 to train and validate the algorithm through a 10-fold cross validation and the remaining 1/3 was used to test purposes. After 50 executions, an average of 88% accuracy was reached in the test set. A summary of the different parameters involved in the genetic search is shown in Table 4.2.

Parameter	Value	Parameter	Value
Mutation probability	0.1	Elitism	3
Crossover probability	0.8	Max transformations per allele	1
Population size	50	Transformation probability per allele	0.6 - 1
Max generations	20		

Table 4.2: Parametrisation of the genetic algorithm for the different experiments performed.

4.3.1 Evading the correct family labelling

In order to test the ability of the genetic search to discover new vectors disguising the original family, experiments for 10 samples belonging to each family were performed. In total, the heuristic search was conducted for 290 different samples, seeking to obtain new feature vectors classified as a different family. The fitness function used in this scenario is the one proposed in Equation 4.5.

Family	First Sol.	Avg. Conv.	Avg. Mod.	Feature
Plankton	1	3.3	1.0	ACTION_INPUT_METHOD_CHANGED (0.7)
GinMaster	1	3.7	1.0	SMS_MMS (0.6)
Kmin	1	4.3	1.0	ACTION_USER_PRESENT (0.6)
Glodream	1	4.7	0.8	ACTION_INPUT_METHOD_CHANGED (0.4)
BaseBridge	Inf	Inf	-	-
Nyleaker	1	3.6	1.0	NETWORK__LOG (0.4)
Gappusin	1	3.4	0.9	ACTION_INPUT_METHOD_CHANGED (0.3)
Geinimi	1	3.9	1.0	NETWORK_INFORMATION (0.5)
Imlog	1	4.7	1.2	ACTION_INPUT_METHOD_CHANGED (0.7)
DroidKungFu	1	7.2	0.7	IPC__NETWORK (0.2)
Iconosys	1	3.5	1.1	NETWORK__LOG (0.3)
Adrd	1	3.6	0.8	ACTION_INPUT_METHOD_CHANGED (0.5)
DroidDream	1	4.1	0.8	ACTION_INPUT_METHOD_CHANGED (0.4)
Dougalek	1	3.5	1.0	ACTION_INPUT_METHOD_CHANGED (0.4)
MobileTx	1	3.2	1.0	FILE (0.5)
FakeInstaller	1	3.5	1.0	ACTION_INPUT_METHOD_CHANGED (0.5)
ExploitLinuxLotoor	1	2.1	0.8	ACTION_INPUT_METHOD_CHANGED (0.4)
Steek	1	3.9	1.0	ACTION_USER_PRESENT (0.4)
Opfake	1	4.8	0.9	ACTION_INPUT_METHOD_CHANGED (0.5)
Nandrobox	1	3.2	1.0	ACTION_INPUT_METHOD_CHANGED (0.4)
Xsider	1	3.1	1.0	ACTION_INPUT_METHOD_CHANGED (0.6)
Yzhc	1	4.5	0.8	ACTION_USER_PRESENT (0.4)
Fatakr	1	3.2	1.0	ACTION_USER_PRESENT (0.7)
FakeRun	1	4.4	1.0	ACTION_INPUT_METHOD_CHANGED (0.4)
Mobilespy	1	3.1	0.9	ACTION_MAIN (0.4)
Hamob	1	3.4	1.0	ACTION_INPUT_METHOD_CHANGED (0.3)
Jifake	1	2.6	0.8	android.net (0.3)
Fakengry	1	2.6	0.6	UNIQUE_IDENTIFIER_DB_INFORMATION (0.2)
SMSreg	1	1.6	0.9	ACTION_INPUT_METHOD_CHANGED (0.3)

Table 4.3: Results of the genetic search performed towards a family change. For each family it is shown the number of generations required to find the first solution, the average number of generations needed to converge, the average number of modifications applied to each position and the most changed feature for that family.

Results are shown in Table 4.3. As it can be seen, only one generation is required to find at least one valid solution. On average, the genetic algorithm requires 4 generations to converge

(which means between 50 and 350 queries to the classifier). These two values are particularly important because they denote that to find a valid individual able to evade the correct classification is relatively an easy task. BaseBridge represents a special case. No individuals were found able to produce a misclassification in samples of this family. For the rest of the families, DroidKungFu presents the most complex case, where the highest number of generations was needed to achieve convergence. In contrast, SMSreg is the family for which the genetic search requires the lowest number of iterations. This table also includes the minimum number of changes required to cause the misclassification, which is closed to 1. Lower values are due to those samples that are already incorrectly classified, so no change is needed to evade the correct classification.

Regarding the most important feature decisive to provoke the misclassification (shown in the last column of the table), ACTION_INPUT_METHOD_CHANGE plays a fundamental role. In general, any sample implementing this action will be automatically classified as BaseBridge. This is also the reason for which BaseBridge cannot be evaded: the presence of this permission unavoidably conducts the sample to be classified to this family. In particular, this feature refers to an intent defined by the application for receiving a broadcast reporting that the input method has been changed[2].

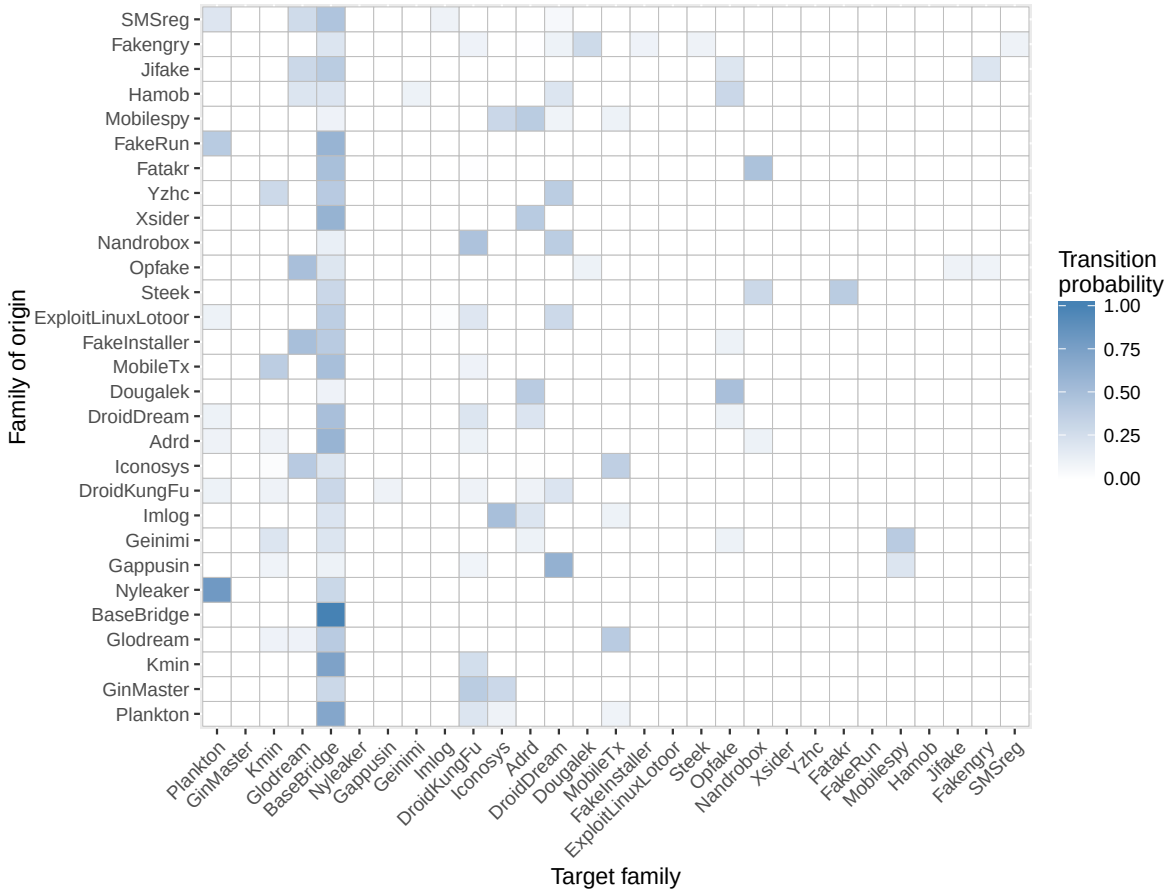


Figure 4.1: Matrix showing the transition probabilities from each family of origin to the different fictitious family destinations.

[2]More information can be obtained at: `https://developer.android.com/reference/android/content/Intent.html#ACTION_INPUT_METHOD_CHANGED`

The most common transitions of each family, that is to say, the labels to which the resulting new vector is classified to, were also studied (see Fig. 4.1). The most likely label destination is again BaseBridge. An important number of mutated vectors are allocated to this family by the classifier. At the same time, it can also be noted a common pattern among families, showing that transitions are not symmetric. This means that when a family is able to reach a fake family through the modifications indicated by the genetic search, the opposite route will be hard to achieve. This make sense when taking into account the limitation imposed in the modifications introduced, which can only be incremental. Then, when a family is represented by a vector entailing higher values in relevant features in comparison with a secondary family, the transition between both will be more likely in one single direction. However, there are exceptions, such as the pair composed by Plankton and DroidKungFu. There also 9 families which are never reached during this genetic search (i.e. GinMaster or Nyleaker, among others).

Moreover, as revealed in the experiments, Plankton and Nyleaker families share almost the same set of intent actions. The difference lies in that Plankton features two more actions. Further, Kmin and GinMaster have a very similar behaviour to DroidKungFu, since just one single modification in these families allows to classify their vectors to this destination family.

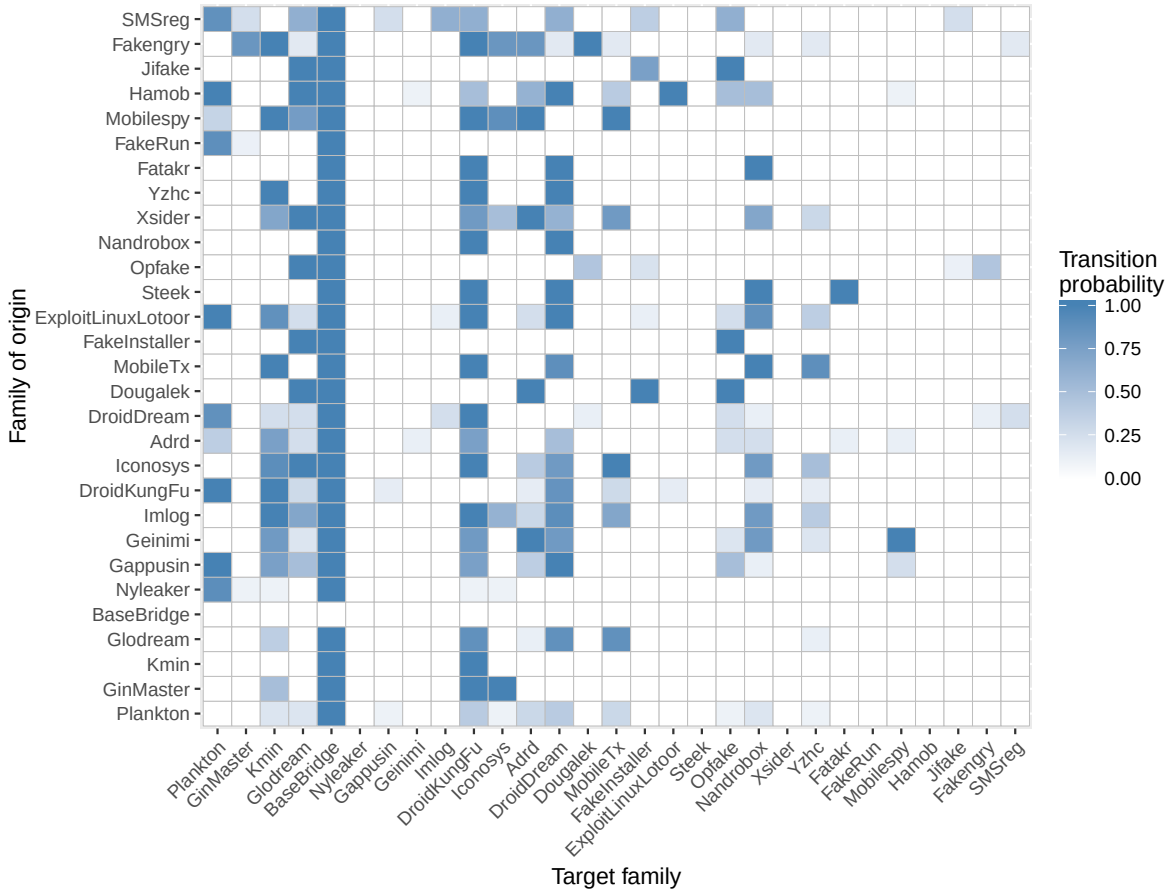


Figure 4.2: Matrix showing the transition probabilities from each family of origin to the different targeted families.

4.3.2 Targeting specific families

The ability of the heuristic search proposed to target specific families was also tested. While in the previous scenario individuals minimise the probability of being classified to the origin family, in this second one the fitness function pursues individuals approaching a target family previously specified (the function is defined by Equation 4.6).

The same process used in the previous subsection is followed for testing the ability to target specific families. But in this case, for every sample of a particular family, each possible destination is tested. The ability to reach the different families is shown in Fig. 4.2. An individual tile in the plot defines the number of samples for which the genetic algorithm found a valid solution (i.e. a new feature vector shaping the new target family) divided by the number of samples of that family. It can clearly be seen how the samples of all families can be disguised as BaseBridge samples. The most remarkable fact found is that there are families which are much more easier to reach than others. For instance, while DroidKungFu or DroidDream can be fetched from multiple origin families, others such as Hamob cannot be accessed from any family.

In general terms, the two experiments shown above demonstrate that it is actually feasible to introduce disturbances in feature vectors in order to disguise the real family of a malware sample. This leads to conclude that machine learning based classification methods in the Android malware domain are liable to be attacked.

4.4 Countermeasure

The purpose of this section is to provide a countermeasure able to detect disrupted feature vectors trying to produce an intentional misclassification. This section demonstrates that the use of a more robust classifier can help to improve the performance when facing modified samples. Fig. 4.3 shows the design of this countermeasure.

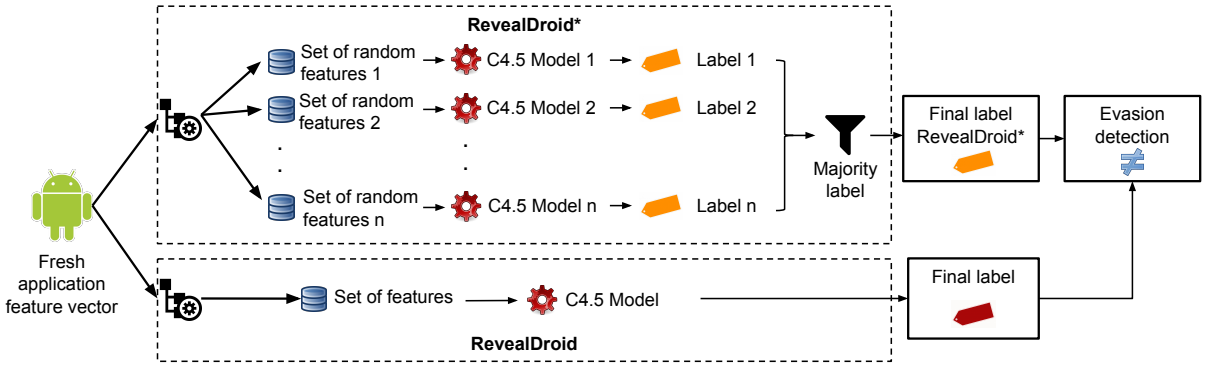


Figure 4.3: Countermeasure designed to deal with the IagoDroid attack.

The new classifier designed to implement this countermeasure is named RevealDroid*, and acts as an ensemble of n classifiers. Each engine composing the ensemble is tied to a specific set of features. Unlike certain ensemble classifiers such as Random Forests [Bre01], features are not share among estimators, indeed, each one is focused on a specific subset. The final label

is decided by means of a voting scheme. Ensemble classifiers have proved to be more robust in comparison to single classifiers against adversarial attacks [CKOF09].

By randomly allocating these features to the different estimators, the attacker will face a major barrier, since he/she will not be able to relate features and classifiers. Furthermore, each individual estimator contributes to the final labelling, so an effective attack would require to disrupt a significant portion of them. In particular, each different estimators keeps the same parameters and the global set of features remains identical.

In order to test the performance of RevealDroid* in comparison to RevealDroid, the same evaluation was made. It resulted in 88% accuracy, a quite similar value. However, the main benefit is the ability to detect potential misclassifications. As shown in Fig. 4.3, both classifiers operate in parallel. Each one provides a label and the related probability of that label. In the comparison of both results lies the strength of the countermeasure: RevealDroid* will alert of a potential misclassification when both classifiers differ.

4.4.1 Reversing the attack

Once detected a sample trying to evade the classifier, it will be placed in quarantine. However, it is also convenient to follow a backtracking process in order to identify potential origin families. For this purpose, data represented in Fig. 4.2 can be used to analyse the most likely transitions which have the fake label delivered by RevealDroid as destination.

A particular example can be drawn from those individuals classified to the Kmin family. As shown in Fig. 4.4, samples incorrectly classified to the Kmin family have 6 potential origin families.

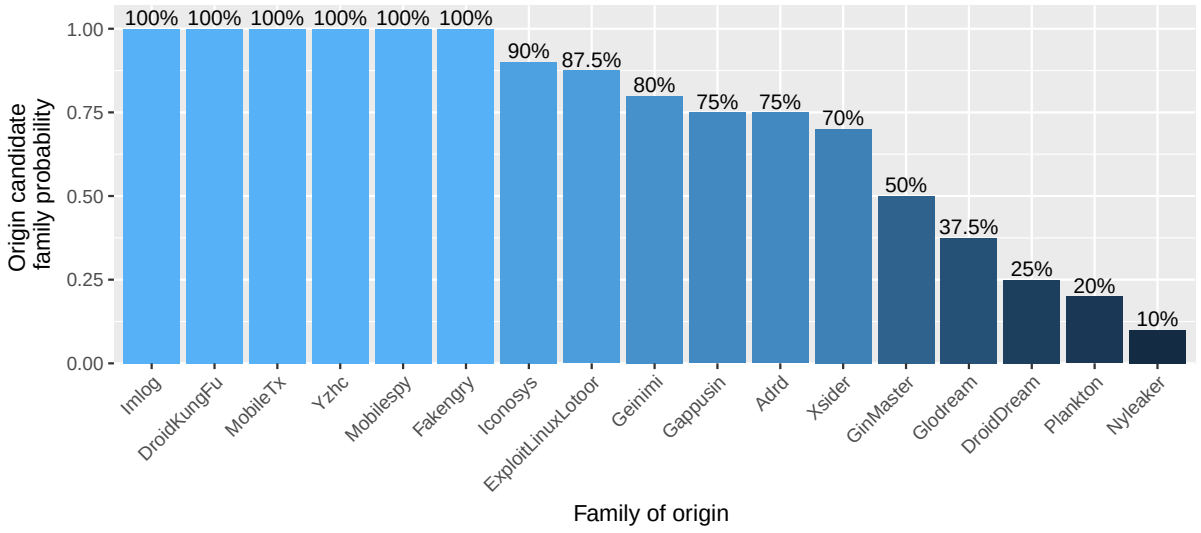


Figure 4.4: Potential families and probabilities of origin for samples camouflaged as Kmin.

Chapter 5

AndroPyTool and OmniDroid

"No sensible decision can be made any longer without taking into account not only the world as it is, but the world as it will be."

- Isaac Asimov, *Asimov on Science Fiction*

The different malware detection algorithms postulated in the course of this work have required broad sets of features extracted from Android malware and benign samples in order to be trained, validated and ultimately tested. In this regard, there is a lack of publicly available data sources. Although it is possible to find collections of APKs, a limited number of datasets provide already extracted feature vectors. This forces researchers and developers to search for tools focus on particular sets of characteristics, and to execute each of them individually until the desired set of features is obtained.

To assist in this process, AndroPyTool [MCC18] was designed, implemented and publicly released aiming at automating the mining process of a varied set of static and dynamic features. The use of this tool allows to efficiently obtain diverse behavioural information that would otherwise require to invest significant time.

In second place, and focused on the already stressed gap of a dataset of feature vectors, the OmniDroid dataset[1] was built. Working from large batches of Android applications, the AndroPyTool was run for each of them, bundling all the results into a dataset containing behavioural descriptive features from 22,000 benign and malicious applications. The major objective of this dataset is to help researchers when training and testing detection tools. These two contributions are presented in this chapter.

5.1 AndroPyTool: an automated framework for static and dynamic feature extraction from Android applications

The extraction of representative behavioural characteristics from Android applications conforms an essential task when designing detection tools able to distinguish between malicious and benign applications. In this process, a large number of existing tools allow to extract specific types of

[1] Available at AIDA Datasets Repository: `https://aida.ii.uam.es/datasets/`

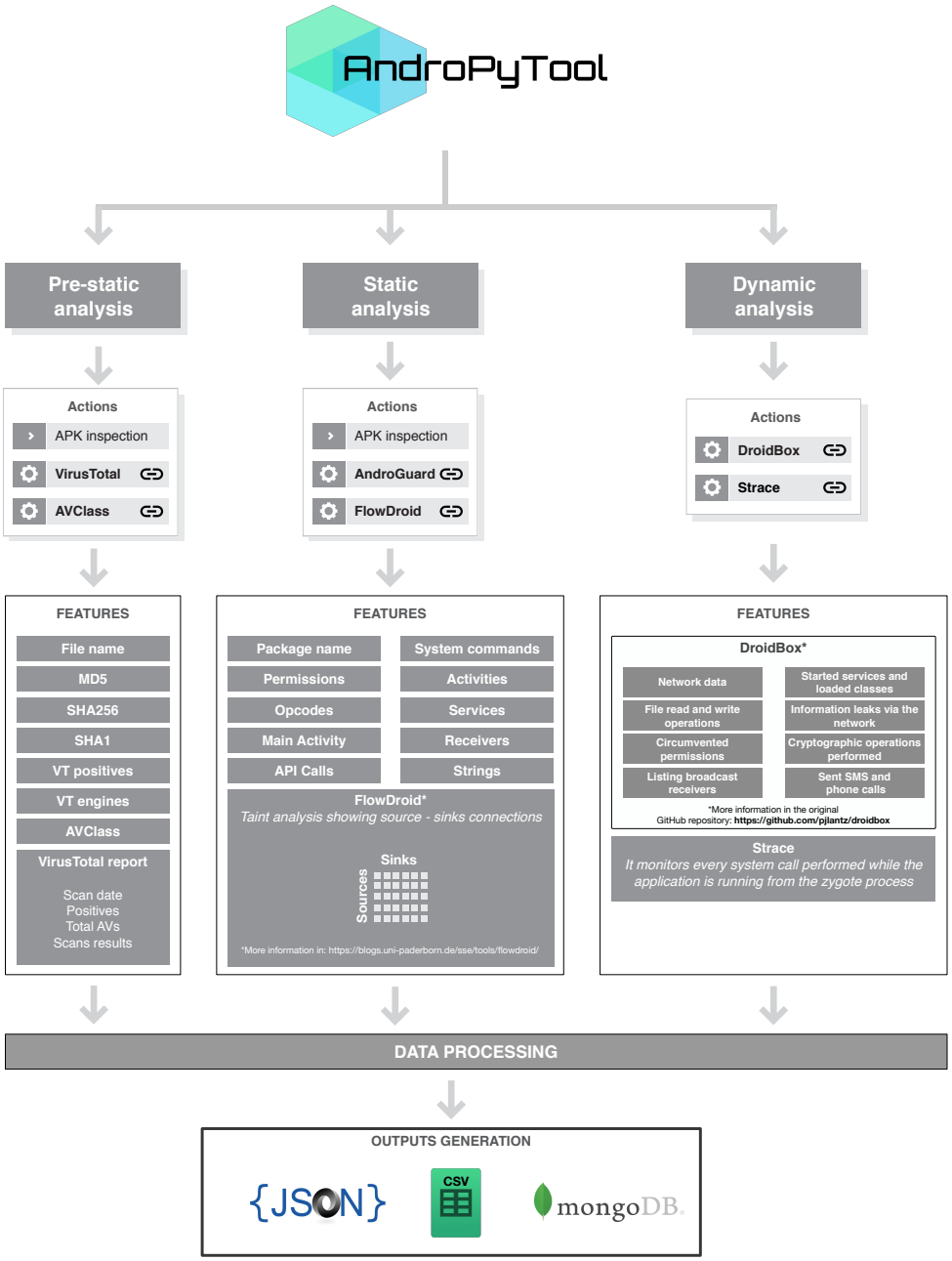


Figure 5.1: Scheme of the different features and the extraction tools used in AndroPyTool.

features at different levels. For instance, it is possible to employ tools focused on extracting static features from the Android Manifest, while others mine information flows.

The individual use of each of these tools offers an extensive description of the behavioural characteristics, however, following this process with large sets of samples becomes a tedious task. With the primary objective of facilitating and accelerating this process, an automated framework named AndroPyTool [MRC18, MLCC18] was designed and implemented. It integrates the most important exponents of Android malware analysis tools in order to provide fine-grained analysis of suspicious samples, including diverse static and dynamic features.

AndroPyTool [MCC18] is an open source Python tool and library which, through a series of steps, retrieves varied information from a given sample by leveraging different existing libraries and tools. A general scheme of AndroPyTool is displayed in Fig. 5.1. The information gathered throughout this process can be categorised into three different categories: *pre-static*, *static* and *dynamic* features. The former category is detached from *static* features for a better organisation. The next subsections describe this seven-step process, detailing the features extracted, how they are processed and presenting the final outputs.

5.1.1 Tool operation

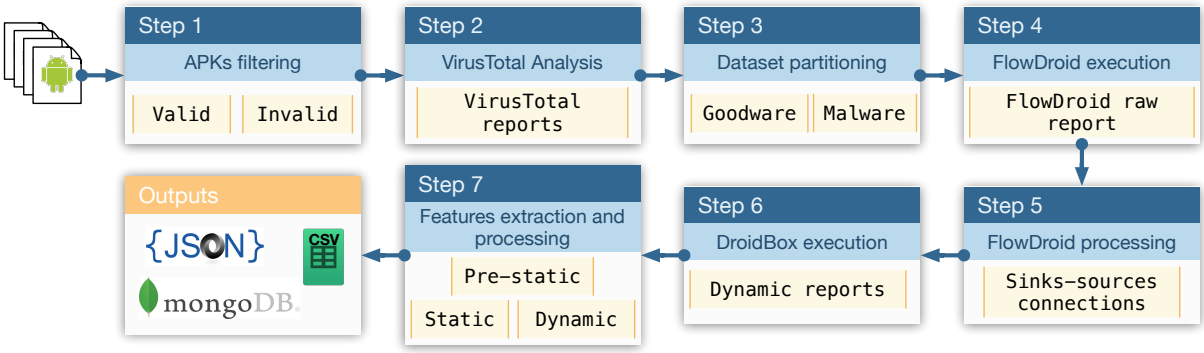


Figure 5.2: Diagram showing the seven-step process followed by AndroPyTool to extract a wide set of static and dynamic features.

AndroPyTool is designed as a modular framework written in Python where a number of scripts are sequentially executed. This enables the smooth integration of new malware analysis tools in the future, or to improve each of them separately. The operation of AndroPyTool follows a seven steps process (see Fig. 5.2) as described below:

- **Step 1: Invalid applications filtering.** This step works as a filtering process which allows to discard invalid applications, those that are not properly constructed, and which therefore may not be executed to obtain dynamic traces. For this purpose, the AndroGuard library for Python [DG13] is used. It includes a function which determines if a given sample is *valid* by parsing the Android Manifest. If this file can be successfully retrieved, then the sample is considered as *valid*. When this first step ends, applications are placed into two different folders, `/valid/` and `/invalid/` according to the result obtained with AndroGuard.
- **Step 2: VirusTotal analysis.** VirusTotal [Vir] is a free online service that allows to scan files, executables or URLs among other formats, with more than 70 antivirus engines. For each of them, and in case of testing positive, a label defines the type or family of malware found. The main goal of this analysis is to provide the ground truth data necessary to categorise each sample between malicious or benign. In order to execute this step, a valid VirusTotal API key must be provided as argument.
- **Step 3: Dataset partitioning.** At this stage, samples are categorised into two different categories based on the report downloaded from VirusTotal and a threshold parameter ϵ defined by the user, aimed at counterbalancing the number of false positives. Thus, those samples that are labelled as malware by at least ϵ antivirus engines are allocated to the

malware set. On the contrary, samples with a lower number of positives are considered as benignware.

- **Step 4: FlowDroid execution.** Information flows pose a profitable data source. Tracing information, from the method where it is originated to the function where it is received, brings useful details regarding the operation of the code. This taint analysis based process allows to retrieve source-sink connections based on a predefined list of calls, representing information leaks that could reveal malicious patterns. The FlowDroid tool [ARF+14] was selected for this purpose, a static taint analysis tool for discovering information flows in Android applications. This step is, together with the dynamic analysis step, the most computationally expensive task. It requires significant time and resources to perform a thorough analysis.

- **Step 5: FlowDroid results processing.** After executing FlowDroid, raw logs containing full details of the taint analysis process are obtained. In this step, these logs are parsed to retrieve a relation of source-sink connections. The reason of dividing the analysis with FlowDroid in two steps (execution and results processing) lies in that different representations or categorisations can be made once the tool execution has finished.

- **Step 6: DroidBox execution.** DroidBox is a tool designed for the dynamic analysis of Android applications [Lan15, Lan11]. It executes a target sample in an emulator while all the actions performed are monitored and captured. This allows to gather a large set of valuable information describing the different interactions of the sample with the device functionalities. Cryptographic operations, file read and write operations, or information leaks via network are some of the events inspected. A modified version [MCC17] of the DroidBox original repository was used in order to include the execution of the Strace tool at the Linux level, to run the emulator in a non-GUI environment and to increase the number of automatically induced interactions in the sample under analysis.

- **Step 7: Feature extraction.** The final step is in charge of processing all the information gathered in the above steps and of extracting a large sets of pre-static and static features. The different files composing the application are inspected with the help of AndroGuard in order to retrieve static features and parsing the *smali* files (obtained from decompiling the original Dalvik Executable *DEX* files) to obtain, opcodes, strings or system commands included throughout the code. This step generates the final outputs of AndroPyTool: a feature file for each apk detailing all the features extracted[2] in JSON format and a reduced version in CSV format. AndroPyTool also allows to directly export all the information gathered from a set of samples to a Mongo database. This step also allows to generate a file containing feature vectors which can be used to train or evaluate machine learning algorithms.

5.1.2 Features extracted by AndroPyTool

Table 5.1 offers a summary of the pre-static, static and dynamic features that AndroPyTool extracts. The different subsections provide a description of each individual feature according to the corresponding category.

[2]Strace logs are not included in this individual feature file due to the large size of these reports.

	Feature	Description
Pre-static	Filename	Filename of the APK
	VT positives	Number of antivirus which test positive for malware
	VT engines	Number of antivirus used in the analysis
	AVClass	Agreed malware label from several detection engines according to [SRKC16]
	MD5	MD5 checksum of the APK
	SHA1	SHA-1 checksum of the APK
	SHA256	SHA-256 checksum of the APK
Static	API Calls	Count of system calls performed by an APK
	Main Activity	Name of the Main Activity
	Opcodes	Count of opcodes performed by an APK
	Package name	Name of the package
	Permissions	Which permissions uses the APK
	Intent receivers	Set of an APK's receivers
	Intent services	Services used by an application
	Intent activities	Activities declared by an APK
	Strings	Set of defined strings (with use count) within an APK
	System commands	Set of system commands ran by the app
	FlowDroid	Path to the results obtained by FlowDroid [ARF+14]
Dynamic	DroidBox	Analysis performed with the DroidBox dynamic analysis tool
	Strace	A list of all the actions performed at the Linux level

Table 5.1: Summary of the pre-static, static and dynamic features that are extracted by AndroPyTool

5.1.2.1 Pre-static features

Pre-static features feed general information of the sample, with the primary objective of identifying and categorising each sample. The main difference with static features lies in that pre-static characteristics do not imply decompiling or to access the code. In this segment of features, the next fields are included for each application:

- **File name:** the original name of the file.
- **VT positives:** the number of antivirus engines included in the report obtained from VirusTotal which consider the sample as malicious. This allows to label the sample as malware or benignware according to a threshold ϵ.
- **VT engines:** the total number of antivirus engines included in report downloaded from VirusTotal. This value allows to obtain a rate of antivirus which test positive for malware in combination with the previous described field.
- **AVClass:** if the sample is determined to be malware, an agreed family categorisation is calculated with the AVClass tool [SRKC16].
- **MD5:** the MD5 checksum of the sample.
- **SHA1:** the SHA-1 checksum of the sample.
- **SHA256:** the SHA-256 checksum of the sample.

5.1.2.2 Static features

This second fraction of features include information gathered from the Android Manifest and from the code. Static features arrange an useful representation of applications, since they reveal a large set of details regarding the spectrum of actions that the application can take. In this section, the FlowDroid static taint analysis tool is also employed to discover information flows. The relation of features extracted is the following:

- **API calls:** a full record of all the API calls invoked throughout the code. They constitute one of the most important static features, since they provide meticulous information of the actions taken. However, reflection or dynamic code loading techniques are able to surpass this kind of analysis, avoiding to show how the malicious payload is triggered.
- **Main activity:** the name of the main activity as shown in the Android Manifest, which can help to track the application and to find similar samples.
- **Opcodes:** instructions extracted from the bytecode that provide behavioural information at a low level. A study on the effectiveness of this feature for malware detection has already been made [CDLM+15].
- **Package name:** the name of the main package, and that works as the unique application ID.
- **Permissions:** they conform a security mechanism of the Android platform to protect the access to sensitive components (i.e. the microphone). They force the user to explicitly grant the access to that component. A list of the permissions requested by the sample allows to draw general behavioural patterns.
- **Intent receivers:** a list of the different receivers declared in the Android Manifest. They allow to know which intents the application is designed to receive.
- **Intent services:** a list of long processes defined by the application to be run in background.
- **Intent activities:** a list of the activities defined by the sample.
- **Strings:** this feature is used to include all strings found within the *smali* code. Some malware pieces conceal the malicious payload as string variables, thus hampering its detection.
- **System commands:** found within the strings have also been represented as a different feature. They could disclose functionality not represented by the API calls feature (i.e. to unzip a file with the *zip* command).
- **FlowDroid:** a matrix representing the number of information flows found between each pair of source and sink.

5.1.2.3 Dynamic features

Dynamically extracted features group characteristics that have been captured in runtime, and that show the actual behaviour of the sample. This allows to record events that could not be detected through static analysis, such as actions resulting from code dynamically loaded.

- **DroidBox:** all the events captured with DroidBox during the dynamic analysis are also processed by AndroPyTool. All these events are categorised into different categories. A further description of this tool can be found in Section 2.4.2.
- **Strace:** a log registering all the events captured at the Linux level is also retrieved by the modified version of DroidBox employed [MCC17].

5.1.3 Implementation and use of AndroPyTool

When designing and implementing AndroPyTool, there has been an attempt to facilitate its use as much as possible. It can be used as a command line Python tool which offers a large number of execution options, in order to run specific steps (i.e. to perform only dynamic analysis). Besides, it can also be used as a Docker container, which allows to directly run AndroPyTool without installing individual libraries. A further explanation of both methods is provided below.

- **Docker:** This is the fastest and more practical option to run AndroPyTool. It does not require to install dependencies, to download the different required repositories or to configure the Android emulator for the dynamic analysis stage. It is only needed to have Docker installed and to download the Docker image from Docker Hub. Then, the container can be launched by providing a path to a folder containing the APKs to be analysed. Furthermore, the use of Docker allows to run multiple parallel instances of AndroPyTool. Below are shown the two command lines needed to download the AndroPyTool image and to start the container performing all the analysis steps.

```
1 docker pull alexmyg/andropytool
2 docker run --volume=</PATH/TO/FOLDER/WITH/APKS/>:/apks alexmyg/andropytool -s /
    apks/ -all
```

- **Source code:** AndroPyTool can also be executed from its source code. To do that, a series of system libraries, the Android SDK, different Android packages, repositories and Python libraries must be installed. This option also allows to invoke the different functionalities of AndroPyTool from other projects or libraries or to directly launch each script individually. Detailed information describing the installation can be found in the GitHub repository [MCC18].

5.2 The OmniDroid dataset

The existence of datasets containing Android malware or benignware is mostly limited to packs of APKs. Datasets such as the Android Malware Genome Project [ZJ12], Drebin [ASH+14], the Androzoo project [ABKT16] or the Android Malware Dataset [WLR+17] offer raw samples which can be used to later extract the desired set of features. Other datasets such as Droid-Cat [RF16, RFB17] or AndroMalShare [And13] provide reduced groups of features extracted from applications. In order to fill this gap, the OmniDroid dataset was built.

OmniDroid arises from the application of AndroPyTool to a large set of benign and malicious samples. It has been built with a view to providing a benchmark dataset useful for training and

testing Android malware detection tools based on machine learning mechanisms. It includes a plethora of static and dynamic features which empowers the application of feature selection techniques, feature representations, classification methods or pattern recognition algorithms. OmniDroid has been released under a *Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License* [cca13] and it can be downloaded from the AIDA Datasets Repository[3].

The samples composing the OmniDroid dataset were downloaded from two different sources. On the one hand, a collection of 100,000 samples was provided by the Koodous Team[4] for research purposes, containing both benign and malicious samples. On the other hand, malware samples were also gathered from the AndroZoo[5], increasing variety in the malware samples. The AndroPyTool itself was used to filter and to analyse the samples. At the same time, applications with a repeated package name and those that could not be executed in the Android emulator employed by the dynamic analysis tool DroidBox were removed. With the goal of generating a balanced dataset of malware and goodware, all samples were submitted to the VirusTotal portal in order to obtain a ground truth revealing the nature of each sample. Finally, if only one antivirus engine which consider the sample as malware ($\epsilon = 1$) is required to decide the nature of the sample, the dataset is composed by 11,000 benign and an equal number of malicious samples. A deeper description of this dataset can be obtained from the related contribution [MLCC18].

Malware		**Benignware**	
Permission	**% samples**	**Permission**	**% samples**
INTERNET	96.21%	INTERNET	94.82%
ACCESS_NETWORK_STATE	85.45%	ACCESS_NETWORK_STATE	72.95%
WRITE_EXTERNAL_STORAGE	81.31%	WRITE_EXTERNAL_STORAGE	61.49%
READ_PHONE_STATE	80.21%	WAKE_LOCK	41.60%
ACCESS_WIFI_STATE	60.40%	ACCESS_WIFI_STATE	39.14%
WAKE_LOCK	49.05%	READ_PHONE_STATE	37.13%
ACCESS_COARSE_LOCATION	41.99%	VIBRATE	33.33%
GET_TASKS	39.12%	ACCESS_FINE_LOCATION	27.70%
ACCESS_FINE_LOCATION	37.00%	ACCESS_COARSE_LOCATION	27.45%
VIBRATE	36.87%	GET_ACCOUNTS	26.82%

Table 5.2: Most frequent permissions declared in the Android Manifest for each application in the malware and benignware sets of the OmniDroid dataset.

By analysing the features extracted for benign and malicious samples, different conclusions can be made. For instance, there is a notorious greater use of SMS and telephony services among malware samples, a fact which is revealed by the declaration of a larger number of permissions (see Fig. 5.2) or API packages related to these functionalities. The `READ_PHONE_STATE` permission, which allows to obtain personal information such as the phone number or a list of ongoing calls, is much more used among malware samples. The same occurs with `RECEIVE_BOOT_COMPLETED`, a permission employed to launch the malicious payload when the device has been rebooted. Other static features, such as opcodes or API packages, do not allow to extract big differences when comparing both sets.

The differences in the use of information flows were also studied (see Fig. 5.3 and 5.4). Links related to the `NO_CATEGORY` and `NOT_EXISTING` categories are ommited, since they represent non-private data flows. An important number of flows can be found connecting `SMS_MMS` and `IPC`

[3] `https://aida.ii.uam.es/datasets/`

[4] `https://koodous.com/`

[5] `https://androzoo.uni.lu/`

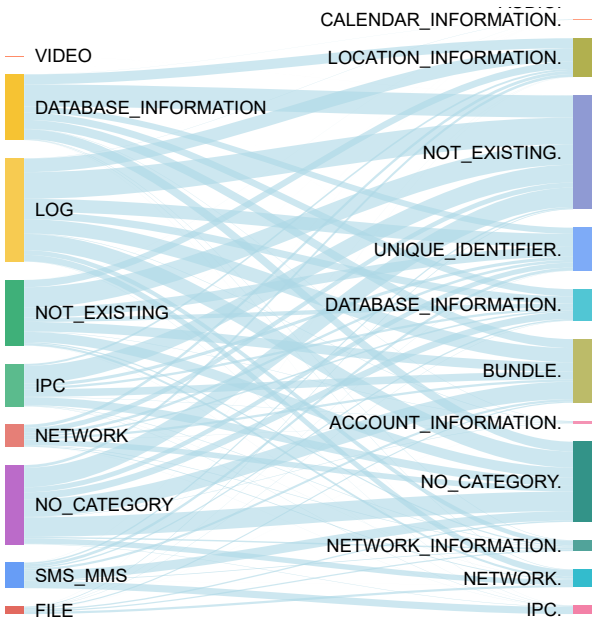


Figure 5.3: Number of information flows between categories of sinks and sources discovered by FlowDroid among all samples in the malware set.

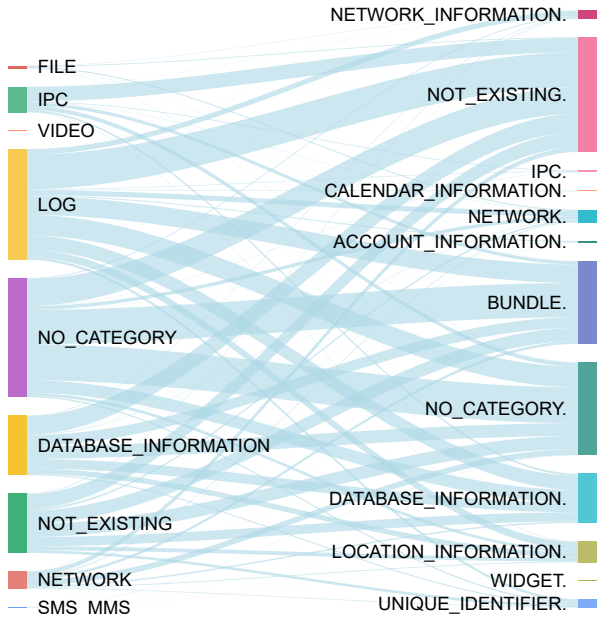


Figure 5.4: Number of information flows between categories of sinks and sources discovered by FlowDroid among all samples in the benignware set.

in the malware set. However, the category `SMS_MMS` show a lower number of interactions in the benignware dataset.

Finally, Fig. 5.5 shows the cumulative sum of the number of operations after a 300 seconds execution in the emulator. Malware features a greater use of cryptographic operations. This finding can be attributed to two possible reasons: due to the deployment of the malicious payload, in many cases encrypted to avoid its detection by antivirus engines, or due to its use by recent

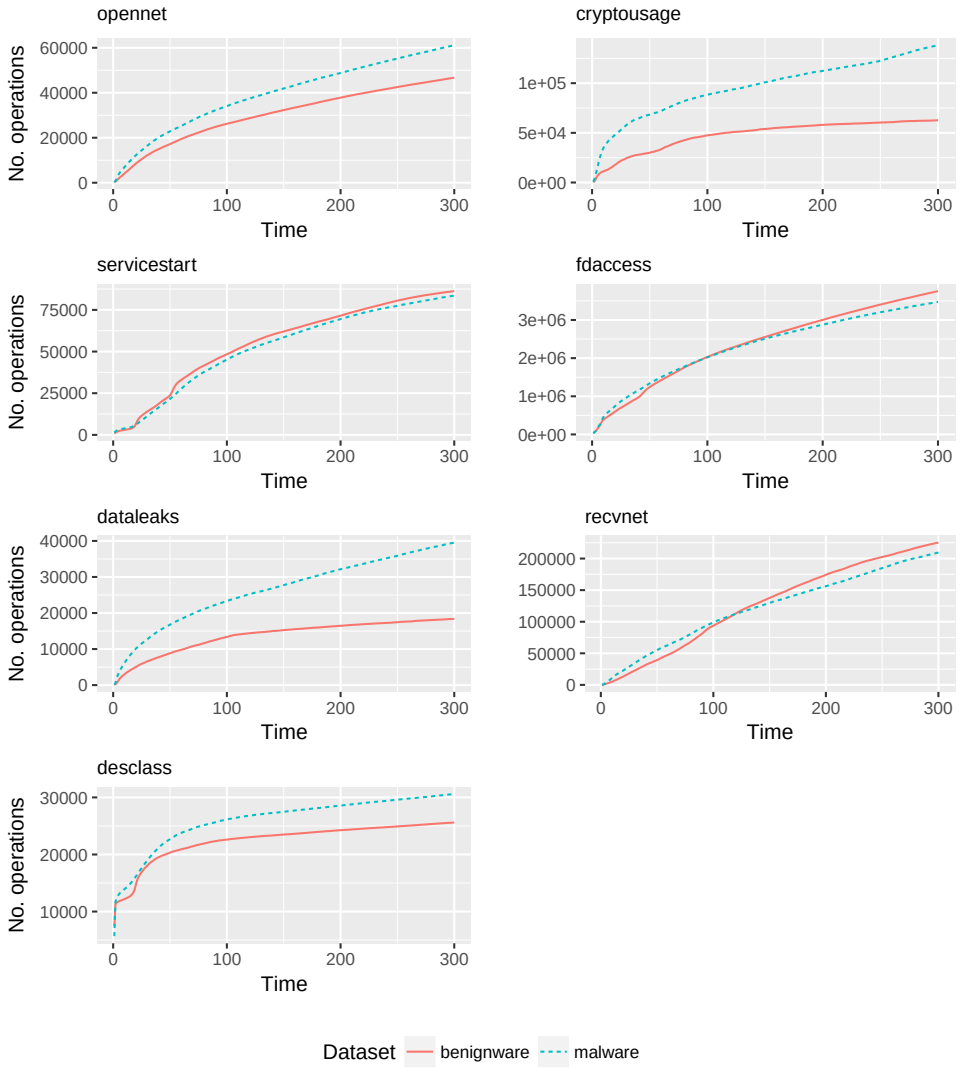


Figure 5.5: Cumulative sum of the number of operations, when *Droidbox* tool is used, for all samples over both (benignware and malware) datasets.

ransomware samples, which encrypt all user's data. In the rest of categories of events, both types of samples exhibit a similar behaviour.

Chapter 6

Conclusions and future work

"There are limits beyond
which your folly will not carry you.
I am glad of that. In fact, I am relieved."

- Isaac Asimov, *Robot Dreams*

In this final chapter, a series of conclusions reached after all the research conducted in the course of this dissertation are presented. Then, the different Research Questions raised in Chapter 1 are answered, with the goal of providing useful details and comments for future researchers working in the Android malware detection and classification scenarios. Further, the final section suggests several potential future lines of work.

6.1 Conclusions

Throughout the different chapters of this dissertation, the detection and classification problem of Android malware has been addressed from different perspectives, all of them related to the use of machine learning algorithms.

In Chapter 2, the Jisut family of Android ransomware was presented. The study of the different variants of this family has allowed to observe important behavioural patterns. It was noticed how different variants are created over time, by embedding new changes in old samples of the family. This is an important finding to take into account, since the early detection of the former applications of a new family can help to detect new samples and variants in the future. The analysis has also made it possible to detect some practices which need to be considered in order to develop more accurate malware detection and classification tools. For instance, the most modern samples of Jisut hide the malicious payload in files which are decrypted in runtime. This has special relevance when using static analysis techniques, which could not detect this content. It has also been explored a series of patterns which highlight the importance of monitoring certain events. The declaration of certain permissions or a particular intensive use of system calls could be related to traces malware.

After the analysis of the Jisut family, different mechanisms are proposed in Chapter 3 for detecting and classifying Android malware. Starting with the detection problem, different combinations of features and classification algorithms have been tested. Among these, static features

have proved to be a powerful instrument to represent the behaviour of each sample, and they have been used to train different ensemble classifiers that have reached high accuracy values.

Dynamically extracted features were also tested. In this case, a Markov Chains based representation is adopted to transform the logs obtained with DroidBox. First, a sequence of events is captured for each application. Then, a matrix of transition probabilities between states (the events of the original sequence) is built according to the different transitions observed. Furthermore, the importance of each event or state is also considered by calculating its frequency among all the states. The matrix of transition probabilities is vectorised and the state frequencies are added to create the feature vector which represents each application. Ensemble classifiers were also used in this scenario, showing lower accuracy values in comparison to the experiments involving static features. The same chapter describes a novel fusion approach of static and dynamic features aimed at improving the results achieved by each group of features independently. Through a voting scheme, the two classification algorithms which provided the best results for each group of features are combined, softly improving previous results.

The classification of Android malware into families is also discussed in this dissertation. For the experiments, a dataset containing Android malware families and a set of dynamic features (following the previous Markov Chains based representation) were used. The results, obtained using deep learning architectures, classic machine learning algorithms, and also techniques for dealing with imbalanced data show that it is possible to use these techniques for creating accurate Android malware family classification tools.

Chapter 4 analysed the issue at hand by testing its resilience against adversarial attacks. A state-of-the-art classifier, named RevealDroid, enabled to demonstrate that machine learning aided classification tools can be defeated by modifying the feature vector of a sample with incremental changes. A countermeasure is also analysed in this section, showing that ensemble classifiers which distribute the categorisation of the sample to different estimators allow to counteract the attack raised.

Finally, Chapter 5 has shown an overview of the AndroPyTool framework developed in the course of this work. This tool allows to automatically obtain a large set of both static and dynamic features from Android applications. Furthermore, by running AndroPyTool over a large set of samples of benign and malicious samples, it was possible to generate a dataset of feature vectors. The goal is to provide the community with a balanced dataset composed of a wide set of already extracted features.

6.1.1 Response to Research Questions

This sections answers each of the Research Questions raised in Chapter 1:

RQ1: *Which are the most important malicious practices among Android malware samples to be considered when designing detection tools?*

The study performed on a large number of samples belonging to the Jisut family of Android ransomware [MHCC18] described in Section 2.3 has allowed to extract a series of important details. All this information, mainly related to their implementation or to structural patterns shared among samples, allows to understand not only this family, but also to analyse the most common mechanisms employed by malware targeting the Android platform.

By tracing similarities between samples, it has been demonstrated how the Jisut ransomware has originated different variants which implement small variations. At a more general level, a temporal evolution can be observed, where variants are taken as the starting point to develop new samples, through the use of incremental modifications. This is an interesting finding, since it means that there can be found strong similarities between samples of the same family, even if they belong to different variants. At the same time, this trend highlights the importance of the classification of malware into families, an essential task to detect new variants of the same family and also new unknown families. This knowledge can be later leveraged to build detection tools with a wider coverage of malware families.

The analysis also arises the importance of cryptographic operations in families of ransomware. For instance, some of the samples of this family encrypt user's files in order to force him/her to pay the ransom. This action causes an unusual number of API calls invocations related to cryptographic operations. In this sense, a control of the API calls invoked must be deemed in detection mechanisms, since it expresses patterns reflecting behaviours specific to ransomware samples.

Another interesting practice is the use of novel concealment techniques which was observed in the newest variants of Jisut. By hiding the malicious payload into separated files or libraries, in some cases encrypted, the malware tries to hinder static malware analysis techniques. Thus, a dynamic analysis based approach appears to be more convenient to deal with this issue. The use of a sandboxing environment, where the suspicious application is executed and all the interactions are monitored, allows to capture fine-grained details that would not be feasible to extract if a static analysis procedure is chosen.

- **RQ2:** *Is it possible to find large labelled dataset of features extracted from Android malware and benign samples?*

  While there is a plethora of literature focused on the Android malware detection problem, there is also a lack of datasets containing already extracted features from samples, needed to train machine learning aided detection and classification tools. In most of the cases, authors employ sets of executables from which the desired set of features is obtained. This process becomes more complicated when using multiple features and several specific extraction tools are needed. Current datasets which offer behavioural information instead of just providing the original executable of each application are limited. As it was discussed in Section 3.1, some of them offer a reduced number of logs [RF16, RFB17] while others include a small number of features from only malware samples [And13, SRKC16].

  In order to make it easier the process of developing and testing new malware detection and classification mechanisms, it was decided to build a new dataset, covering a large number of both statically and dynamically extracted features. For this purpose, a framework named AndroPyTool which integrates different state-of-the-art malware analysis tools was built. This tool allows to efficiently and automatically extract the most used static and dynamic features in just one step, instead of using multiple specific software. AndroPyTool has been released as an open source repository [MCC18].

  Then, a large number of benign and malicious samples were gathered from Koodous[1] and AndroZoo[2] and analysed with AndroPyTool. This allowed to generate the OmniDroid

[1] `https://koodous.com`
[2] `https://androzoo.uni.lu`

dataset, which contains pre-static information, static and dynamic features from 11,000 benignware samples and 11,000 malware samples. This dataset is publicly available and it can be downloaded from the *AIDA Datasets Repository*[3]. The goal is to provide researchers with a benchmark dataset that they can use to build or test new detection mechanisms.

- **RQ3:** *Can Machine Learning classification methods be combined with static features to detect Android malware accurately?*

  Section 3.1 describes new different methods for Android malware detection and classification. Through a representation based on static features, it has been proved that ensemble classifiers mainly composed by decision trees can be used for building accurate malware detection and classification tools. Binary and numerical features make this kind of algorithms appropriate to solve this task.

  Particularly, features such as API calls, permissions declared, opcodes, intent filters, services or system commands have been used in the experiments. From them, different conclusions can be made. For instance, the use of different combinations of static features for malware detection has evidenced that API calls provide the best behavioural description to distinguish between malware and benignware accurately. Even when combined with other important data sources such information flows, the individual results achieved with system calls are not improved.

  In general, static analysis has evidenced to be an efficient and also accurate procedure to feed machine learning classifiers, reaching high accuracy and precision values. In comparison to dynamically extracted data, this approach does not require to execute the sample in a resource intensive and time consuming sandboxing environment. Instead, each APK is decompressed and different data are extracted from the different files and resources. This enables to detect malicious samples with close to 90 percent accuracy through a Random Forest classifier.

- **RQ4:** *Is it possible to apply Machine Learning over dynamic traces to detect Android malware accurately?*

  While static features provide a fast and efficient behavioural description of the sample, they can also fail at showing the most important malicious patterns under certain circumstances. For instance, modern obfuscation techniques, which encapsulate the malicious payload of the sample into hidden files, can make a static description useless. It is only when the suspicious sample is executed that the full functionality can be captured, including the dynamic loading of the hidden pieces of malicious code. On that basis, dynamically extracted features were studied in Section 3.2.

  Unexpectedly, the experiments, performed with ensemble classifiers, show a reduction in the number of samples succesfully allocated to their correct category in comparison to static features based detection methods. There are several potential causes of this outcome. On the one hand, the set of events monitored by DroidBox can be insufficient to detect fine-grained patterns only associated with one particular category. On the other hand, a dynamic approach could not cover the whole operation of the suspicious application (i.e. the malicious payload is only triggered when the user access to a particular section of the sample).

  For that reason, the original operation of the *MonkeyRunner* tool was modified in order to send a larger number of actions to the screen and buttons of the application when executed.

[3] `https://aida.ii.uam.es/datasets/`

The results also point to the need for a combination of static and dynamic features. The former enables to cover a wider number of the actions that the application could invoke. Dynamic features allow, conversely, to report specific characteristics, which can be crucial to determine the nature of the sample. Thus, it is mandatory to study if a combination of both static and dynamic features can lead to build stronger classifiers.

- **RQ5:** *Is it possible to combine static and dynamic features in order to build more effective detection mechanisms?*

  In Section 3.3, a model for the fusion of static and dynamic features was presented. The classification algorithms which showed the best performance in classifying static and dynamic features are combined through a voting scheme, where each group of features contributes to the final decision. The importance of each group of features in the decision is given by two weights calculated through a grid search: 0,7 for the classifier receiving static features as input and 0,3 for the classifier which receives dynamic information. It can be said that static information is specially important in the final categorisation.

  The fusion approach hardly improves the results obtained by the two groups of features individually, from 89,3% to 89,7% accuracy. Although the difference is small, it reflects that dynamic features could help, in a very reduced number of cases, to better detect malicious patterns. Besides, the combination of both features sources is also important to build classifiers more robust against adversarial attacks. As shown in Chapter 4, ensemble classifiers, and also a wide number of features, hamper the success of these attacks, since the exploratory space becomes bigger and more complex.

  The use of better dynamic analysis tools can also help to improve the results in fusion approaches of static and dynamic features. In this sense, it is necessary to study new dynamic analysis mechanisms, employing more advanced emulators, undetectable by the sample under analysis, and which are able to capture a wider range of interactions of the sample with the operating system. This will enable to enhance the performance of current malware detectors and family classifiers.

- **RQ6:** *Is it feasible to attack machine learning classifiers to produce family misclassifications?*

  The attack implemented and tested described in Chapter 4 has demonstrated that machine learning aided Android malware classifiers can be circumvented, thus producing family misclassifications. Through the use of a heuristic search guided by a genetic algorithm, the original feature vector of a sample can be modified by adding incremental features. This can lead to allocate the sample to a different random family or to target a specific previously selected family. The changes can be introduced in the sample by using opaque predicates never executed, thus avoiding to modify the semantic of the sample.

  This attack proves that machine learning based methods are vulnerable. In particular, classifiers that rely on a reduced number of features to deliver a label can be more easily defeated. To deal with this situation, a countermeasure is raised and implemented, where the original classifier is replaced by a set of classifiers which receive independent groups of features. This allows to complicate the implementation of these attacks, since the classification this time depend on the decision of several estimators.

6.2 Future Work

While all the work performed has tried to properly deal with the Android malware detection and classification problem from different points of view, there have been also observed certain points where this research can be extended:

- The analysis of the Jisut family has shown that it is possible to find interesting behavioural patterns. Studies on other important families of Android malware are required in order to bring to light implementation details which can help to design and build stronger detection tools. Furthermore, they can also be useful to improve classification methods, by discovering new variants of known families or by grouping under the same family different samples. At a lower level, this kind of studies allows to analyse the encryption and decryption mechanisms and the methods used to lock user's devices, among others.

- In general, although the results obtained have shown high accuracy and precision rates, it is still required to analyse new features, representation, data processing techniques and learning algorithms, since there is space for improvement. More in particular, other features need to be explored, covering a larger number of files, obtaining information from compiled libraries and also studying the presence of dynamic code loading functions. It is also necessary to study combinations of characteristics in order to improve the results obtained with API calls.

- The experiments using dynamically extracted features evidence that further research needs to be done in this regard. While these features can definitely help to improve detection and classification tools, more advanced stealthy emulators are required. It is necessary to study how the sample under analysis can be stimulated through more realistic interactions, to monitor a wide series of events and to analyse new procedures to combine this information with statically extracted features.

- Malware classifiers are a powerful tool to allocate samples to their corresponding family, thus keeping better track of the different existing families, enhancing the detection of zero day malware and detecting new variants of current known families. The triage process is an essential task to deal with malware, specially with the possible damages caused if the device is succesfully infected. By knowing its malware family, it is much more easy to apply the most convenient steps to avoid its propagation and to mitigate its effects. More research can be done in this regard, improving current malware classifiers.

- The attack designed, implemented and tested which was described in Chapter 4 was used to demonstrate that machine learning aided classification tools can be circumvented. This entails a series of considerations which must be taken into account. Future research should be focused on testing the resilience of these tools and proposing new countermeasures to deal with potential attacks.

- Finally, there is also space for improvement in AndroPyTool and the OmniDroid dataset. The former can be extended by integrating new malware analysis and reverse engineering tools, with the goal of extracting a wider set of features. This can help to develop better detection and classification mechanisms and also to build new datasets. In this sense, the OmniDroid dataset can be improved in different directions: incrementing the number of

samples, providing a better description of the different malware families included so the dataset can be used for family classification, or defining a richer selection of features.

Part II

Publications

PUBLICATION 1

An in-depth study of the Jisut family of Android ransomware

(IJ-1) **Martín, Alejandro**; Julio Hernández-Castro & David Camacho: "An in-depth study of the Jisut family of Android ransomware." *IEEE Access*, Vol. 6, pp. 57205-57218, 2018, DOI: 10.1109/ACCESS.2018.2873583
Impact factor = 3.557 (JCR, 2017) [Q1, 24/148, Computer Science, Information Systems].

- **Contributions of the PhD candidate:**
  - First author of the article.
  - Contributions made in the conception of the presented idea.
  - Analysis and application of reverse engineering techniques to a large number of varied samples of the Jisut family.
  - Analysis of the common implementation and behavioural patterns among samples.
  - Co-author of the interpretation and discussion of results provided.
  - Co-author of the manuscript, figures and tables presented.

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received July 19, 2018, accepted September 5, 2018, date of publication October 4, 2018, date of current version October 29, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2873583

An in-Depth Study of the Jisut Family of Android Ransomware

ALEJANDRO MARTÍN[1], JULIO HERNANDEZ-CASTRO[2], AND DAVID CAMACHO[1]

[1]School of Engineering, Autonomous University of Madrid, 28049 Madrid, Spain
[2]School of Computing, University of Kent, Canterbury CT2 7NF, U.K.

Corresponding author: Alejandro Martín (alejandro.martin@uam.es)

This work was supported in part by the Next Research Projects, such as the Comunidad Autónoma de Madrid under Grant S2013/ICE-3095 (CIBERDINE: Cybersecurity, Data and Risks), in part by the Spanish Ministry of Science and Education and Competitivity (MINECO), and in part by the European Regional Development Fund (FEDER) under Grant TIN2014-56494-C4-4-P (EphemeCH) and Grant TIN2017-85727-C4-3-P (DeepBio).

ABSTRACT Android malware is increasing in spread and complexity. Advanced obfuscation, emulation detection, delayed payload activation or dynamic code loading are some of the techniques employed by the current malware to hinder the use of reverse engineering techniques and anti-malware tools. This growing complexity is particularly noticeable in the evolution of different strands of the same malware family. Over the years, these families mature to become more effective by incorporating new and enhanced techniques. In this paper, we focus on a particular Android ransomware family named Jisut, and perform a thorough technical analysis. We also provide a detailed overall perspective, which will hopefully help to create new tools and techniques to tackle more effectively the threat posed by ransomware.

INDEX TERMS Ransomware, Jisut, android, malware, malware families.

I. INTRODUCTION

When current mobile operating systems made their first appearance, late in the first decade of the current century, there was already an extensive know-how on designing and fighting against malware aimed at personal computers. The emergence of malware targeting these new mobile platforms was a foretold event. The importance reached by smartphones in our daily lives have made them a particularly attractive target, and this is specially true of the Android platform. Whether due to its more open structure or to its notoriously higher market share, most of malware developer's efforts focus on Android. Some of the advantages offered by the Android platform unfortunately make it also an excellent target for developing and distributing malware, not only by experienced developers and cybercriminals, but also by beginners.

The increasingly key role that smartphones play in our daily lives turn them into a perfect bridge for extorting victims. Unsurprisingly, ransomware has emerged as a very profitable business, allowing to blackmail a victim by locking access to the device, frequently in combination with encrypting data files or throwing false accusations of illegal activity, with the ultimate goal of demanding a hefty ransom.

Although there is an abundance of literature studying Android malware, most of these works focus on a small number of research paths: they either center around designing detection tools [1], evaluating the effects of obfuscation tools [2], on malware classification, or on detecting samples containing a malicious payload [3]. Curiously, the work we encompass in this paper, that is, a thorough research focusing on a fine-grained analysis of the features and evolution of a single malware family, seems to constitute a new approach.

We think that an in-depth study of the most important Android malware families can help to understand their evolution, both from a low level perspective (to evaluate implementation details) and from a high level (to assess common patterns between variants of the same family). While this has been a pointless exercise in the past, mostly due to the extreme simplicity of the known malware families, the current complexity and the consistent evolution and improvement they are now experiencing warrants, in our opinion, the need for a more detailed screening.

In this paper we aim to provide a deep insight on a specific Android malware family called Jisut, which has been mainly distributed on Chinese markets (although there can be found variants translated to other languages) and has taken many different shapes, leading to numerous Jisut variants.

The common denominator of these is that they ask for a ransom after having locked the device with a permanent screen, or after encrypting user's files, and that they share clear structural patterns. However, as it will be shown later, there are also versions which only pursue to lock the operation of the system, while offering no recovering option.

Throughout the different sections of this paper, the Jisut family and its most important variations are carefully examined. It will be shown how these variants have emerged, and how they had evolved to lead to new variants. At the same time, the locking and encrypting mechanisms are inspected and also exploited, providing the necessary details for recovery when getting infected by this ransomware.

The contributions of this research can be summarized as follows:

- To describe the Jisut family of Android ransomware and its most important variants, outlining their purposes, providing the most significant implementation details and studying their encryption and/or screen-locking mechanisms.
- To perform a temporal analysis of the evolution of the different variants found in the wild, studying how modifications and improvements are successively included.
- To explore the weaknesses of this ransomware, in order to provide the necessary details to recover both the device and user data.

The rest of the paper is organised as follows: Section II describes the background and related work, Section III presents the Jisut family and some information regarding the evolution of its most important variants, Section IV describes the technical details of this family, Section V includes a series of remarks based on the analysis performed and Section VI provides some conclusions and recommendations.

II. BACKGROUND AND RELATED WORK

A. ANDROID RANSOMWARE EVOLUTION

In its almost ten years of existence, Android has been constantly pointed as the main target of malware authors. Despite all the new security policies and other novel countermeasures implemented, Android remains attractive as a platform to design and develop new malware. Although when Android first appeared in 2008 an extensive experience in building malware for personal computers already existed, the limitations of the platform made it difficult to translate it to Android. But this appears to be changing, particularly since 2016. As Malwarebytes Labs state in their 2017 State of Malware Report [4], Android is evolving to accommodate more complex software and, hence, more powerful malware.

A clear evidence of this growing complexity is Android ransomware, which is now our main focal point [5] in this work. Starting from a brief definition, "a ransomware is a kind of malware which demands a payment in exchange for a stolen functionality" [6], it is possible to categorise samples of this type of malware into two different classes, depending on the procedure adopted to coerce the victim [7]: lockers (also called screen-lockers) or cryptoransomware. Added to this, we also have a related category, scareware.

Regarding lockers, they try to stop most of the device functionality by making use of persistent screens which cannot be closed, or by locking the device with a password. In the case of cryptoransomware, the malware encrypts user's files, so it is necessary to pay the ransom to recover them. Depending on the encryption methods used, we can identify [8]: private-key ransomware, public-key ransomware or hybrid ransomware, where a random secret key is generated in the device and encrypted using public-key cryptography. Finally, in scareware [9] the coercion procedure involves threatening or frightening the victim. For instance, making public some personal information or falsely accusing the victim of holding illegal content (i.e. child pornography).

Regarding ransomware specifically designed for Android, the first implementation able to encrypt files was called Simplocker, reported in 2014 [10]. It showed a screen accusing the victim of having child pornography while the user files were encrypted in the background. A ransom was asked for unlocking the victim's data, which was encrypted using a fixed key that can be found in the ransomware code. Later, an evolution of this malware was described in 2015 [11], able of communicating with its authors. In this new variant, Simplocker is more complex and, for example, employs unique keys.

Other family of malware usually cited in security reports is Lockerpin [12]. While old versions of this family tried to lock the victim's device by constantly prompting a screen, recent samples make use of the native Android locking system. This procedure, for which the user has to grant Device Administrator privileges, is really effective and cannot be easily removed or bypassed. The Jisut family, also described in the 2017 Trends in Android Ransomware by ESET [12], has been widely spread in the Chinese market. With similar aims and methods to the previous mentioned families, Jisut locks the device by showing a permanent screen where the user is encouraged to pay a ransom. Currently, many other ransomware families are active: Slocker, Koler or LockDroid are some of the most dangerous families that have emerged over the last years [13].

B. ANDROID MALWARE FAMILIES ANALYSIS IN THE LITERATURE

So far, research related to Android malware has usually studied it from a general perspective, taking sets of samples of varied families as a whole, without explicit attention to the specifics that each kind of malware family presents. To the best of our knowledge, only one previous research has made a deep analysis of a malware family. In that study, the GinMaster [14] family is described quite technically, analysing the different generations that have appeared over time, and mentioning the improvements which have been sequentially added.

Other literature focused on this topic adopts a more general perspective. Thus, an interesting research by

Zhou and Jiang [15] offers overall details of a big set of Android malware families, providing a few technical details and some general patterns. Andrubis [16], [17] draws a wide analysis of a huge dataset of Android malware samples, with the aim of providing a dataset of features, but no technical details of the families are provided. Monika and Lindskog [13] perform a study showing general trends among Android families, describing their appearance over the years. However, the approach taken is very general, and particularities and technical details are not provided.

Other literature is focused on developing analysis and detection tools. For instance, multiple research studies broad feature sets to discern the nature of applications. The use of third-party calls [1], string-based features [18] or API-calls, permissions and network addresses [19] are some of the features extracted from sets of malware and benign software to build detection tools. To these features, other tools have also incorporated more determinant features such as taint analysis, used by Revealdroid [20], or dynamically extracted information, as it is the case of Droid-Sec [21]. DroidSieve [2] is also focused on presenting a tool for malware detection and classification. This tool constitutes an interesting step forward against obfuscated malware, giving special attention to obfuscation-invariant features and directly extracting information from the *DEX* files.

Specifically focused on Android ransomware, Andronio *et al*. [22] concentrate on extracting features able to detect malware thanks to the use of encryption processes, threatening texts or locking services. A similar approach opts for including into the model threatening pictures or logos [23]. The use of API packages has also been studied [24] to discern between apps of different nature without specific previous knowledge. Instead of using code-level features, the effects of ransomware have been measured by monitoring hardware metrics, such as processor or memory usage [25]. The particular weaknesses of the Android platform when dealing with ransomware has also been studied by Yang *et al*. [26]. However, neither these nor previous literature analyse malware families independently.

The need to focus on the specifics of each family has also been highlighted in the literature [27]. Wei *et al*. state that when gathering a dataset of malware samples, detailed and reliable information must be provided. This means, according to the authors, that each type of malware must be profiled independently and that manual analysis become mandatory.

III. THE JISUT RANSOMWARE

The Jisut family started spreading in 2014. There are no available reports on the number of users infected, but it is probably a significant figure, for the reasons shown below. We can, however, approximate the number of different samples detected by antivirus engines during these years. For instance, based on the database of the VirusTotal Intelligence portal,[1] 4,693 different samples have been detected by at least one antivirus from 2014 as belonging to the Jisut family.[2]

Nevertheless, even classifying these samples as variants of the Jisut family is a non-trivial issue. Some of these are also categorised as Slocker, or as belonging to other families by different antivirus. This problem has been already highlighted in several research works, which showed that the procedure for naming malware families [28] is inconsistent. This is clearly visible when uploading a sample to the VirusTotal service, as the categorisation performed by the different antivirus can vary significantly.

Even when two engines agree on the type classification of a piece of malware, they can call it as belonging to different families. Added to this is the fact that there are some engines which attribute no explanatory names (i.e. just a number sequence) to malicious samples. Different researchers have concentrated on addressing this problem, and have built tools to offer an agreed tag [29].

However, sometimes it is possible to observe how different malware families along different variants are distinguishable due to the use of common structural patterns. Although Jisut has unmistakable patterns, retrieving samples of different variants becomes an arduous task. In this research, in order to gather a varied and representative set of Jisut samples, a manually intensive work to search for individual samples was necessary. Throughout the paper, we will mainly refer to these variants with their main package name.

With regard to the structure and general characteristics of the Jisut family, it is important to stress the simplicity observed in its coding style. This fact suggests authorship by people with a lack of experience, possibly young. These beginners probably started by reading the easy-to-find documentation available in many Chinese webs and blogs containing instructions on how to develop a simple lock-screen ransomware.

Among the variants found, the same base structure can be identified. On top of this structure, we find from variants implementing very small changes to versions where the attacker opts for adopting a totally different cryptoransomware-based model instead of the screen locking scheme. Five screenshots of some of the most important variants of this family are shown in Fig. 1.

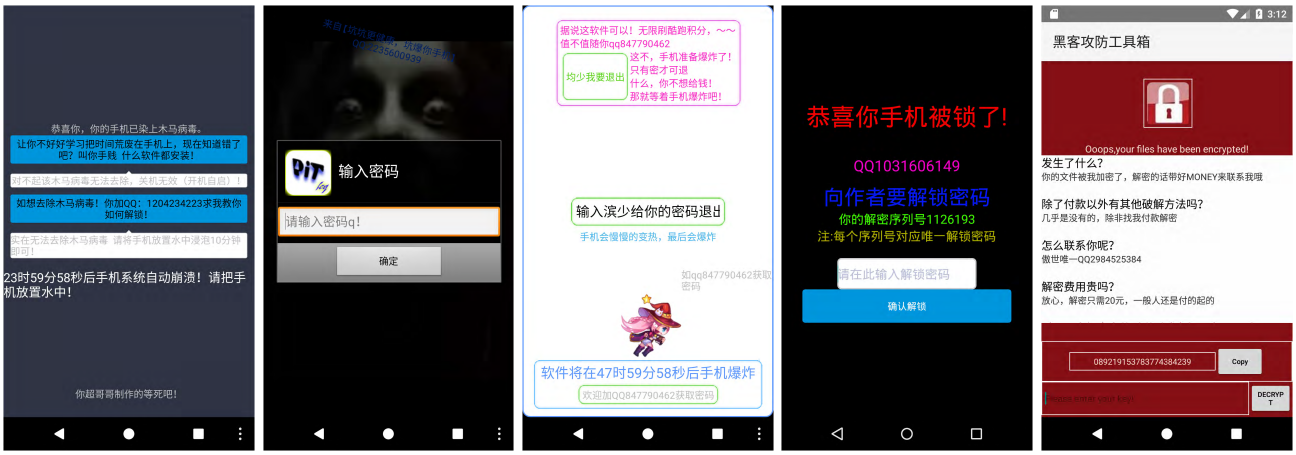
A. THE EVOLUTION OF JISUT

We first have analysed the evolution of this family in terms of number of distinct samples found, month by month, by the VirusTotal portal and reported as Jisut by at least one antivirus[3] (see Fig. 2). This family has had two moments of wide popularity: When it appeared in June 2014, new samples were continuously found for almost a year. At the beginning of 2016 it was reactivated, and it reached its global maximum

[1] http://virustotal.com/intelligence/

[2] We have applied a threshold of two minimum different sources uploading a sample, in order to avoid minor variations which have not spread widely.

[3] We have applied a threshold of two minimum different sources uploading a sample in order to avoid minor variations.



(a) (b) (c) (d) (e)

FIGURE 1. Screenshots of the main variants of the Jisut ransomware, sorted by year. (a) Variant tk.jianmo.study (2014). (b) Variant lichongqing_shuang (2014). (c) Variant nero.lockphone (2015). (d) Variant qqmagic (2016). (e) Variant Hongyian - Huanmie (2017).

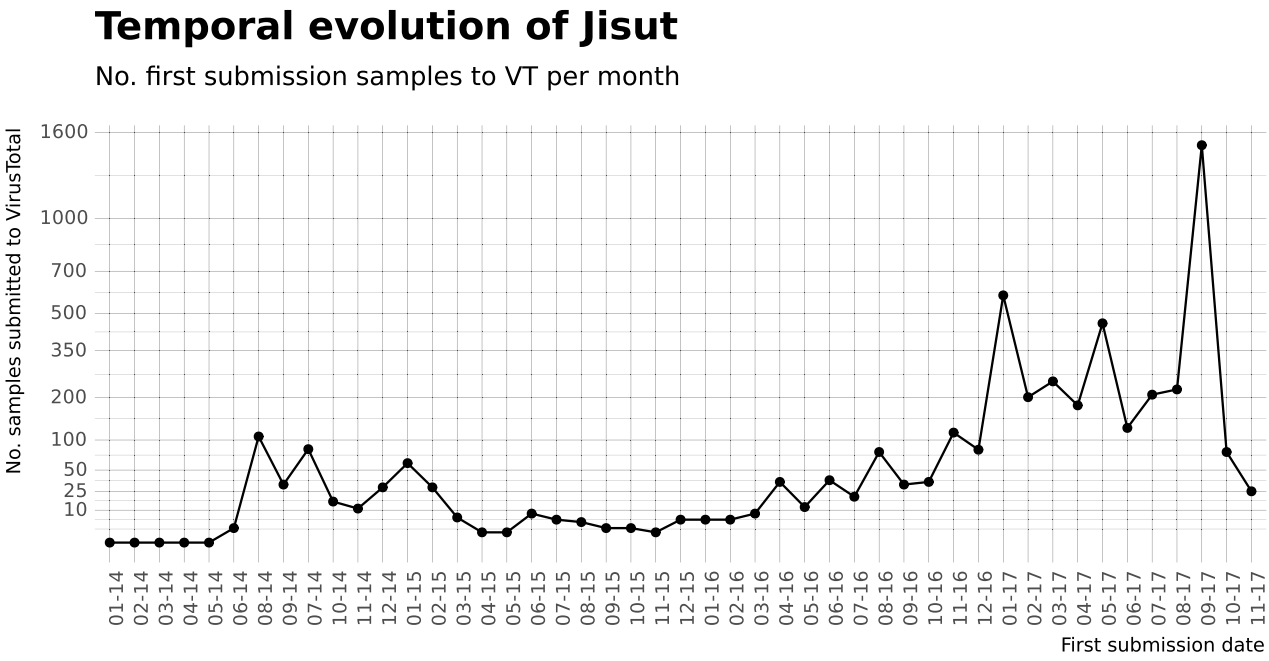


FIGURE 2. Evolution of the number of samples categorised as Jisut submitted to VirusTotal, per month.

in September 2017. In this month, around 1,500 new samples of different variants were found.

From these, different broad sample sets, which share almost an identical code but which include minor changes (i.e. a different package name or a different message on the screen), can be identified. We call these sets *variants*. The differences found between variants may include different encryption mechanisms, different forms of scaring a victim, etc. Fig. 3 shows the most important variants (which are described in depth below) of the Jisut malware. In this figure, interesting behavioural patterns among variants can be identified. The most significant characteristic lies in how the number of uploads has peaks of different size depending on the variant. For instance, the *tk.jianmo.study* generation, which can be considered as the original one, had a peak relevance during the second half of 2014 and the beginning of 2015. Then a long hibernation is easy to spot. After that, at the beginning of 2017, the most important peak is reached detecting 70 new samples in January.

It should be also noted how the *Nero.lockphone* variant appeared when the original family was decreasing in popularity, at the beginning of 2015. From that moment, both variants have followed a very close pattern. With two recent peaks in January and September 2017, both variants seem to behave in a very similar fashion. This fact could reflect an organised campaign, where the same people work simultaneously with different variants, but it could also be the result of a ripple effect. One way or another, there is also a seasonal component. The four highest peaks in the plot, August 2015, January 2015, January 2017 and September 2017, correspond to a period immediately after holidays. This makes sense, particularly among young people, who have significant more exposure time during holidays. Holiday gifts in the form of new smartphones can also play a role.

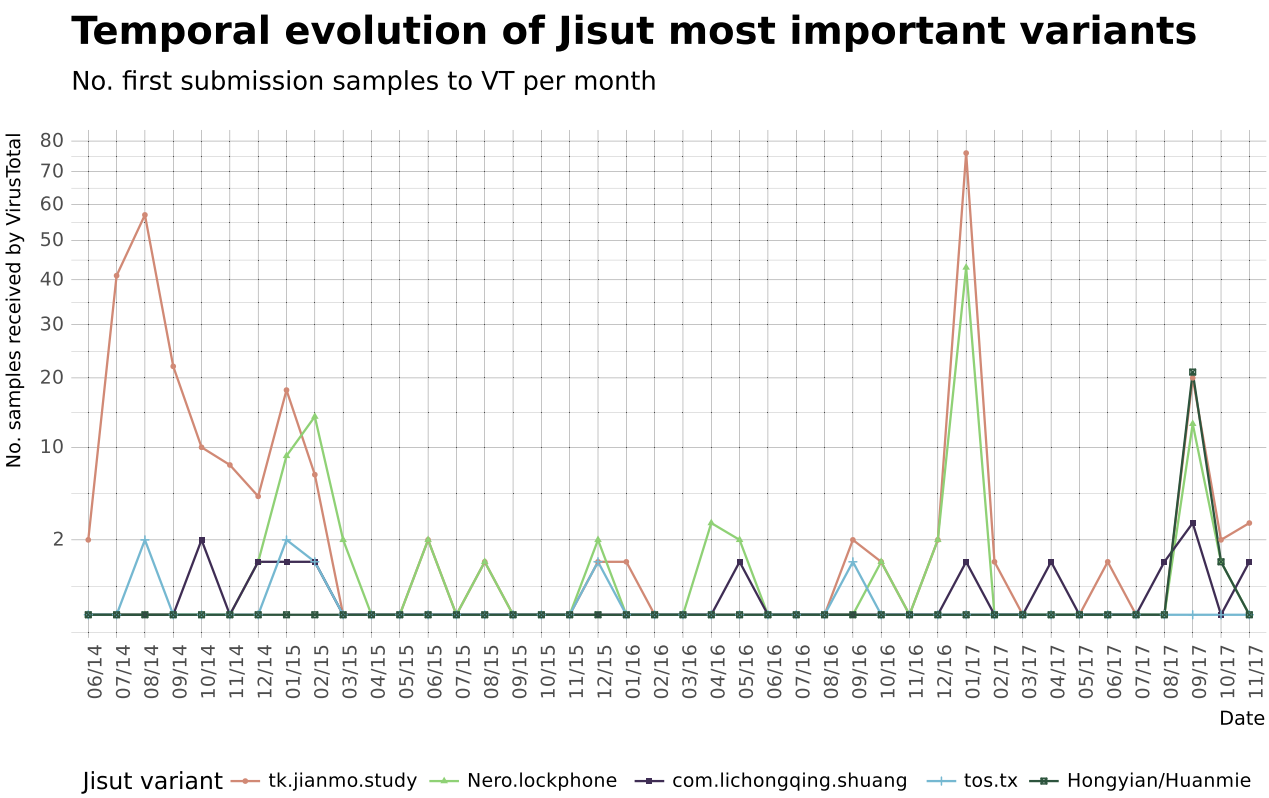


FIGURE 3. Evolution of the different Jisut variants, in terms of number of samples submitted to VirusTotal, per month.

In contrast with the two previous variants, the plot suggests other variants follow different trends. In the case of *com.lichongqing.shuang* or *tos.tx*, their repetitive pattern over time has a reduced number of new samples detected. This difference among variants can be ascribed to different criminal groups working independently.

IV. TECHNICAL IMPLEMENTATION DETAILS OF JISUT

This section deepens the analysis of these variants, revealing technical implementation aspects, such as the the necessary actions to undermine the integrity of the infected system and the procedures used to encrypt user's files. At the same time, the evolution of each family is analysed separately.

A. THE JIANMO VARIANT

In June 2016, the first samples categorised as Jisut were detected.[4] These samples, whose main package is *tk.jianmo.study*, implement a lock-screen malware.

1) APPLICATION ANALYSIS

Once installed and launched,[5] this ransomware shows a screen (see Fig. 1a) which reports that the device has been infected by a Trojan virus, and that the user must contact the author via the QQ messaging service within 24 hours. Otherwise, user's data will be definitely removed. At the bottom, a timer registers the remaining time. We have checked the operation of this counter, proving that turning the system clock back makes no difference (so the user cannot extend the time). By taking a look into the code, we can see that a file located in the path `/data/data/tk.jianmo.study/shared_prefs/TimeSave.xml` is continuously updated to store the remaining time. In a few samples of this variant that we have studied, when the timer expires, the user's files are not removed (this functionality is in fact not implemented). However, as it will be shown, there are numerous variants which actually materialise this threat.

This malware is composed by just one package with several classes:

```
/
└── tk.jianmo.study
    ├── BootBroadcastReceiver.class
    ├── BuildConfig.class
    ├── MainActivity.class
    ├── R.class
    └── killprocessserve.class
└── LogCatBroadcaster.class
```

The first class, `BootBroadcastReceiver.class`, implements the necessary code to restart the app if it is closed, by means of a BroadcastReceiver which launches `MainActivity.class` if a *Broadcast* is received. This last class manages the timer of the app, as explained, and overrides the `onKeyDown()` method in order to control which buttons are pressed:

The previous code works together with the following method:

[4]The first sample on 6th June 2016. Can be identified by SHA-256:789f8bfedf8f04ee8fe9c01cc0bda76604a89bf6fc641cd75dc9221a1a2a7ac3

[5]For this analysis, we have use the sample identified by SHA-256:4aaf1687316ffa6de108e12768b8434a9f12b07ea6953450cbf8a2a6b633fdc1

```
1  public boolean onKeyDown(int paramInt, KeyEvent
       paramKeyEvent){
2    if (n == 4) {
3      if (keyTouthInt == 0) {
4        usedTime = SystemClock.
       currentThreadTimeMillis();
5        keyTouthInt = 1;
6        usedTime = System.currentTimeMillis(); }
7      else if (keyTouthInt == 1) {
8        keytouch(usedTime, keyTouthInt, 1); }
9      else {
10       keytouch(usedTime, keyTouthInt, 4); }
11   }
12   if (n == 3) {
13     keytouch(usedTime, keyTouthInt, 5);
14   if (keyTouthInt == 6) {
15     new MyDialogFragment().show(getFragmentManager
       (),"mydialog");}
16   }
17   if (n == 82){keytouch(usedTime,keyTouthInt,100)
       ;}
18   if (n == 25){keytouch(usedTime,keyTouthInt,2);}
19   if (n == 24){keytouch(usedTime,keyTouthInt,3);}
20   if (n == 26){
21     Toast.makeText((Context)this, (CharSequence)
       "眙眙眙眙眙眙，开机自启，关机也没用的峨，
       "扣电池也没用的峨！！！！睦眙眙", 0).show();}
```

Listing. 1. Detection of keystrokes in the Jianmo variant.

```
1  public void keytouch(long paramLong, int paramInt1
       , int paramInt2){
2    this.newTime = System.currentTimeMillis();
3    if ((this.newTime - paramLong <= 'B') && (
       paramInt1 == paramInt2)){
4      this.usedTime = this.newTime;
5      this.keyTouthInt = (paramInt1 + 1);
6      return; }
7    this.keyTouthInt = 0; }
```

Listing. 2. Keystroke detection in the Jianmo variant.

The goal of these code bits is to detect when a particular sequence of keys are pressed. This is used to hide the deactivation mechanism, which is prompted when the user presses a certain sequences of keys. Said sequence is provided by the criminal when the ransom has been paid. The method used consists on evaluating when a particular key has been pressed. As it can be seen in Listing 1, several conditional statements compare the key pressed. Then, the *keyTouch()* method is called with the *keyTouchInt* value and a constant. When these two values are equal and the last key was pressed less than 2 seconds ago, the value of *keyTouchInt* is incremented by 1 in line 6, Listing 2. If these two conditions are not met, the value of the variable is reset to 0 (line 7, Listing 2). If the value of keyTouchInt reaches the value of 6 (line 14, Listing 1), a dialog is prompted which asks the user to introduce a code while threatening the victim it will delete all its data if not. The sequence of keys, in terms of KEYCODES is: *4-4-25-24-4-3*, that correspond to the keys:

> KEYCODE_BACK - KEYCODE_BACK - KEYCODE_VOLUME_DOWN - KEYCODE_VOLUME_UP - KEYCODE_BACK - KEYCODE_HOME

The last key is the HOME key, which although the Android system does not allow to directly detect when pressed (the *onKeyDown()* method is not called) is commonly used in lockware like this by overriding the method *onAttachedToWindow()* and changing the type of the window, as it can be seen below (see Listing 6). However, this trick is no longer functional in the newest Android versions.

```
1  public void onAttachedToWindow(){
2    getWindow().setType(2004);
3    super.onAttachedToWindow();
4  }
```

Listing. 3. Override of *onAttachedToWindow* method in the Jianmo variant.

2) VARIATIONS OF THIS VARIANT

Throughout 2014, this variant was spread featuring only minor changes. In most of them, modifications are limited to different messages or package names. However, it is valuable for this work to glimpse through how attackers employ simple alterations to build new pieces of malware, since they allow us to gather further insights on the key trends of the evolution of ransomware.

For instance, one common pattern found among samples that are almost clones is the use of different package names, mostly by adding suffixes to the original name. This might be an attempt to upload new samples to markets such as Google Play and/or to produce, through new signatures, false negatives by one or more antivirus. For instance, among the samples of this variant found in 2014, from 35 to 41 of the antivirus included in the VirusTotal service test for positive, depending on the sample. Worse still, an average of 35% of the antivirus engines incorrectly return a negative classification. Examples of these new package names, derived from the original `tk.jianmo.study` are `tk.jianmo.studyds21` or `tk.jianmo.studypj7m76mo`.

Alternatively, differences also exists at the code level. In another sample,[6] we can observe an slightly different specification of the *onKeyDown* method. But in this particular case we are facing a useless piece of code, since it does not lead to unlock the secret screen.

Other variation of this method found in a different sample[7] is used to define a different sequence of key presses to unlock the secret screen. This time, the user must press twice the

[6]Identified by SHA-256: 9e99dd63b41dffb12af7a82bad4efc80bf095edcd6fe3dc718630dc76335b28a

[7]Identified by SHA-256: d2a5aed7c26caf55721460f252d6119c0ab6ffefbda875c42fccb1e5c71de873

back key followed by a different key. Then, it is possible to introduce the deactivation key, which is a string formed by 10 spaces.

B. THE LICHONGQING SHUANG VARIANT

One of the branches originated in 2014 evolved into a curious type of scareware (see Fig. 1b). Analysing a sample of this year,[8] we found it plays a loud scream sound and shows and frightening picture. The creator tries to scare and to coerce the victim into paying the ransom. This lock-screen malware also employs a hidden menu, which is activated through a long press in the upper section of the screen. Again, the key is assigned in the code, in plain text, to a variable. This makes it easy to extract. In this particular sample, the key is: "*2235600939*".

The malware makes use of the *MediaPlayer* resource to play the scream sound:

```
1  this.audioMgr =((AudioManager)getSystemService("
       audio"));
2  this.maxVolume =this.audioMgr.getStreamMaxVolume
       (3);
3  this.m = new MediaPlayer();
4  this.m = MediaPlayer.create(this, 2130968576);
5  this.m.setLooping(true);
6  this.m.prepare();
7  this.m.start();
8  new Handler().postDelayed(new Runnable(){
9    @Override
10   public void run(){
11     Object localObject = mService.this;
12     Class localClass = Class.forName("com.
       lichongqing.shuang.Activity2");
13     localObject = new Intent((Context)localObject,
       localClass);
14     ((Intent)localObject).addFlags(268435456);
15     mService.this.startActivity((Intent)
       localObject)
16     mService.this.Y();
17     mService.this.m.start();
18     ((Vibrator)mService.this.getSystemService("
       vibrator")).vibrate(mService.access$L1000004(
       mService.this), 0);
19   return; }
```

Listing. 4. Mediaplayer invocation in the Lichonqing Shuan variant, preventing volume decrease.

The code, shown in Listing 4, starts by setting the volume to its maximum level (line 2). Then it invokes the mediaplayer to play the sound on an infinite loop, while continuous actions to increase the volume are sent in order to counter any attempts by the to decrease it. In line 18, it also employs the vibration function,

C. THE NERO.LOCKPHONE VARIANT

Samples of this variant (see Fig.1c) were detected for the first time in 2014, but it was in 2015 when it was widely spread. Although the graphical interface of this variant[9] is indeed substantially different from the samples previously mentioned, the behaviour and intentions are identical. Proof of this can be found just by taking a look at the code, where it can be seen that the operation is also basically the same. It encourages the user to contact the criminals through the QQ chat app (where it is presumed he will ask for a ransom). At the code level, the package structure contains the same classes with identical names. The only major difference lies in the deactivation procedure. On this occasion, the text box to introduce the deactivation code is shown from the outset on the screen.

The unlock code is also saved as plain text within the code:

```
1  this.up = ((int)System.currentTimeMillis());
2  if (this.up - this.down < 200){
3    if (!MainActivity.access$L1000002(MainActivity.
       this).getText().
      toString().equals("旭哥QQ1767332988!")){
4    break label236;
5    }
6    MainActivity.this.stopService(this.val$kill);
7    System.exit(0);
8  }
```

Listing. 5. Unlock procedure in the Nero.Lockphone variant.

The ransomware checks (Listing 5) the time the button on the left of the smartphone is pressed (line 2). When the user performs a long press the app shows a counter, probably to confuse the user. When the button is only briefly pressed, the code inserted by the user is compared against the string "旭哥QQ1767332988!". If both values are the identical, the application terminates (line 7).

D. THE QQMAGIC VARIANT

The messages shown by the previous analysed versions display various kinds of threats to incite the victim to pay a ransom. However, the malicious payload is limited to screen locking, with unlocking possible after using a key provided in plain in the code. Even when this key is encrypted, the original one can be easily obtained since we can observe how it has been encrypted with a symmetric key. However, this *qqmagic* variant implements some interesting improvements which make the process of obtaining the unlocking code through reverse engineering much more complicated.

For instance, in a sample of this variant,[10] the attacker makes use of SMS services in order to receive a password, randomly generated and encrypted. Thus, each time this ransomware is installed by a different victim, a new and different password is generated, which is shared with the attacker through a SMS. This allows to generate victim dependent numbers, which the attacker use to generate victim dependent deactivation codes. In Listing 6 it is possible to see how

[8]Identified by SHA-256: 8043461bc97509bdf3300376898040d5dba4b5f5804e942c1d0b4fb4119b69f9

[9]Identified by SHA-256: 4bed20bdb3586dfea0b7a09e28a0126ebc05669551d53c4c9ac69aaee5ca8f69

[10]Identified by SHA-256: b914c0dd57ffcb1c96cf37d61a3ae052a5372f01c5fac3ea0535bbdb0da862dd

two variables, which are used to calculate the unlocking password, are initialised (lines 1 and 2), how a DES object is initialised with a string (line 3) and also how the SmsManager service is used (line 6):

```
1 this.pass = ((Math.random() * 10000000));
2 this.passw = ((int)(Math.random() * 1000000));
3 this.des = new DesUtils("QQ1031606149");
4 this.share = getSharedPreferences("GreyWolf", 0);
5 this.editor = this.share.edit();
6 this.sms = SmsManager.getDefault();
```

Listing. 6. Password unlocking, QQmagic variant.

After analysing these objects, the malware checks if there is a network connection. If it is possible to use network services (lines 2-9), the app transmits the randomly generated code, which will be used by the attacker to generate a deactivation code. If it is not possible to use SMS services (lines 12-18), the app employs a DES algorithm to decrypt a text provided in plain to be used as the encryption password, so the functionality of the app is guaranteed.

```
1 if (isNetworkConnected(getApplicationContext())){
2   if (this.share.getLong("m", 0) == 0){
3     this.editor.putLong("m", this.pass);
4     this.editor.commit();
5   }
6   try{
7     this.editor.putString("passw", this.des.
      encrypt("" + this.passw));
8     this.editor.commit();
9     this.ppss = this.share.getLong("m", 8) + "";
10  }
11  catch (Exception localException1){
12    try{
13      this.password = this.des.decrypt(this.share.
      getString("passw", ""));
14      new Thread(){
15        public void run(){
16          s.this.sms.sendTextMessage(s.h(...);}
17        }.start();
18        return;
19  }}}
```

Listing. 7. Deactivation code computation with no network connection in the QQmagic variant.

One of the common code snippets shared with other variants of Jisut is the class where the DES algorithm is implemented, which is identical among these variants. This algorithm is also used to decrypt the content received by SMS from the attacker:

As it can be seen in lines 9 and 10, a decryption object is invoked to transform two strings which are provided in plain text.

In addition, the *qqmagic* variant[11] goes one step further and implements the necessary code to actually carry out the

[11]Identified by SHA-256: 506f668438477b7476674957d14407d207de1f576e5c9de2852490b43a6a013b

```
1 public void onReceive(Context paramContext,
2   Intent paramIntent){
3   this.ds = new DesUtils(
4      "还想反编译你个傻逼没门");
5   StringBuilder localStringBuilder;
6   Object localObject1;
7   int i;
8   if (paramIntent.getAction().
      equals("android.provider.Telephony.
      SMS_RECEIVED")){{
9     ...
10    this.jj = this.ds.decrypt(
        "84f113ee155ba43f1e280b54401fffab");
11    this.jj1 = this.ds.decrypt(
        "84f113ee155ba43f1e280b54401fffab");
12    ...
13    ((Intent)localObject2).putExtra("nnr", new
      StringBuffer().append(this.jj).append("!").
      toString()) ...);
```

Listing. 8. SMS decryption in the QQmagic variant.

removal of all user files, if the ransom is not paid after a period of time. Nevertheless, the important enhancement found in this sample is the use of an advanced obfuscation software. The author employs Ijiami,[12] a tool for hard obfuscation based on collecting the code into compiled libraries of native code, where applying reverse engineering becomes a particularly tedious and time-consuming task. Unzipping the original apk file of this sample delivers the following tree:

```
/
├── AndroidManifest.xml
├── assets/
│   ├── ijm-x86.so
│   ├── libckey.so
│   ├── libckeygenerator.so
│   ├── libdalvik_patch.so
│   ├── libfilescanner.so
│   └── libh.so
├── classes.dex
├── META-INF/
├── res/
└── resources.arsc
```

The files highlighted in blue contain these compiled libraries which are loaded at runtime to build a new apk. In line 3, *ijm-x86.so* is loaded:

As shown in Listing 10, different folders are remounted with read and write permissions. Then the new apk is placed in the system apps folder (line 8) after giving the necessary access and execution permissions with the following procedure:

The use of this technique poses an additional challenge to the use of reverse engineering techniques. Although there are advanced techniques available to deal with obfuscated code, the use of this scheme by the ransomware is really effective

[12]http://www.ijiami.cn/

```
private void d(String paramString){
  paramString = new FileOutputStream(
    paramString);
  InputStream localInputStream =
    getAssets().open("ijm-x86.so");
  ...
}
protected void onCreate(
  Bundle paramBundle){
    ...
    d(this.path + "/zihao.l");
    ...
}
```

Listing. 9. Runtime libraries compilation, QQmagic variant.

```
void rootShell(){
  execCommand(new String[] {
    "mount -o rw,remount /system",
    "mount -o rw,remount /system/app",
    "cp /sdcard/zihao.l /system/app/",
    "chmod 777 /system/app/zihao.l",
    "mv /system/app/zihao.l
      /system/app/zihao.apk",
    "chmod 644 /system/app/zihao.apk",
    "reboot" }, true);
  }
```

Listing. 10. Allocation of access and execution permissions, QQmagic variant.

to make classical and specially static analysis tools almost pointless. For instance, if we observe static API calls by disassembling the app, we will not encounter any malicious behaviour since this is actually contained in separated compiled files. The only suspicious element here lies in invoking the call needed to load the external library. Nevertheless, this is a process which cannot be solely attributed to malicious code as many benign applications employ it to defend from piracy or due to other legitimate security reasons.

E. THE HONGYAN AND HUANMIE VARIANTS

These variants also resemble the SLocker family in some aspects (and in fact a few antivirus wrongly classify them as SLocker). They provide interesting implementation differences and show clearly the process whereby new subvariants are created. As in the case of the other variants analysed in this document, the procedure followed by this malware is quite simple: once the application has been installed and launched, it displays a screen with a Chinese message which falsely informs that the device configuration is being checked.

We have found two main versions of this variant, which we have called the Hongyan and the Huanmie versions (*color* and *disillusionment* in English) in reference to the package name. One of the most remarkable details of these variants is that we can explicitly observe the process by which a variant gets transformed into a new one. This process will be described at the end of this subsection.

The Hongyan version has been chosen for a deep analysis.[13]

1) APPLICATION ANALYSIS

After a few minutes, or if the app is closed and launched again, it shows the screen displayed in Fig. 1e, that reports that the user data has been encrypted and that it is necessary to contact whoever caused it by using the *QQ* messaging service. It also mentions the amount needed to unlock the files, which is 20 yuans (this small value was probably chosen to maximise the number of paying victims). The presentation screen also shows a large number, which is expected to be provided to the attacker when contacting him to obtain the deactivation key, for which a text field is provided below.

This version really encrypts data. We left a few decoy files with different extensions in the `/sdcard/` partition. When the app was launched, all files were immediately encrypted and the extension 文件已被幻灭劫持 was added to them (it varies between different samples of this variant). The ransomware does not make any distinction between file types, it encrypts any file whatever its format is.

Taking a look at the package folder tree helps identifying the different parts of this malware. The subpackage *Xbox* contains the encryption tools, with methods that call the algorithms implemented in the `javax.crypto` native library and some new methods that allow to convert between strings and bytes. The `com.android.admin.hongyan` includes the main code section of the app, including the main file `MainActivity.class` which is in charge of calling the necessary classes to launch the malicious payload.

```
/
├── Xbox
│   ├── AESUtils.class
│   ├── Base64Utils.class
│   ├── FileEncryptUtils.class
│   └── StreamGobbler.class
├── com.android.admin.hongyan
│   ├── BuildConfig.class
│   ├── MainActivity.class
│   ├── R.class
│   ├── des.class
│   ├── lock.class
│   ├── newone.class
│   └── 幻灭打造.class
├── it.sauronsoftware.base64
│   ├── Base64.class
│   ├── Base64InputStream.class
│   ├── Base64OutputStream.class
│   └── Shared.class.class
└── LogCatBroadcaster.class
```

Among the rest of files, `des.class` invokes the DES algorithm used to decrypt the text which will define the

[13]The sample chosen for this analysis is identified by SHA-256: 5212b6a8dd17ccfc60f671c82f45f4885e0abcc354da3d007746599f10340774

encryption key. *lock.class* contains the necessary code to calculate the key provided to the user in the screen, and checks whether the deactivation key introduced is correct. `newone.class` performs the user data encryption process, and also makes use of the code defined in *LogCatBroadcaster.class* to automatically reactivate the encryption process if it stopped. 红颜 implements the *SHA-1* and *MD5* hash functions. Finally `it.sauronsoftware.base64` implements some auxiliary functions to deal with data operations.

2) ENCRYPTION PROCESS

The encryption method employed in this malware is fairly straightforward. Using the `javax.crypto` built-in library of the Android API (see Listing 11), the app executes the AES algorithm over any user file.

```
1 paramString1 = new SecretKeySpec(toKey(Base64Utils
      .decode(paramString1)).getEncoded(), "AES")
2 localObject = Cipher.getInstance("AES");
3 ((Cipher)localObject).init(1, paramString1);
4 paramString1 = new CipherInputStream(paramString2,
       (Cipher)localObject);
```

Listing. 11. AES encryption in the Hongyan variant.

Since no parameters are provided in the algorithm call, the cipher configuration is provider specific. In Oracle Java JDK 7, the configuration used is AES + ECB + PKCS5Padding. According to the taxonomy described by Ahmadian *et al.* [8], this variant belongs to the private-key cryptosystem ransomware (PrCR).

The author tries to hide the encryption/decryption key in the code through a worthless obfuscation mechanism, consisting on several concatenated decryptions of a large text using a secondary decryption object whose key is coded in plain:

```
1 this.des = new des("红颜");
2 this.des = new des(this.des
    .decrypt("57e58af4e6039eaf"));}
3 this.key = this.des.decrypt(
    this.des.decrypt(this.des.decrypt(
    this.des.decrypt(this.des.decrypt(
    this.des.decrypt("LARGE_TEXT")))))));
```

Listing. 12. Encryption of decryption key, Hongyan variant.

As it can be observed in the first line, a *des* object is initialised using two Chinese characters. This object represents a DES encryption algorithm (newly implemented using `javax.crypto`) where the two characters are the encryption/decryption key. In the second line, this object is used to decrypt a 16 characters text, whose result is used to reinitialise the `des` object. However, this step is redundant and strangely useless, since the result obtained by the decryption of the 16 characters text is equivalent to the two previous Chinese characters, so it leads to the same argument and the decryption object remains identical.

In the third line, the encryption/decryption key is obtained applying the above mentioned *des* object to several nested decryptions of a large text provided in plain. This let us know the decryption key by just externally executing this piece of code. The resulting key is: "**GiEhjghmZIO7RTWyycQ9PQ==**". Although this key is different from the one that is expected to be introduced by the user to trigger the deactivation process, it allows a full recovery of every file, even when after the malware has been removed.

3) DEACTIVATION PROCEDURE EXAMINATION

A glance at the code level also allows us to reach all the necessary details to understand how both the key provided to the user and the deactivation key are generated. Although in most of the samples there are signs of the use of obfuscation techniques, the code can be easily untangled. First of all, a striking piece of code reveals (listing 13) that the app retrieves the IMEI number (line 1):

```
1 this.imei = ((TelephonyManager).getSystemService("
      phone")).getDeviceId();
2 paramBundle = 红颜一笑尽是伤.
      getsha_1(红颜一笑尽是伤.
      getMD5String(this.imei))
```

Listing. 13. IMEI code retrieval in the Hongyan variant.

In the next line (line 2), two hash functions are composed, taking as input the IMEI number. Thus, a variable saves the result of the SHA-1 of the MD5 of the IMEI, which is the value later displayed in the red ransomware screen. At this point, if the user provides this number to the attacker, he will send back the deactivation code.

In the same package class (named `lock.java` in most samples) we can also find the procedure to check whether the deactivation code inserted by the user is correct. It is simply a string comparison between the value inputted by the user and a transformation of the number provided to the attacker, based again on the the use of cryptographic hash functions:

```
1 lock.this.mm.getText().toString().equals(
    红颜一笑尽是伤.getsha_1(
    红颜一笑尽是伤.getMD5String(
    this.val$xx))))
```

Listing. 14. Deactivation code check in the Hongyan variant.

Actually, this new value is computed through a similar process to the one described before: it is the SHA-256 of the MD5 of the value given on the screen. In short, the key which deactivates the ransomware (and starts the decryption of user's data) is computed as:

$$SHA-1(MD5(SHA-256(MD5(IMEI)))) \quad (1)$$

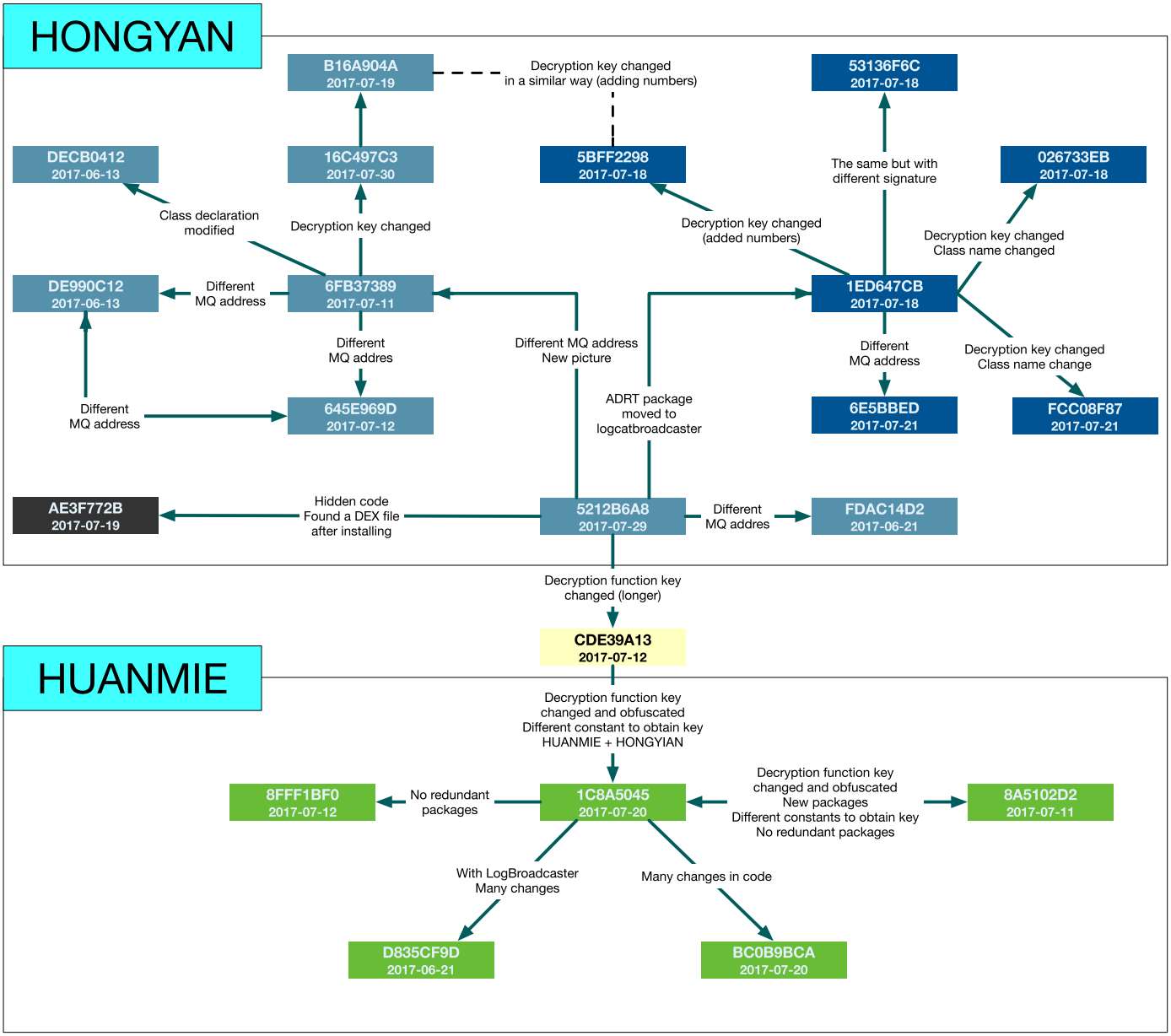


FIGURE 4. Different samples of the Hongyan and Huanmie variants of the Jisut ransomware. Each sample is identified by the first 8 characters of its SHA-256 hash.

When the user introduces this value and clicks on the *Decrypt* button, all the files are decrypted and the ransomware can be uninstalled.

4) VARIATIONS OF THIS VARIANT

The above analysis is intended to describe the particularities of the Jisut variant. However, after a long manual search through the VirusTotal Intelligence service, we have found multiple samples which implement a plethora of interesting but mostly minor changes. A comparative assessment of these samples allows us to evaluate how different modification were sequentially introduced. Fig. 4 shows the differences we found between a number of important samples of this variant. Each sample is represented by the first 8 hex characters of their SHA-256 signature.[14] The first submission date of the sample to the VirusTotal portal is also included.[15]

In general terms, we have found that the Hongyan version is the one which has led to most variations. The sample identified by *5212B6A8* in the diagram (the first 8 characters of the SHA-256 hash) has led to new samples with minor changes (as shown in the left part of the upper box) and to another set of applications where the *adrt* package has been extracted to include the `LogCatBroadcaster.class` as

[14]The complete signatures can be found at `http://aida.ii.uam.es/jisutnoransom/index.php/jisut-hashes/`

[15]This date does not represent when the sample was built or deployed, but when it was first uploaded to the VirusTotal portal. This is the reason of having samples in Fig. 4 shown as offspring of samples with a newer date

a new class (in the group of apps placed at the right of the box).

On the other hand, an important branch starts with sample *CDE39A13*. It can be seen as the first attempt to make the encryption key harder to retrieve, although the underwhelming implementation of this idea just consists on a bigger text needing to be decrypted in order to obtain said key. This sample leads to a new subset where substantial changes are included. For instance, within the code of sample *1C8A5045*, together with a lot of useless classes we can find again a clone of the previous versions under path `com.a.a.android.admin.hongyan`. But a new package has been added under `com.a.a.android.admin.huanmie`, which seems to be mostly a copy of previous ones with some modifications aimed to hinder attempts at reverse engineering. This is also a clear evidence of the evolution of malware, where an old version is taken to build a new and better one. Surprisingly, the encryption process remains identical so we can still easily decrypt every file with just a few lines of code.

In this sample, the main difference lies in the computation of the deactivation key:

```
1  int k=6000, n=1, i1=2, m=3;
2  int f157 = (n + i1) + m;
3  int f162 = f158 + 666;
4  int f161 = f162 + f157;
5  int f156 = ((f161 - f161) + f162)-(f157 + 666);
6  int f159 = (f161 * f156) + 666;
7  f158 = f159 + 666;
8  int f155 = f158 + f159;
9  int i = (f155 - f158) + k;
10 int j = i - k;
11 int key_decryption = ((i - j - k) + (m + n + i1 +
       1));
12 if (paramAnonymousView.equals(getsha_1(
       getMD5String(this.val$xx + key_decrypt))){
```

Listing. 15. Obfuscated key deactivation, Hongyan variant.

The above code was obtained using the *JADX* tool, although it produces some decompilation problems probably due to the use of Chinese characters. There are a number of computations which finally lead to a value which is concatenated to `this.val$xx`. While this last value is the same as the resulting from Equation 1, now it is concatenated with a new value computed by this confusing procedure. As the result of the decompilation process, there is one missing variable declaration, the one related to f158. It appears that the value of this variable is not relevant at all. When simplifying all the computations, the variables start to cancel each other out. The last variable key_decryption is:

$$((i - j - k) + (m + n + i1 + 1)) \tag{2}$$

Lets replace j, which is i-k:

$$i - i + k - k + m + n + i1 + 1 \tag{3}$$

The remaining variables are constants: m = 3, n = 1, i1 = 2. So:

$$\text{key\_decryption} = m + n + i1 + 1 = 7 \tag{4}$$

So, in the end, the new deactivation key is calculated in almost the same way as in the previously variant. The only real change involves the additional concatenation of a "7":

$$SHA - 1(MD5(SHA - 256(MD5(IMEI)) + \text{“7”})) \tag{5}$$

Finally, a more advanced variation (AE3F772B) was found, where the malicious payload is hidden following a procedure already taken by other ransomware. In this case, several files with an `.acc` extension contain the compiled code, which is loaded at runtime.

F. THE COM.BLL.APKIN VARIANT

This variant was first reported in 2017 by Lukas Stefanko [30] as a ransomware capable of talking to victims. Again primarily targeting Chinese users, this version asks for device administration privileges and informs the user that it is necessary to pay the ransom in order to unlock the device together, also displaying a classical locking screen stating the QQ number which the user must contact. The application lies in *MainActivity.class*, which is in charge of detecting when a key is pressed, and to launch a method which decrypts a text file. This file can be found under `assets/bll`, and contains a large seemingly random text.

```
/
├── resources.arsc
├── res/
├── META-INF/
├── classes.dex
├── classes-dex2jar.jar
├── assets/
│   └── bll
└── AndroidManifest.xml
```

The method initialises a large array with Chinese characters, building which seems to be a decryptor based on simple transformations. But this time, they are not totally useless. Instead, the file is read as a bytes array and passed as an argument to the *enorde()* object (line 18 in Listing 16), which is a decryption method previously initialised with the key *bll* (see line 3). The *enorde* class contains both an encryption and decryption method based on different transformation and bytes operations. When applied to the `bll` file, it results in a new text file which actually is a new apk. This new apk is saved in a file on the external storage directory (see line 6), and then it is read again (see line 11).

This new apk has been obfuscated using the Jiagu 360[16] tool, as the name of the compiled libraries suggest. Among the files found in this new apk, there are references to the *JavaMail* library, which indicates the use of mail services for communication.

[16]http://jiagu.360.cn/

```
public void jiemi(final String s){
  final enorde enorde = new enorde();
  com.bll.apkin.enorde.key = "bll";
  try {
    ...
    final String string = new StringBuffer().
    append(Environment.getExternalStorageDirectory
    ().getPath()).append(replace).toString();
    final InputStream open = this.getAssets().open
    (s);
    final byte[] array3 = new byte [open.available
    ()];
    open.read(array3);
    open.close();
    final FileOutputStream fileOutputStream = new
    FileOutputStream(string);
    fileOutputStream.write(enorde.decoder(array3))
    ;
    fileOutputStream.flush();
    fileOutputStream.close();
    this.apkin(new File(string));
  }
  catch (Exception ex) {
    ex.printStackTrace();
}}
```

Listing. 16. Hidden app recovering process, Hongyan variant.

V. DISCUSSION

As shown in the previous sections, the Jisut family has explored different modifications and refinements in order to improve its ability to lock users' devices and obtain a ransom from its victims. Although some of the techniques exposed do not entail a high degree of technical sophistication, they can be used to help in understanding the operation of the criminal group behind the ransomware, and possibly as well to establish authorship. Some of the later techniques reveal a higher degree of technical acumen, particularly those that dynamically load code. This, in our opinion, makes the use of dynamic analysis tools mandatory to deal with the most recent ransomware variants. We also believe the study performed in this work can have valuable didactic contents for anyone starting its journey in Android malware forensics. Furthermore, while we have focused on the Android platform, other environments such as iOS are not exempt from this kind of threat. Although in general malware exploits specific weaknesses of the target operating system, it is expected that many of the common patterns and techniques will be spread across platforms.

VI. CONCLUSION

The Jisut family can boast of a long and illustrious career infecting Android smartphones. The family has evolved in interesting ways to produce new variants, where both the graphics and technical details vary while the core of the ransomware is nearly identical. Throughout this paper we have analysed the most important variants of this ransomware, describing how they take control of the device and try to coerce the user to pay a ransom. We have described their encryption, deactivation and screen locking mechanisms, information that we hope will be useful for past, present and future victims. At the same time, we have also shown how these variants evolve and how past versions are taken as a template to build up new, more powerful and more complex variants.

The main objective of our work is to help not only victims and beginners in Android forensic and malware analysis, but also those interested in designing anti-malware tools. For this we provide them with a detailed characterisation of a currently active ransomware family. In our future work, we plan to extend the approach followed in this paper to analyse other Android malware families and to perform more detailed comparative assessments.

ACKNOWLEDGMENT

This work was supported in part by the Next Research Projects, such as the Comunidad Autónoma de Madrid under Grant S2013/ICE-3095 (CIBERDINE: Cybersecurity, Data and Risks), in part by the Spanish Ministry of Science and Education and Competitivity (MINECO), and in part by the European Regional Development Fund (FEDER) under Grant TIN2014-56494-C4-4-P (EphemeCH) and Grant TIN2017-85727-C4-3-P (DeepBio). This work was also funded by InnovateUK as part of the authenticatedSelf (aS) project, under reference number 102050, and partly sponsored by the ICT COST Action IC1403 Cryptacus in the EU Framework Horizon 2020.

This project has received funding from the European Unions Horizon 2020 research and innovation programme, under grant agreement No.700326 (RAMSES project). The authors also want to thank EPSRC for project EP/P011772/1, on the EconoMical, PsycHologicAl and Societal Impact of RanSomware (EMPHASIS), which additionally supported this work.

REFERENCES

[1] A. Martín, H. D. Menéndez, and D. Camacho, "MOCDroid: Multi-objective evolutionary classifier for Android malware detection," *Soft Comput.*, vol. 21, no. 24, pp. 7405–7415, 2017.

[2] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "DroidSieve: Fast and accurate classification of obfuscated Android malware," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, 2017, pp. 309–320.

[3] A. Martín, H. D. Menéndez, and D. Camacho, "Genetic boosting classification for malware detection," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jul. 2016, pp. 1030–1037.

[4] MalwareBytes. (2017). *2017 State of Malware Report*. [Online]. Available: https://kitedistribution.co.uk/wp-content/uploads/2017/03/StateofMalware_Report_final_PT.pdf

[5] G. Davis and R. Samani. (2018). McAfee mobile threat report Q1, 2018. McAfee. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2018.pdf

[6] A. Gazet, "Comparative analysis of various ransomware virii," *J. Comput. Virol.*, vol. 6, no. 1, pp. 77–90, 2010.

[7] K. Cabaj and W. Mazurczyk, "Using software-defined networking for ransomware mitigation: The case of cryptowall," *IEEE Netw.*, vol. 30, no. 6, pp. 14–20, Nov. 2016.

[8] M. M. Ahmadian, H. R. Shahriari, and S. M. Ghaffarian, "Connection-monitor & connection-breaker: A novel approach for prevention and detection of high survivable ransomwares," in *Proc. 12th Int. Iranian Soc. Cryptol. Conf. Inf. Secur. Cryptol. (ISCISC)*, Sep. 2015, pp. 79–84.

[9] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda, "Cutting the gordian knot: A look under the hood of ransomware attacks," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2015, pp. 3–24.

[10] J. Hamada. *Simplocker: First Confirmed File-Encrypting Ransomware for Android | Symantec Connect Community*. Accessed: Feb. 10, 2018. [Online]. Available: https://www.symantec.com/connect/blogs/simplocker-first-confirmed-file-encrypting-ransomware-android

[11] N. Chrysaidos. *Mobile Crypto-Ransomware Simplocker Now on Steroids*. Accessed: Sep. 1, 2018. [Online]. Available: https://blog.avast.com/2015/02/10/mobile-crypto-ransomware-simplocker-now-on-steroids/

[12] R. Lipovsky, L. Stefanko, and G. Branisa, "The rise of Android ransomware," White Paper, 2016.

[13] P. Zavarsky *et al.*, "Experimental analysis of ransomware on windows and Android platforms: Evolution and characterization," *Procedia Comput. Sci.*, vol. 94, pp. 465–472, Jan. 2016.

[14] R. Yu, "Ginmaster: A case study in Android malware," in *Proc. Virus Bull. Conf.*, 2013, pp. 92–104.

[15] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2012, pp. 95–109.

[16] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "ANDRUBIS—1,000,000 apps later: A view on current Android malware behaviors," in *Proc. 3rd Int. Workshop Building Anal. Datasets Gathering Exper. Returns Secur. (BADGERS)*, Sep. 2014, pp. 3–17.

[17] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis: Android malware under the magnifying glass," Vienna Univ. Technol., Vienna, Austria, Tech. Rep. TR-ISECLAB-0414-001, 2014.

[18] A. Martín, H. D. Menéndez, and D. Camacho, "String-based malware detection for Android environments," in *Proc. Int. Symp. Intell. Distrib. Comput.*, 2016, pp. 99–108.

[19] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, vol. 14, 2014, pp. 23–26.

[20] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh, and S. Malek, "Obfuscation-resilient, efficient, and accurate detection and family identification of Android malware," Dept. Comput. Sci., George Mason Univ., Fairfax, VA, USA, Tech. Rep., 2015.

[21] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-Sec: Deep learning in Android malware detection," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 371–372, 2014.

[22] N. Andronio, S. Zanero, and F. Maggi, "HelDroid: Dissecting and detecting mobile ransomware," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2015, pp. 382–404.

[23] A. Gharib and A. Ghorbani, "DNA-Droid: A real-time Android ransomware detection framework," in *Proc. Int. Conf. Netw. Syst. Secur.*, 2017, pp. 184–198.

[24] D. Maiorca, F. Mercaldo, G. Giacinto, C. A. Visaggio, and F. Martinelli, "R-PackDroid: API package-based characterization and detection of mobile ransomware," in *Proc. Symp. Appl. Comput.*, 2017, pp. 1718–1723.

[25] S. Song, B. Kim, and S. Lee, "The effective ransomware prevention technique using process monitoring on Android platform," *Mobile Inf. Syst.*, vol. 2016, Mar. 2016, Art. no. 2946735.

[26] T. Yang, Y. Yang, K. Qian, D. C.-T. Lo, Y. Qian, and L. Tao, "Automated detection and analysis for Android ransomware," in *Proc. IEEE 17th Int. Conf. High Perform. Comput. Commun. (HPCC), IEEE 7th Int. Symp. Cyberspace Saf. Secur. (CSS), IEEE 12th Int. Conf. Embedded Softw. Syst. (ICESS)*, Aug. 2015, pp. 1338–1343.

[27] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current Android malware," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2017, pp. 252–276.

[28] F. Maggi, A. Bellini, G. Salvaneschi, and S. Zanero, "Finding non-trivial malware naming inconsistencies," in *Proc. Int. Conf. Inf. Syst. Secur.*, 2011, pp. 144–159.

[29] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVCLASS: A tool for massive malware labeling," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*, 2016, pp. 230–253.

[30] F-Secure Labs. *Trojan: Android/SLocker Description*. Accessed: Feb. 10, 2018. [Online]. Available: https://www.f-secure.com/v-descs/trojan_android_slocker.shtml

ALEJANDRO MARTÍN received the B.Sc. degree in computer science from the Universidad Carlos III de Madrid in 2014, and the M.Sc. degree in computer science and technology from the Universidad Carlos III de Madrid in 2015. He is currently pursuing the Ph.D. degree with the Autonomous University of Madrid, where he is also involved with the AIDA Research Group. His main research interests are related to machine learning and cybersecurity, focused on malware detection and classification problems.

JULIO HERNANDEZ-CASTRO was with the University of Portsmouth, U.K., and Carlos III University, Spain. He is also affiliated with the Kent Cybersecurity Center. He is currently a Professor of computer security with the School of Computing, University of Kent. His research interests are wide, covering from RFID security to lightweight cryptography, including steganography and steganalysis and the design and analysis of CAPTCHAs. He has been a Pre-Doctoral Marie Curie Fellow and also a Post-Doctoral INRIA Fellow. He is currently the Vice-Chair of the EU COST Project CRYPTACUS. He receives research funding from InnovateUK Project aS, EPSRC Project 13375, and EU H2020 Project RAMSES.

DAVID CAMACHO received the Ph.D. degree in computer science from the Universidad Carlos III de Madrid in 2001, and the B.S. degree in physics from the Universidad Complutense de Madrid in 1994. He is currently an Associate Professor with the Computer Science Department, Universidad Autonoma de Madrid, Spain, where he is the Head of the Applied Intelligence and Data Analysis Group. He has published over 250 journals, books, and conference papers. His research interests include data mining (clustering), evolutionary computation (GA, GP), multi-agent systems and swarm intelligence (ant colonies), automated planning and machine learning, or video games among others. He receives research funding from the Spanish Ministry of Science and Education and Competitivity (EphemeCH and Deepbio), and from the EU (Justice, ISFP, Erasmus+, and H2020).

PUBLICATION 2

Android malware detection through hybrid features fusion and ensemble classifiers: the AndroPyTool framework and the OmniDroid dataset

(IJ-2) **Martín, Alejandro**; Raúl Lara-Cabrera & David Camacho: "Android malware detection through hybrid features fusion and ensemble classifiers: the AndroPyTool framework and the OmniDroid dataset." *Information Fusion*, Ed. By Elsevier. Accepted. DOI: `10.1016/j.inffus.2018.12.006`
Impact factor = 6.639 (JCR, 2017) [Q3, 4/103 Computer Science, Theory & Methods].

- **Contributions of the PhD candidate:**
  - First author of the article.
  - Contributions made in the conception of the presented idea.
  - Contributions made in the design of the tool and dataset presented.
  - Implementation of the tool presented.
  - Generation of the dataset proposed.
  - Design and execution of the experiments.
  - Co-author of the interpretation and discussion of results provided.
  - Co-author of the manuscript, figures and tables presented.

Android malware detection through hybrid features fusion and ensemble classifiers: the AndroPyTool framework and the OmniDroid dataset

Alejandro Martín[a], Raúl Lara-Cabrera[b], David Camacho[a,*]

[a] *Computer Science Department*
Universidad Autónoma de Madrid, 28049 Spain
[b] *Departamento de Sistemas Informáticos*
Universidad Politécnica de Madrid, 28031 Spain

Abstract

Cybersecurity has become a major concern for society, mainly motivated by the increasing number of cyber attacks and the wide range of targeted objectives. Due to the popularity of smartphones and tablets, Android devices are considered an entry point in many attack vectors. Malware applications are among the most used tactics and tools to perpetrate a cyber attack, so it is critical to study new ways of detecting them. In these detection mechanisms, machine learning has been used to build classifiers that are effective in discerning if an application is malware or benignware. However, training such classifiers require big amounts of labelled data which, in this context, consist of categorised malware and benignware Android applications represented by a set of features able to describe their behaviour. For that purpose, in this paper we present *OmniDroid*, a large and comprehensive dataset of features extracted from 22,000 real malware and goodware samples, aiming to help anti-malware tools creators and researchers when improving, or developing, new mechanisms and tools for Android malware detection. Furthermore, the characteristics of the dataset make it suitable to be used as a benchmark dataset to test classification and clustering algorithms or new representation techniques, among others. The dataset has been released under a *Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License* and was built using AndroPyTool, our automated framework for dynamic and static analysis of Android applications. Finally, we test a set of ensemble classifiers over this dataset and propose a malware detection approach based on the fusion of static and dynamic features through the combination of ensemble classifiers. The experimental results show the feasibility and potential usability (for the machine learning, soft computing and cyber security communities) of our automated framework and the publicly available dataset.

Keywords: Malware analysis, Android, Hybrid features fusion, Malware dataset

1. Introduction

Due to the presence of technology in all areas of our daily lives, cyber security has become one of the main concerns to be addressed by the society as a whole. In recent years, there has been a large number of attacks and, what is even more remarkable, to a wide variety of objectives. Some recent well-known examples include denial of service attacks such as that performed by the Mirai botnet [1] and a massive data hijacking led by the ransom-ware Wannacry [2].

Furthermore, mobile devices are everywhere nowadays due to their popularity. Even in big companies this has been noticed, thus implementing new policies such as BYOD (Bring Your Own Device) and increasing the number of telecommuting employees. But, on the other hand, this blurs the perimeter security even more in these companies. Mobile devices can be considered as an entry point in any attack vector since the security measures are not so developed as in PCs. Hence, it is required to research new techniques for the automatic detection of malware for mobile devices, especially those that use Android operating system, since it represents over the 80% of the market share compared to iOS (around 15%), according to the Worldwide Quarterly Mobile Phone Tracker [3].

Cyber attacks manage to produce unprecedented levels of disruption, where attackers usually leverage diverse tools and tactics, such as zero-day vulnerabilities and malware [4]. This situation makes malware detection techniques worth studying and improving, in order to prevent and/or mitigate the effects of cyber attacks. Machine learning techniques can help to satisfy this demand, building classifiers that discern whether a precise Android application is malware or benignware. Algorithms such as Decision Trees [5], Support-Vector Machines [6] and Naive Bayes [7], to name a few, are able to build such classifiers. Going further, ensemble methods for machine learning [8]

* *Corresponding author*
Email addresses: alejandro.martin@uam.es (Alejandro Martín), raul.lara@upm.es (Raúl Lara-Cabrera), david.camacho@uam.es (David Camacho)

30th January 2019

Accepted Manuscript
To appear in: Information Fusion (Elsevier)
DOI: https://doi.org/10.1016/j.inffus.2018.12.006

aim at effectively integrating many kinds of classification methods and learners to benefit from each ones advantages and overcome their individual drawbacks, hence improving the overall performance of the classification.

Nevertheless, machine learning techniques require large datasets of representative features extracted from real samples, which defines one of the goals of this paper: to present a comprehensive dataset of *dynamic* and *static* features from Android applications called *OmniDroid*. This dataset can help other researchers to improve and develop new automatic malware detection techniques for Android devices. At the same time, the characteristics of this dataset make it suitable to be employed as a benchmark dataset to apply and test different algorithms and techniques, such as classification, clustering, association rule learning, pattern recognition or even to test representation learning algorithms. In order to make the *OmniDroid* dataset easy to use, all the data are provided in JSON and CSV formats and are publicly available.[1]

The *OmniDroid* dataset has been built using *AndroPyTool* [9, 10], an automated open source tool for dynamic and static analysis of Android applications. Hence, another goal of this paper is to present *AndroPyTool* in depth and to describe its functionality.

The main contributions of this paper can be summarised as follows:

- *AndroPyTool*, a new tool for the automatic extraction of both static and dynamic features from sets of Android applications, is presented.
- The aforementioned tool has been used to generate a dataset (called *OmniDroid*) of static and dynamic features extracted from Android benignware and malware samples. The dataset is publicly available under a *Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License* [11].
- A thorough analysis showing statistics regarding the differences found between the two categories of samples and among the samples belonging to the same set.
- The performance of several ensemble classifiers from the state of the art have been studied to assess the feasibility of the features selected to build new detection or classification models based on ensemble techniques. It also illustrates the potential advantages of using the tool and the dataset when building ensemble methods to detect and classify Android malware.
- Finally, an Android malware detection approach, based on the fusion of static and dynamic features through the combination of an ensemble of classifiers following a voting scheme, is presented.

[1] `https://aida.ii.uam.es/datasets/`

2. Basics on Android malware detection

Malware, in its different forms, represents a major issue affecting different platforms, from personal computers to smartphones or to the Internet of Things (IoT). From the appearance of the first virus designed for computers in the 70's, trojans, viruses or spyware, among others, have been developing different shapes to deal with the countermeasures imposed by operating systems and anti-virus engines. This has led to a race where security experts are always pursuing black hat hackers. To tackle this problem, malware detection tools try to extract and model the behaviour of a suspicious sample, which is compared against benign or malicious patterns in order to make a decision. This section describes the most used features capable of representing these behaviours and how they are used in the literature.

2.1. Basic features for Android malware analysis

In order to model the behaviour of a benign or malicious app, it is required to establish a representation able to describe in depth its actions and purposes. Two approaches to conduct this modelling process are possible: a static or a dynamic analysis, presenting different procedures to be performed, at the same time that entail a series of advantages and disadvantages. In order to characterise the behaviour of a given sample following static analysis techniques, it is possible to inspect the package, to decompile the code or to access the different files contained in the package (i.e. the Android Manifest). This allows to gather a set of relevant and useful features, such as a list of API calls invoked throughout the code or the set of Android permissions required in order to deploy the whole functionality of the sample.

In contrast, dynamic analysis opts for capturing the actions that are actually triggered by the suspicious sample, leveraging an emulator or even a physical device to run the app while a monitoring agent captures a series of indicators, such as hardware components accessed, network traffic or system calls invoked. One of the main benefits of following a dynamic approach lies in its ability to capture events invoked in obfuscated sections of code or included in code dynamically loaded. In these two examples, static analysis faces a major barrier.

Detection and classification tools leverage groups of specific features, whether static, dynamic or combinations of both, through different representation techniques, such as histograms, graphs or Markov models. This subsection describes the most employed and cited features in the literature and how they are handled when building detection and classification tools. A summary of the different state-of-the-art approaches is provided in Table 1.

API calls are within the most used features. Whether static or dynamically extracted, they allow to model the behaviour of a sample and to characterise the actions it can take. For instance, DroidMat [24] performs API Calls

Detection/ classification method	API calls	Files access	Intents	Permissions	Network data	Hardware	Native Code	Hidden files/ dynamic loading	ICC	Metainf.	Opcodes	System commands	Strings	Taint analysis
ADROIT[12]				✓						✓				
Andro-Dumpsys[13]	✓		✓	✓								✓		
AndroDialysis[14]			✓	✓										
Andromaly[15]					✓	✓								
Apposcopy[16]	✓													✓
Dendroid[17]											✓			
DREBIN[18]			✓	✓		✓				✓				
DroidAnalytics[19]	✓							✓			✓			
DroidAPIMiner[20]	✓													
DroidDet[21]	✓		✓	✓										
DroidFusion[22]	✓		✓	✓				✓				✓		
DroidLegacy[23]	✓													
DroidMat[24]	✓		✓	✓		✓			✓					
DroidMiner[25]	✓		✓	✓										
DroidScribe[26]		✓			✓			✓	✓					
DroidSieve[27]	✓		✓	✓			✓	✓		✓	✓		✓	
Gascon et al.[28]	✓													
ICCDetector[29]			✓						✓	✓				
Madam[30]	✓	✓		✓						✓				
Manilyzer[31]			✓							✓				
Marvin[32]	✓	✓	✓	✓	✓			✓		✓				
MAST[33]			✓	✓			✓	✓						
Peiravian et al.[34]	✓			✓										
RevealDroid[35]	✓		✓											✓
Sheen et al.[36]	✓			✓										
Wang et al.[37]	✓		✓	✓		✓								
Yerima et al.[38]	✓			✓				✓				✓		
Yerima et al.[39]	✓			✓								✓		
Zhang et al.[40]	✓													✓
OmniDroid	✓	✓	✓	✓	✓			✓		✓	✓	✓	✓	✓

Table 1: Relation of features extracted and used by different Android malware detection and classification state-of-the-art approaches. The bottom lines indicates the set of features that the presented dataset OmniDroid contains. In the case of the detection of files accessed, the use of dynamic code loading techniques or information related to network connections, these are extracted during the dynamic analysis performed with DroidBox.

tracing from the different app components. DroidAPIMiner [20] also focuses on such API calls that are considered critical by the authors and include the arguments used when the call is invoked. API calls have also been used in the form of call graphs [28].

A relation of the Intents declared by an application can also be deemed as a key information source to categorise the behaviour of a sample. As in the case of API calls, a list of actions defined by Intents can be static or dynamically extracted. There are many examples in the literature studying this feature. In MAST [33], different indicators of the application functionality, including Intent actions, are analysed based on the assumption that these factors differ from benign samples. This feature has also been used to group malicious samples based on similarity [13]. For further information, the effectiveness of Intent actions for revealing malicious behavioural patterns has been evaluated by Feizollah et al. [14].

Android permissions are one of the most relevant characteristics in the Android malware detection scenario, even though they cannot offer a detailed description of the intentions that a suspicious application could take. Malware detection models based on machine learning algorithms have been trained with datasets of permissions combined with API calls [34]. In combination with many other features, such as filtered intents, network information or data regarding the use of hardware components, Support Vector Machines are trained in Drebin [18] for malware family classification. With a special focus on detecting obfuscated malware, DroidSieve [27] combines permissions with a list of invoked components or API calls, among others.

From a more general point of view, other methods base their malware detection mechanisms solely on meta-information provided by the developers. Such is the case of ADROIT [12], a system that trains machine learning algorithms with a set of features extracted from the Android Manifest and performs a text mining process on the description text of the application. Another example is Manilyzer [31], focused on using the information which can be gathered from the Android Manifest files to train machine learning algorithms.

Other possibilities to face this problem, involving statically features extracted as well, are based on the employment of opcodes or system commands. Droid Analytics [19] is a system aimed at assisting to retrieve opcode level information from Android malware. A study on the effectiveness of opcodes for malware family classification can also be found in the literature [41]. System commands

have been used in combination with a parallel machine learning classifier [39] or Bayesian classification [38]. Another possibility is to analyse the information transmitted through the Inter-Component Communications (ICC) service, which can be used by malware to perform malicious actions such as installing a new application [29].

Within the set of features under a static analysis approach, *taint analysis* has also served for designing detection and classification tools. FlowDroid [42] can be considered as the most important exponent of this kind of analysis. RevealDroid [35] is an accurate and obfuscation resilient Android malware detection tool where information flows extracted with FlowDroid play a fundamental role to detect malicious patterns. Another example of the use of taint analysis is Apposcopy [16].

All the features described so far are mainly extracted through the use of static analysis techniques, what is to say, they do not imply code execution. While static features are easy to extract and provide detailed information, they present some inherent shortcomings. One of the most important examples is the inability to trace the actual behaviour of a sample when running in a real device, which prevents, in many cases, to reveal the malicious payload of a malware instance. There is an important number of researches trying to bypass this problem by monitoring the application actions in an emulator. One example is Andromaly [15], which applies machine learning to varied metrics of the system behaviour collected in runtime. Another instance of this kind of analysis is DroidScribe [26], which monitors API calls invoked in runtime to detect Android malware.

2.2. Malware detection tools and classification techniques

It is possible to find in the literature several tools aimed at detecting Android malware, as well as classifying and categorising them among families that share certain features.

For instance, RevealDroid [35] is both a malware classification and detection tool which employs a varied set of static features including sensitive API calls, API packages, API flows obtained with FlowDroid [42] and a list of actions of the Intents. Then, a C4.5 decision-tree and a 1-Nearest-Neighbour (1NN) machine learning algorithms are trained, taking as input all these features extracted from Android benign and malware samples belonging to different families. DroidSIFT [40] is a semantic-based classifier, which uses API dependency graphs to train a Naïve Bayes classifier. In Dendroid [17], code structures serve to compare samples and to train both a classification and a clustering algorithm. Drebin [18] follows the same pattern, it extracts a large of features including hardware components, requested permissions, app components, filtered intents, restricted API calls, used permissions, suspicious API calls and network addresses to train Support Vector Machines.

DroidMiner [25] builds behaviour graphs called Component Behaviour Graphs (CBG) based on the communication between API functions and sensitive Android resources and different classification algorithms such as SVMs or Random Forest are trained. DroidAPIMiner [20] extracts API calls based on their presence in malicious samples and are used to perform, as in previous examples, a training process of different machine learning algorithms. DroidLegacy [23] performs family classification by extracting signatures that can be found in malicious applications and that can be used in repackaged, originally benign, samples. Then, similarity measures are used to compare samples. MAST [33] helps in the triaging process making use of features extracted from the application package. Finally, MOCDROID [43] uses an evolutionary approach and third party calls to perform malware detection.

From the classification model perspective, a broad range of techniques have been used in this scenario. Among these, ensemble based classifiers have been repeatedly employed showing a good behaviour. These methods have proved to be powerful in imbalanced problems [44] as well as in binary and multi-class domains [45]. For instance, statically extracted features (including permissions, API calls and a set of system events) are used to feed a rotation forest model [21], which improves the results when compared against a classic Support Vector Machine classifier. A blend of API calls and permissions also defines the set of features employed to train a multi-feature collaborative decision fusion (MCDF) [36], through a pool of classifiers including J48, Random Tree and Decision Stump.

In DroidFusion [22], the classification is performed at two different levels. First, several low level classifiers are trained, involving Random Tree, REPTree, J48, Voted Perceptron and also ensemble methods. Then, their results are combined to define the final output using four proposed ranking-based methods. In a similar approach [37], 11 different static features categories are filtered using a SVM classifier and then are sent to the input of an ensemble classifier composed of five models: SVM, Random Forest, K-Nearest Neighbors, Classification and Regression Tree (CART) and Naive Bayes. These models are combined through a majority voting scheme.

All previous approaches, no matter the algorithm employed, require datasets of features extracted from malware and harmless samples in order to train their models.

2.3. Android malware datasets

Machine learning based malware detection tools require datasets of samples labelled as malware or benign to be trained and tested. The use of these datasets is essential in order to build reliable malware detection or classification methods. For that purpose, different datasets of Android samples have been made public over the past years. The Android Malware Genome Project [46] was launched in 2012, however, it was abandoned in 2015. Drebin [18] contains malware samples labelled as different malware families, but it is too old, since the samples

were gathered from August 2010 to October 2012. Contagio [47] periodically publishes new malware applications found in the wild. The AndroZoo project [48] offers a huge dataset containing more than 5,700,000 samples which are currently being analysed with different antivirus engines. Other source where it is possible to find Android malware families is the *android-malware* GitHub repository [49] which includes APKs of different varieties. More recently, the Android Malware Dataset [50] was released. It contains 24,553 samples gathered from 2010 to 2016 of 71 malware families.

In addition to datasets, there are also online services that make it possible to retrieve both benign and malicious applications. For instance, many researchers collect samples directly from application stores, such as Google Play or Aptoide, specially when searching for benignware. On the other hand, malware can be obtained from online applications such as the VirusTotal Intelligence service[2].

While these datasets offer sets of raw samples, other projects provide features or logs already extracted from samples. In this line, DroidCat [51, 52] is composed of 440 and 508 logs of malicious and benign samples, respectively. AndroMalShare [53] also provides a dataset of features extracted from malware samples, including static and dynamic analysis. If compared to *OmniDroid*, AndroMalshare only includes malware samples and the features set is not as broad. The Kharon project [54] provides a reduced set of Android malware samples of different families and includes some technical implementation details. The Koodous portal[3] offers a huge dataset of malicious and benign samples only for research purposes, in this case including a report of features. In contrast, *OmniDroid* provides a wider set of features. Androzoo [48] provides a large set of Android Apks for research purposes.

In this work, samples provided by Koodous and Androzoo were used as the starting point in order to analyse a large set of samples considered as malware and considered as harmless.

2.4. Android malware analysis tools

To day there exist a plethora of available tools for the analysis of suspicious Android applications aimed at extracting different types of characteristics. For instance, a GitHub repository [55] offers an overview of the existing tools and links to useful resources. In particular, there is an important number of utilities focused on reverse engineering processes able to extract groups of features. For example, AndroGuard [56] is a Python library which extracts varied information from code, resources or the Android Manifest.

Other well-known tool is smali/backsmali [57], an assembler and disassembler of the DEX files which contain the bytecode of the application. Apktool [58] allows to decode the resources contained in the executable file. It also provides other powerful utilities such as repacking the sample. Dex2jar [59] is aimed at converting Dalvik bytecode files into Java compiled files with extension *jar* extension. This allows to later decompile the code using other tools such as jd-gui [60], which reconstructs the code and allows to visualise it. A similar functionality is provided by jadx [61]. FlowDroid [42] performs taint analysis over Android applications, providing useful and detailed information related to the different information flow which occur during the application lifecycle.

From the dynamic analysis perspective, DroidBox [62] enables to monitor a wide series of events such as accesses to files, network traffic or DEX files dynamically loaded in run time. An alternative to DroidBox is CuckooDroid [63]. The greatest weakness of both tools lies in that they are implemented for old versions of the Android platform.

3. AndroPyTool

Currently, there is a plethora of tools for the extraction of static and dynamic features from Android applications. However, each one focuses on a precise kind of features, so obtaining a comprehensive set of both static and dynamic features becomes an Herculean task. It involves setting up every tool with their respective configuration files and environment, as well as grouping the results obtained individually to build a complete data set. That is the motivation behind the development of AndroPyTool, in other words, to offer an all-in-one solution to the Android malware research community capable of performing a full feature extraction process from suspicious samples.

AndroPyTool is an integrated framework developed in Python aimed at obtaining varied dynamic and static features from a set of Android applications. It embeds the most used Android malware analysis tools, performs inspection on the source code and retrieves behaviour information when the sample is run within a controlled environment. The tool provides a detailed report for every analysed application, including a large batch of fine-grained representative characteristics.

Any interested researcher can get involved in the project by making improvements or fixing bugs. The source code of the project is hosted at GitHub [10]. Furthermore, AndroPyTool can be used through a Docker container, which allows to easily run the tool in just two steps. Refer to the source code repository to get a full description of the arguments and to know how to install and launch the tool without using the Docker image (i.e. using the source code).

This section comprises a description of the aforementioned tool, detailing what are the computed features.

3.1. Tool operation

As an integrated framework, AndroPyTool includes several Android analysis and processing tools to provide

[2] `https://www.virustotal.com/intelligence`
[3] `https://koodous.com/`

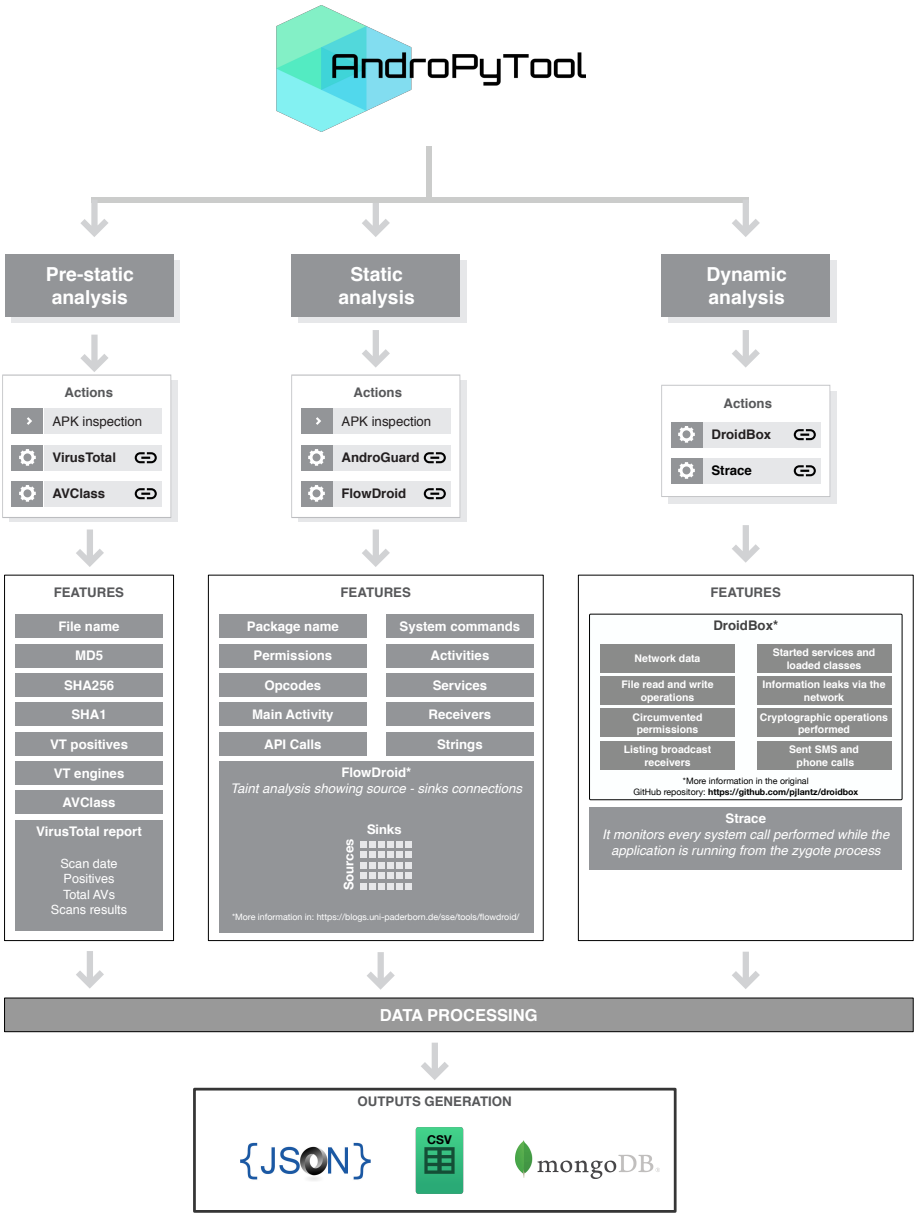


Figure 1: AndroPyTool feature extraction process. The application follows a seven steps process in which static and dynamic analysis tools are executed (steps 1–6), to finally process and group these results into an unified dataset (step 7).

detailed reports on the features and behaviour of applications. To achieve this, every application file (namely *apk*) follows a pipeline comprising seven steps (see Figure 1), which are detailed below:

1. **APK filtering**: The first step aims at inspecting every sample by using the AndroGuard tool [56] to determine whether the sample is a valid Android application.
2. **Virustotal analysis**: the tool retrieves a report of the application from the Virustotal online web application. The report contains the results from the scan performed by Virustotal as well as resulting data from the analysis of the application by more than 60 distinct anti-malware engines.
3. **Dataset partitioning**: in this step, which is optional, each sample is labelled as malware if tagged this way by at least ϵ antivirus based on the VirusTotal report. This criteria can be modified by the final user of the tool, who can establish his/her own threshold ϵ.
4. **FlowDroid execution**: this tool, based on taint analysis, is run against every sample.
5. **FlowDroid results processing**: there is a processing step of those results provided by FlowDroid,

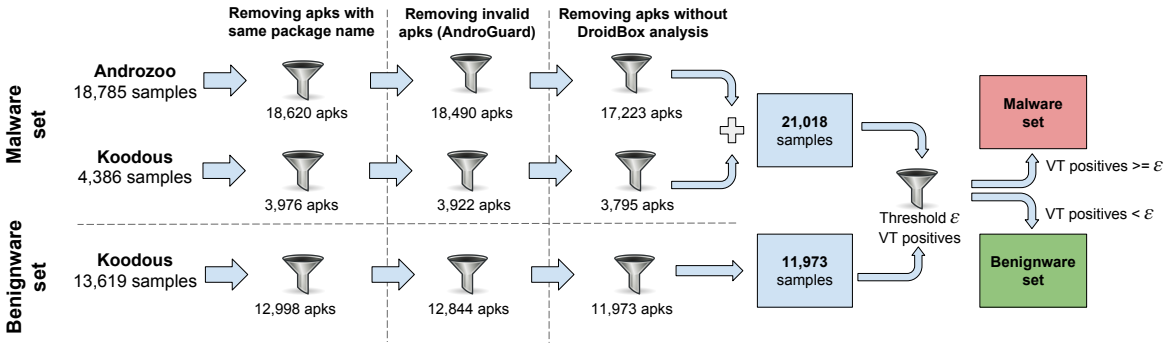


Figure 2: Filtering processes applied to build the OmniDroid dataset. The final allocation of benignware and malware samples depends on the ϵ parameter, which defines the minimum number of antivirus which test positive for malware in order to consider a sample as malicious or benign.

in order to extract the connections between sources and sinks. The reason of dividing the extraction and processing of information flows lies in that different representations are possible, thus allowing to modify the processing step independently.

6. **DroidBox execution**: a customised version of DroidBox [62], an Android dynamic analysis tool, that includes the *Strace* tool is run against the sample in this step.
7. **Feature extraction**: in this step, the tool gathers, reorganise and structure the results as reports of extracted features. The combined dataset of features extracted for all applications is provided in the following formats: as a comma-separated value (CSV), as a JavaScript Object Notation file (JSON), and as a MongoDB database.

Next section provides a description of the Omnidroid dataset, including the features extracted by the tool for every given Android application along the aforementioned process.

4. Dataset description

The OmniDroid dataset was built using AndroPyTool [10]. With this tool, a large set of samples from two different sources were analysed. In the first place, both benign and malicious samples were gathered from a dataset of 100,000 samples that the Koodous[4] Team kindly gave us for research purposes. Additional samples from the AndroZoo[5] portal have been included to supplement the first set and also to promote variety within the malware set of samples, thus avoiding potential sources of bias (virus creators, creation date, etc.).

A filtering process was followed in order to remove repeated applications and those that are considered as invalid (they cannot be actually installed and executed). As it can be seen in Fig. 2, a first filter consists of removing those samples that are represented by repeated packages names, thus avoiding several instances of the same application and their related feature vector. Then, invalid applications were detected with the AndroGuard tool [56] included into AndroPyTool in order to remove apps that cannot be executed. The third step pursues the same objective, but in this case discarding samples that could not be actually executed in the Android emulator used by DroidBox. All samples define a minimum SDK version under API 16 in their Android Manifest, ensuring that this is not a disadvantage for the use of DroidBox, which uses this API level.

AndroPyTool run on the different sets of samples until a considerable number of samples was analysed: 21,018 malware samples and 11,973 benign samples. Although the Koodous dataset already makes a distinction between malicious and non-malicious samples, all samples (including those obtained from AndroZoo) were submitted to VirusTotal. This allows to obtain an updated report for every application which includes the scan result obtained from a series of antivirus engines and which can be used to label each sample as malware or benignware. Thus, the rate of positives delivered by the antivirus engines implemented by the VirusTotal portal is included as pre-static information for each sample. Since a low rate of antivirus reporting malicious content could be due to false positives, the final allocation of samples to a malware or benignware set will be in hands of the users of the OmniDroid dataset, who can establish their own criteria. In this line, a procedure such as the one implemented by AndroPyTool can be used, where the label of each sample is set according to a threshold ϵ.

Due to the high computational load required to obtain the full analysis from each sample, the finally built dataset contains a subset of the applications gathered. This reduction was addressed aiming to keep a balanced dataset containing both malware and benignware samples. For that purpose, the threshold parameter ϵ was set to 1. According to this criteria, the finally generated and published

[4] https://koodous.com/
[5] https://androzoo.uni.lu/

dataset is composed by 11,000 malware and 11,000 benignware samples.

All the features present in this dataset have been extracted from the aforementioned set of applications by executing AndroPyTool. A general overview of the number of features extracted can be depicted in Table 2, where the most important groups of characteristics are represented. Table 1 compares the features contained in the OmniDroid dataset against the characteristics used by different methods proposed for Android malware detection and classification.

Feature	Count	Feature	Count
Permissions	5,501	Services	4,365
Opcodes	224	Receivers	6,415
API calls	2,129	API Packages	212
System commands	103	FlowDroid	961
Activities	6,089		

Table 2: Number of the most important features included in the OmniDroid dataset by category.

The next subsections describe each of the features that are provided for each sample.

4.1. Features extracted through pre-static analysis

This entry provides general information about the application, such as its file name, MD5, SHA-1 and SHA-256 checksums, a field reporting the number of positives in its VirusTotal report and the total number of antivirus engines from which scan results were obtained. Although most of these elements cannot be actually considered as behavioural features, these fields can be used to keep track of the APK for any further analysis (see Table 3). This kind of features also includes a categorisation of the sample according to the *AVClass* [54] tool, which aims to achieve a consensus between the outputs given by the different antivirus engines run by VirusTotal (see Section 4.4). This tool starts from a set of labelled samples to extract a list of tokens, detects alias of the same family, applies several filters and reveals the most convenient token for each particular sample.

4.2. Features extracted through static analysis

Once the pre-static features are obtained, a set of new features, which are extracted using static analysis techniques (see Table 4), is gathered from the samples and incorporated into the dataset. These features are: package name, permissions, opcodes, main activity's name, API calls, strings, system commands, and a list of intents whose activities, services and receivers are able to manage independently. All applications have been analysed with the *Flowdroid* [42] tool as well, so these reports are included into *OmniDroid*.

The main goal of this set of features is to provide an insight of the application expected behaviour and the range

Feature	Description
`Filename`	Filename of the APK
`VT_positives`	Number of antivirus which test positive for malware
`VT_engines`	Number of antivirus used in the analysis
`AVClass`	Agreed malware label from several detection engines according to [54]
`md5`	MD5 checksum of the APK
`sha1`	SHA-1 checksum of the APK
`sha256`	SHA-256 checksum of the APK

Table 3: Pre-static features available in the dataset.

Feature	Description
`API calls`	Count of system calls performed by an APK
`Main activity`	Name of the Main Activity
`Opcodes`	Count of opcodes performed by an APK
`Package name`	Name of the package
`Permissions`	Which permissions uses the APK
`Intent receivers`	Set of an APK's receivers
`Intent services`	Services used by an application
`Intent activities`	Activities declared by an APK
`Strings`	Set of defined strings (with use count) within an APK
`System commands`	Set of system commands ran by the app
`FlowDroid`	Path to the results obtained by FlowDroid [42]

Table 4: Static analysis features available in the dataset.

of actions that it could take based on a static analysis of the code which does not imply code execution. While this kind of analysis cannot reflect the real behaviour of the sample, which will only be revealed if the sample is run, it feeds the analysis report of a sample with valuable information. For instance, a list of permissions required offers a general picture of the range of actions that the sample could take. Furthermore, this information can also be used to compare declared and expected (static analysis) behaviour of a sample with the actually performed (dynamic analysis).

In the first place, a complete list of API calls found in the code is provided. It is important to note that those calls that are invoked through reflection or dynamic code loading among other obfuscation techniques are not detected by this kind of analysis. In the OmniDroid dataset, API calls can be found grouped by the class in which they are defined or by their package[6]. Secondly, permis-

[6]A list of all Android API packages can be found at `https://developer.android.com/reference/packages.html`

sions show device functionalities that the application can use, and system commands offer an overview of the actions that can occur at a low level, so they can reveal interesting actions performed, such as privilege escalation.

Other static features include Dalvik opcodes, obtained by analysing the Dalvik bytecode and that are useful to discern the behaviour of the sample at a low level. At the same time, a list of intents that are used to invoke other application components allows to make a profile of the sample based on actions performed and the events that trigger these actions. Another kind of feature included are strings which are found within the code and that could contain sections of code prepared to be executed in run time.

Finally, a report built using the FlowDroid tool is also included in this section. This is a useful tool that performs taint analysis over the application code, in order to discover connections between a source and a sink. These sources and sinks, which have been previously defined by the SuSi framework [64], allow to model those data leaks performed during the application life-cycle. For instance, this allows to discover connections where the IMEI of the device is sent to a third-party using the network. FlowDroid was run limiting the RAM memory to 10GB and the execution time to 5 min.

4.3. Features extracted through dynamic analysis

This category of features groups the dynamic analysis results obtained with a customised version of the well known framework *Droidbox* [62]. This framework provides interesting features that can be analysed, such as network activity, accessed files, sent SMS and cryptographic functions that were captured during the execution. This framework[7] was adapted to obtain more detailed dynamic reports. This modification consist on deploying the Strace tool inside the Android emulator device, which allow to obtain a fine-grained list of all system calls performed in run time at the Linux level. When the analysis ends, this file is extracted and saved. Furthermore, there was also two more modifications applied to DroidBox. On the one side, the behaviour of the *MonkeyRunner* tool already implemented in DroidBox, was changed with the aim of stimulating the sample under analysis with a higher number of simulated user actions on the screen and buttons. On the other hand, this modified version allows to run the sample in a non GUI environment, thus enabling to run multiple simultaneous emulator instances in computing nodes.

In contrast to static analysis, the use of a dynamic analysis tool allows to model the real behaviour exhibited in a simulated environment where the application is executed. Giving capabilities to the sample, such as internet access, allows to capture those actions that are only visible when particular conditions are met. For instance, monitoring the suspicious application while it is being executed allows to reveal system calls which are invoked due to the use of reflection or dynamic code loading.

The OmniDroid dataset contains the whole execution log with all the events captured by the DroidBox tool and a log generated with Strace for each application analysed. These two information sources offer large amounts of data which in case of being used in combination with machine learning classifier need to be filtered and processed. The final user of this dataset is free to apply the most appropriate techniques and methods for this purpose. Furthermore, since these two logs report a sorted list of events where each one has its timestamp attached, it is also possible apply online learning techniques [65].

Each execution was run for 300 s in an emulator running Android 4.1.1 (the version employed by DroidBox) with an armeabi-v7a architecture.

4.4. VirusTotal report

In addition to the aforementioned features, all APKs have been analysed with the online malware detection tool VirusTotal. In fact, what this platform does is to analyse the APK using diverse malware detection engines such as: *AVG*, *Avast* and *F-Secure*, to name a few. Hence, data regarding this category are the results obtained from each malware detection engine, namely a negative/positive detection as well as the categorisation given by each antivirus engine. This report is included as a ground truth to categorise each simple sample between malicious or non malignant and to allocate a family or variant tag to each sample as well. The final user of OmniDroid is in charge of defining the ϵ threshold as the minimum number of positives which determine the nature of the sample.

4.5. Protection against adversarial attacks

Recent research has proven that adversarial attacks against machine learning aided detection tools can provoke misclassifications [66]. The wide set of features contained in the OmniDroid dataset, extracted using both a static and a dynamic analysis approach allow to build resilient detection and classification tools. While methods relying on a reduced set of features can be more easily deceived (i.e. a modification of a single feature can lead to a misclassification), those that employ a large set of characteristics to describe the application behaviour present a significant barrier against this kind of attacks.

5. Dataset analysis and benchmark

All features extracted in this dataset define a large space from which different comparisons and considerations can be made. For instance, it is possible to analyse the presence of particular features when observing malware or benign samples. For these experiments, the threshold ϵ has been set to 1, so benign samples are those that are considered as such by all antivirus engines, while those that

[7] `https://github.com/alexMyG/DroidBox_AndroPyTool`

Malware		Benignware	
Permission	**% samples**	**Permission**	**% samples**
INTERNET	96.21%	INTERNET	94.82%
ACCESS_NETWORK_STATE	85.45%	ACCESS_NETWORK_STATE	72.95%
WRITE_EXTERNAL_STORAGE	81.31%	WRITE_EXTERNAL_STORAGE	61.49%
READ_PHONE_STATE	80.21%	WAKE_LOCK	41.60%
ACCESS_WIFI_STATE	60.40%	ACCESS_WIFI_STATE	39.14%
WAKE_LOCK	49.05%	READ_PHONE_STATE	37.13%
ACCESS_COARSE_LOCATION	41.99%	VIBRATE	33.33%
GET_TASKS	39.12%	ACCESS_FINE_LOCATION	27.70%
ACCESS_FINE_LOCATION	37.00%	ACCESS_COARSE_LOCATION	27.45%
VIBRATE	36.87%	GET_ACCOUNTS	26.82%

Table 5: Ten most frequent permissions declared in the Android Manifest for each application in the malware and benignware sets.

are labelled as malware by at least antivirus are allocated to the malware set.

5.1. Dataset analysis

An evaluation of how different samples employ the different features is a useful mechanism to draw general conclusions regarding the differences between both types of samples. In the next subsection, different features are assessed individually.

5.1.1. Permissions required analysis

Table 5 shows the top 10 permissions used among the malware and the benignware set (in terms of percentage of samples where the permission is reflected in the Android Manifest). Most of the samples of both categories required Internet access in order to be executed, so no conclusion can be drawn from this fact. However, there is one which makes a big difference, the `READ_PHONE_STATE` permission. It allows to read relevant information such as the phone number, ongoing calls or cellular network information. The official Android documentation already warns about the danger of this permission, as it can be used to access very sensitive information. While about 80% of the malware malware samples require this permission, this figure decreases to 37% in the benignware set.

In general, malware is more likely to demand permissions, thus accessing a large set of functionalities. For instance, SMS related permissions are much more present among malicious samples. In particular, `SEND_SMS` or `RECEIVE_SMS` permissions are required by 29% and 24% of the malware samples respectively, while this numbers decrease to 5% within the benign set. This leads to conclude that the use of SMS services is an important indicator to consider whether a suspicious sample is malicious or not. Among the rest of permissions, the use of `RECEIVE_BOOT_COMPLETE` is also relevant in the case of malware. Most of those samples categorised as malware employ this permission to activate the malicious payload once the device has been restarted.

5.1.2. Opcodes analysis

A study on the use of opcodes has also been performed. Table 6 shows the top 10 opcodes found among the smali code in the malware and benignware sets. In general, an analysis based on counting the use of particular *opcodes* will not lead to any conclusive assessment. Proof of this, is that the top 50 opcodes are used by at least 90% of the samples, which means that no differences can be found when comparing both batches of samples. This is an expected circumstance, given that opcodes show instructions at a very high level, so they cannot be used to distinguish relevant behaviours. For instance, the use of *invoke* opcodes allows to know when a system call is invoked from the application, but without the specific call specification, no possible intentions can be inferred.

Malware		Benignware	
Opcode	**% samples**	**Opcode**	**% samples**
return-void	100.0%	return-void	100.0%
invoke-direct	100.0%	invoke-direct	100.0%
invoke-static	99.95%	invoke-super	99.99%
invoke-virtual	99.95%	invoke-virtual	99.99%
new-instance	99.95%	move-result-object	99.97%
move-result-object	99.95%	invoke-static	99.96%
const/4	99.94%	new-instance	99.96%
move-result	99.92%	goto	99.88%
goto	99.88%	move-result	99.87%
if-eqz	99.78%	const/4	99.87%

Table 6: Ten most frequent opcodes found in the smali code obtained for each application in the malware and benignware sets.

5.1.3. System commands analysis

Malware		Benignware	
System command	**% samples**	**System command**	**% samples**
id	67.14%	id	68.56%
start	53.38%	top	66.83%
gzip	41.52%	start	59.55%
date	39.55%	service	59.04%
service	32.61%	gzip	52.35%
log	32.48%	input	47.41%
input	27.59%	mv	46.85%
stop	26.65%	log	43.41%
sh	25.91%	sh	42.89%
top	24.98%	stop	37.33%

Table 7: Ten most frequent system commands found within the code of each application in the malware and benignware sets.

Contrary to the trends seen in the usage of permissions by both types of samples, in the *OmniDroid* dataset the

Malware		Benignware	
API Package	**% samples**	**API Package**	**% samples**
android.app	99.96%	android.app	99.65%
java.lang	99.91%	java.lang	99.56%
java.io	98.8%	android.content	99.44%
android.content	97.85%	java.io	99.02%
android.os	94.58%	android.content.res	98.57%
android.content.res	94.0%	android.widget	96.65%
android.util	91.45%	android.os	96.56%
android.net	91.42%	android.view	96.47%
java.util	90.25%	java.lang.ref	96.47%
android.widget	90.05%	android.util	96.28%

Table 8: Ten most frequent API packages referenced on each application in both the malware and benignware sets.

use of system commands were more significant among benign samples (see Table 7). However, there can be found important details among them. For instance, 19% of the malware samples make use of the `chmod` command, which is necessary to give write and execution permissions to hidden scripts included among the app files. Among the benign samples, this number is reduced to 5%. In contrast, a wide set of system commands that could be tentatively associated with malicious behaviours, are in fact more used among benign samples. For instance, the commands *gzip* and *mv* could be used to decompress a file containing a malicious payload and to place it in the system apps folder, however, both are more commonly employed among benign samples. The rest of system commands do not allow to extract any relevant conclusion that could make a difference between both kinds of samples.

5.1.4. API packages analysis

API calls form a useful mechanism to identify relevant differences between samples with good or non-legal intentions. In this study, these calls are grouped by the API packages in which they are defined, in order to depict general characteristic patterns. In table 8 it is possible to observe this ranking of number of calls per API package. In general, both types of samples exhibit a similar use of the most common API packages. For instance, the *android.app* and *java.lang* packages, which are present in all samples, include the most basic functionality of the Android environment and Java language features respectively. An important difference can be found in the *android.telephony* groups of API calls. While 83% of the malicious samples invoke calls contained in this package, this number is reduced to 63% in the benignware set. As it was already observed when analysing the use of different permissions, services related to telephony and SMS remain very present among malware samples.

5.1.5. Intents analysis

Intents are a key communication element in the Android environment. Basically, they allow to ask permission to the system to run another application component. Thus, Intents describe the actions that an application can take. For instance, an Intent could demand actions such as making a phone call or taking a picture. Table 9 shows the top 10 actions declared in the Android Manifest by Intent-filters grouped by the nature of the samples. There are noticeable patterns which can be inferred from this ranking. Permission *android.intent.action.*`BOOT_COMPLETED` is declared in the Android Manifest in order to allow the application to receive a broadcast intent reporting when the device has been restarted. This is typically employed by malware aiming to hide the malicious payload until the next device boot-up and to force that a malicious application is actually executed even if the user does not start it manually. While about 38% of the malware samples are listening to this broadcast, only 19% of the benign apps do it. Another examples of broadly used permissions among malicious samples is *android.intent.action.*`USER_PRESENT` or those related to the SMS services (as already noted when analysing the use of Android permissions) such as *android.provider.Telephony.*`SMS_RECEIVED`.

5.2. FlowDroid analysis

Information flows represent vital information to understand the behaviour of a sample and are able to reveal patterns that could be associated to a sequence of malicious actions. In order to assess the differences between the malware and the benignware sets, Fig. 3 shows the number of flows found across all samples between categories of sources (at the left) and sinks (at the right). These 31 categories are defined by the SuSi framework [64].

As it can be seen, there is an important number of links between the `NOT_EXISTING` and `NO_CATEGORY` (those related to non-private data) categories. These categories have been omitted in the rest of the study. When analysing information flows in the malware diagram starting from the `DATABASE_INFORMATION` category, an important number reach the `LOCATION_INFORMATION` category, mainly focused on getting information regarding the device location.

There are also important patterns which can be deduced from the malware set. For example, there is a big number of information flows from `SMS_MMS` to `IPC` (interprocess communication). In contrast, `SMS_MMS` category shows little activity in the benign set. At the same time, there is more activity in this set starting from `IPC` and `FILE` categories. As sink, `UNIQUE_IDENTIFIER` category receives an important number of flows, which is a remarkable fact, since it includes relevant calls such as `getDeviceId()` or `getImei()`. These calls are often used by malware for many purposes. For instance, they can be used to generate unique keys which are later employed to encrypt user's files. Another important category with a similar behaviour in both datasets is `LOG`, which includes a broad range of actions such as those related to wallpaper or policies managing.

5.3. Dynamic features analysis

The dynamic traces obtained with *DroidBox* have also been studied. In general terms, both collections show

Malware		Benignware	
Intent	**% samples**	**Intent**	**% samples**
android.intent.action.MAIN	99.86%	android.intent.action.MAIN	99.81%
android.intent.action.BOOT_COMPLETED	38.05%	android.intent.action.VIEW	21.64%
android.net.conn.CONNECTIVITY_CHANGE	26.01%	com.google.android.c2dm.intent.RECEIVE	21.45%
android.intent.action.USER_PRESENT	25.80%	android.intent.action.BOOT_COMPLETED	19.46%
android.intent.action.PACKAGE_ADDED	24.00%	com.google.android.c2dm.intent.REGISTRATION	14.74%
android.intent.action.VIEW	16.96%	android.net.conn.CONNECTIVITY_CHANGE	11.65%
android.provider.Telephony.SMS_RECEIVED	15.42%	com.android.vending.INSTALL_REFERRER	10.67%
android.intent.action.PACKAGE_REMOVED	15.15%	android.intent.action.USER_PRESENT	6.73%
com.google.android.c2dm.intent.RECEIVE	11.47%	android.intent.action.PACKAGE_REMOVED	5.95%
com.google.android.c2dm.intent.REGISTRATION	8.78%	android.appwidget.action.APPWIDGET_UPDATE	5.68%

Table 9: Ten most frequent Android intents in the apps from both malware and benignware sets.

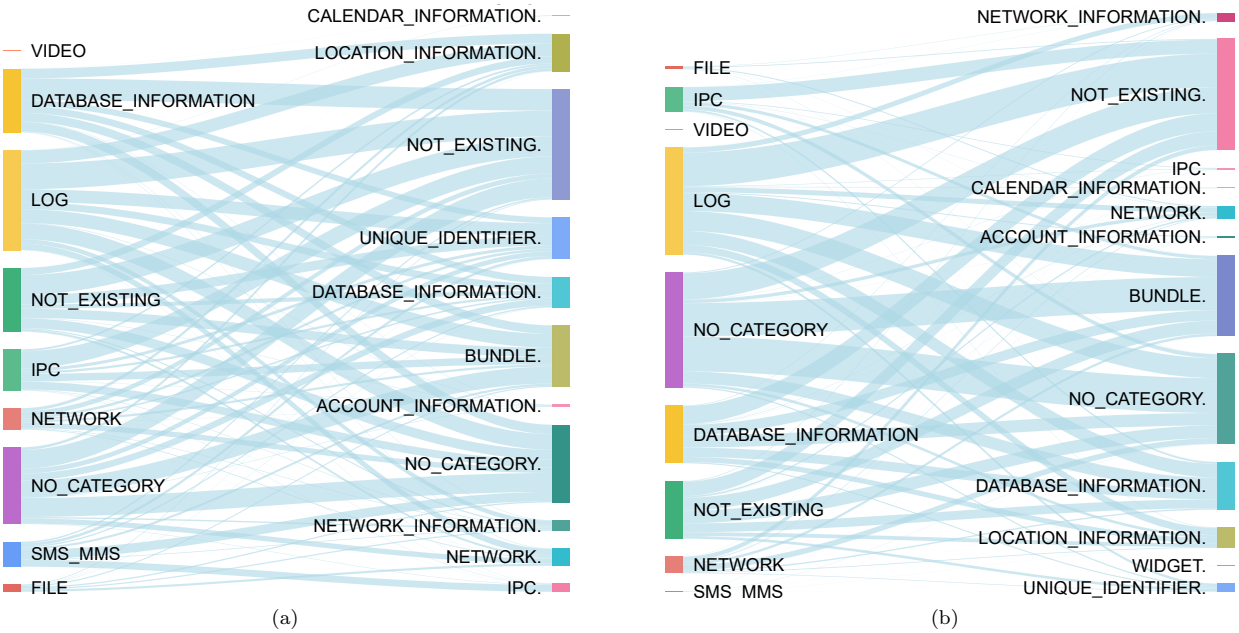


Figure 3: Number of information flows between categories of sinks and sources discovered by FlowDroid among all samples in the (a) malware set and in the (b) benignware set.

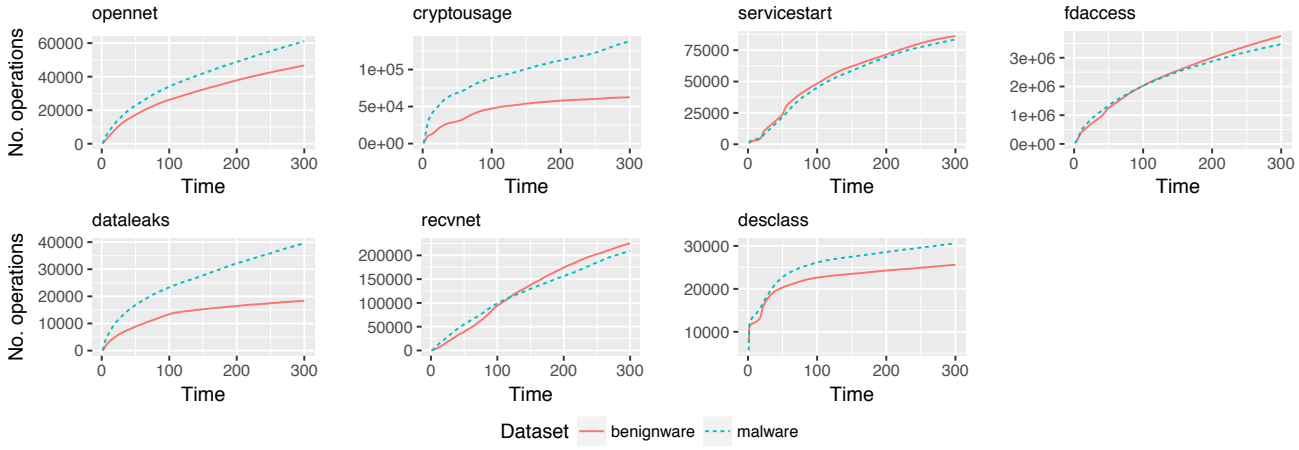


Figure 4: Cumulative sum of the number of operations, when *Droidbox* tool is used, for all samples over both (benignware and malware) datasets.

Dynamic feature	Malware	Benignware
accessedfiles	315.74	341.44
cryptousage	12.56	5.70
dataleaks	3.59	1.67
dexclass	2.78	2.33
fdaccess	315.72	341.42
opennet	5.56	4.25
phonecalls	0.01	0.02
recvnet	19.05	20.48
sendnet	4.85	2.65
sendsms	0.21	0.00
servicestart	7.60	7.84

Table 10: Average number of dynamic operations per execution in the malware and benignware set.

a very similar behaviour when they are executed (see Table 10). The most relevant differentiating factor lies in the number of cryptographic operations performed by the malware set. On average, malicious samples invoke 12.56 operations, while in the benignware set this figure falls by half. A greater use of SMS services can also be reflected in these dynamic traces, which are more present in the malware set.

Fig. 4 shows the cumulative sum of the number of operations through the 300 seconds execution in the emulator for each different category of operations. In general, malware deploys a slightly more active behaviour, a fact which is more remarkable in the case of the `cryptousage` and `dataleaks` categories. In the rest of categories, both sets perform very close. For instance, `servicestart` and `fdaccess` show a completely parallel behaviour. Finally, it should be noted that both kinds of samples mainly load code from *dex* classes in the first 30 seconds.

6. Testing the data using ensemble-based classification algorithms

Many malware detection and classification tools are based on machine learning algorithms, performing a learning process from a training set of samples represented by a set of features. The purpose of the process described in this section is to demonstrate the feasibility and ease of use of the dataset when using it to build classifiers through machine learning methods (in particular, ensemble methods), rather than building a malware classifier achieving a high accuracy. In other words, show that the dataset is usable out-of-the-box, and although some promising experimental results are currently obtained, there is still a large room for improvement. These methods were used separately over the set of static and dynamic features extracted, and finally a fusion based approach where both types of features are combined is proposed. The same threshold related to the minimum number of positives applied in the previous section, defined by $\epsilon = 1$, is again used to train and test these algorithms.

6.1. Classification results based on static features

Statically extracted features can be used to build representative sample vectors where each position represents the number of occurrences that a certain characteristic is present in the sample (i.e. the number of times that a specific API call is invoked). Given a set of samples X of size n:

$$X = \{x_1, x_2, \ldots, x_n\} \tag{1}$$

Each sample x_i is represented by a vector of m static characteristics:

$$x_i = \{sc_i^1, sc_i^2, \ldots, sc_i^m\} \tag{2}$$

At the same time, each sample x_i is categorised as benign or malicious according to its label l_i, $l_i \in \{0, 1\}$ Our objective is to train the classifier than establishes this relation:

$$Cls(x_i) = (p(l_i), l_i) \tag{3}$$

Six well-known state-of-the-art ensemble methods for classification were trained with different combinations of features. These algorithms, executed with the *Scikit-learn* library for Python [67], are: AdaBoost, Bagging (with Random Forest estimators), ExtraTrees, Gradient Boosting, Random Forest (all of them using a parameter of 100 internal estimators) and a Voting classifier combining a Random Forest, KNN and a simple decision tree classifier as estimators with the same weights. The *CSV* file used in these experiments (containing all the labelled features vectors) is also publicly available.

For all the experiments, based on different features combinations, the average accuracy is shown. The results for each experiment is calculated based on a cross validation process of 10 folds. Table 11 depicts the classification results for individuals features as the only input for the classification and for different combinations of features as well.

By analysing the results achieved by each static feature, API calls allow to obtain the maximum accuracy with a random forest classifier, reaching 89.3%. If grouping these API calls by the API package in which they are defined, these features become useless to distinguish between malware and benignware. In the second place, appears the combinations that include the use of API calls, achieving around 89.2% accuracy, which can be considered as a high and also similar value. Other features such as *FlowDroid*, a feature which apparently could reveal important traces related to malicious behaviours (such as sending the device IMEI to an attacker) or system commands are not able to make a division of the feature space with the same level of precision. In general, random forest clearly achieves the best results in most of the experiments in terms of accuracy and precision. Nevertheless, a Bagging classifier obtains similar results for certain combinations of features.

Features set	Metric	AdaBoost	Bagging	ExtraTrees	Gradient Boosting	Random Forest	Voting
Activities	**Acc**	0.506 ± 0.002	0.506 ± 0.002	0.506 ± 0.002	0.506 ± 0.002	0.506 ± 0.002	0.506 ± 0.002
	Prec	0.602 ± 0.036	0.602 ± 0.036	0.602 ± 0.036	0.602 ± 0.036	0.602 ± 0.036	0.602 ± 0.036
API calls	**Acc**	0.859 ± 0.008	**0.891 ± 0.007**	0.89 ± 0.006	0.871 ± 0.009	**0.893 ± 0.006**	0.886 ± 0.006
	Prec	0.859 ± 0.008	**0.892 ± 0.007**	0.89 ± 0.006	0.871 ± 0.009	**0.893 ± 0.006**	0.887 ± 0.006
API Packages	**Acc**	0.5 ± 0	0.5 ± 0	0.5 ± 0	0.5 ± 0	0.5 ± 0	0.5 ± 0
	Prec	0.25 ± 0	0.25 ± 0	0.25 ± 0	0.25 ± 0	0.25 ± 0	0.25 ± 0
FlowDroid	**Acc**	0.677 ± 0.009	0.706 ± 0.008	0.708 ± 0.008	0.681 ± 0.01	0.708 ± 0.008	0.704 ± 0.009
	Prec	0.72 ± 0.013	0.744 ± 0.009	0.748 ± 0.009	0.723 ± 0.013	0.746 ± 0.009	0.744 ± 0.01
FlowDroid, API calls	**Acc**	0.86 ± 0.01	**0.891 ± 0.007**	0.889 ± 0.006	0.872 ± 0.007	**0.892 ± 0.007**	0.886 ± 0.006
	Prec	0.86 ± 0.01	**0.892 ± 0.007**	0.89 ± 0.006	0.872 ± 0.007	**0.892 ± 0.007**	0.886 ± 0.006
FlowDroid, API Packages	**Acc**	0.677 ± 0.009	0.708 ± 0.008	0.709 ± 0.008	0.681 ± 0.01	0.707 ± 0.01	0.704 ± 0.008
	Prec	0.72 ± 0.013	0.745 ± 0.009	0.749 ± 0.009	0.722 ± 0.013	0.745 ± 0.011	0.745 ± 0.01
Opcodes	**Acc**	0.833 ± 0.011	0.873 ± 0.01	0.869 ± 0.007	0.846 ± 0.009	0.874 ± 0.012	0.868 ± 0.009
	Prec	0.833 ± 0.011	0.873 ± 0.009	0.869 ± 0.007	0.846 ± 0.009	0.874 ± 0.011	0.868 ± 0.009
Permissions	**Acc**	0.781 ± 0.01	0.824 ± 0.006	0.824 ± 0.006	0.792 ± 0.008	0.825 ± 0.007	0.821 ± 0.008
	Prec	0.781 ± 0.01	0.826 ± 0.006	0.826 ± 0.006	0.792 ± 0.008	0.827 ± 0.006	0.823 ± 0.008
Receivers	**Acc**	0.824 ± 0.005	0.876 ± 0.01	0.877 ± 0.009	0.84 ± 0.005	0.877 ± 0.01	0.875 ± 0.009
	Prec	0.825 ± 0.006	0.876 ± 0.01	0.877 ± 0.009	0.84 ± 0.005	0.877 ± 0.01	0.875 ± 0.009
Receivers, API calls	**Acc**	0.858 ± 0.01	0.889 ± 0.006	0.89 ± 0.008	0.875 ± 0.007	**0.892 ± 0.008**	0.885 ± 0.007
	Prec	0.858 ± 0.01	0.889 ± 0.006	0.891 ± 0.008	0.875 ± 0.007	**0.892 ± 0.008**	0.885 ± 0.007
Receivers, API calls, Opcodes, Permissions	**Acc**	0.862 ± 0.009	0.89 ± 0.008	0.891 ± 0.008	0.88 ± 0.008	**0.891 ± 0.007**	0.884 ± 0.008
	Prec	0.862 ± 0.009	0.89 ± 0.008	0.891 ± 0.008	0.88 ± 0.008	**0.892 ± 0.007**	0.884 ± 0.008
Receivers, API calls, Opcodes, Permissions, FlowDroid	**Acc**	0.865 ± 0.008	0.889 ± 0.007	0.892 ± 0.009	0.879 ± 0.008	**0.891 ± 0.008**	0.883 ± 0.007
	Prec	0.865 ± 0.008	0.89 ± 0.007	0.893 ± 0.009	0.88 ± 0.008	**0.892 ± 0.008**	0.883 ± 0.007
Receivers, Services, Activities	**Acc**	0.825 ± 0.005	0.875 ± 0.008	0.877 ± 0.007	0.843 ± 0.008	0.876 ± 0.008	0.876 ± 0.008
	Prec	0.826 ± 0.005	0.875 ± 0.008	0.878 ± 0.007	0.843 ± 0.008	0.876 ± 0.008	0.876 ± 0.008
Receivers, Services, Activities, API calls	**Acc**	0.858 ± 0.01	0.889 ± 0.007	0.888 ± 0.006	0.874 ± 0.008	0.889 ± 0.007	0.884 ± 0.007
	Prec	0.858 ± 0.01	0.889 ± 0.007	0.889 ± 0.006	0.874 ± 0.008	0.89 ± 0.007	0.884 ± 0.007
Services	**Acc**	0.515 ± 0.003	0.516 ± 0.002	0.516 ± 0.002	0.515 ± 0.003	0.516 ± 0.002	0.516 ± 0.003
	Prec	0.749 ± 0.013	0.741 ± 0.015	0.743 ± 0.015	0.75 ± 0.012	0.741 ± 0.015	0.742 ± 0.015
System commands	**Acc**	0.761 ± 0.009	0.827 ± 0.007	0.827 ± 0.007	0.776 ± 0.007	0.826 ± 0.006	0.82 ± 0.008
	Prec	0.763 ± 0.009	0.828 ± 0.007	0.828 ± 0.007	0.777 ± 0.007	0.827 ± 0.006	0.821 ± 0.008

Table 11: Performance of several ensemble learning algorithms from the state of the art according to the static feature set used as input. The best overall results are highlighted in bold type.

Features set	Metric	AdaBoost	Bagging	ExtraTrees	Gradient Boosting	Random Forest	Voting
Transitions	**Acc**	0.731 ± 0.01	0.776 ± 0.01	0.775 ± 0.012	0.741 ± 0.007	0.775 ± 0.009	0.763 ± 0.009
Transitions	**Prec**	0.731 ± 0.01	0.777 ± 0.01	0.776 ± 0.012	0.741 ± 0.007	0.775 ± 0.009	0.764 ± 0.009
Frequencies	**Acc**	0.739 ± 0.009	0.78 ± 0.012	0.774 ± 0.01	0.743 ± 0.009	0.778 ± 0.011	0.768 ± 0.008
Frequencies	**Prec**	0.74 ± 0.009	0.78 ± 0.012	0.774 ± 0.01	0.743 ± 0.008	0.778 ± 0.011	0.769 ± 0.008
Combination	**Acc**	0.742 ± 0.009	**0.786 ± 0.007**	0.779 ± 0.007	0.751 ± 0.006	**0.785 ± 0.006**	0.771 ± 0.011
Combination	**Prec**	0.743 ± 0.009	**0.786 ± 0.008**	0.78 ± 0.007	0.751 ± 0.006	**0.785 ± 0.006**	0.772 ± 0.011

Table 12: Performance of several ensemble learning algorithms from the state of the art according to the dynamic feature set used as input. The best overall performance is highlighted in bold type.

6.2. Classification results based on dynamic features

Once the static features have been analysed, it is tested the use of the dynamic information, extracted after the execution of each sample in an emulator monitored by the DroidBox tool. Each analysis delivers a temporal sequence of actions performed throughout the execution, where each action is linked to a category (i.e. file access), a timestamp and a series of parameters (i.e. the path of the file accessed). In order to build a feature vector which can be used to represent the sample behaviour, a Markov chains based representation [68] was employed, as previously were used by Martín et al. [69] to model DroiBox dynamic traces. This model allows to represent the transitions probabilities a_{ij} between a series of n states:

$$S = \{S_1, S_2, \ldots, S_n\} \tag{4}$$

The full dynamic trace of a certain sample x_i is firstly transformed into a $n * n$ matrix of n unique states, which represent the transition probability between each pair of states. Each state is represented by the category of the action and a series of arguments. An example of state is: `fdaccess\operation=read|path=/proc/tty`, which indicates a file operation of type *read* over a file located at `/proc/tty`. All paths were truncated to limit their depth to two levels. In addition, only those transitions with $a_i > 0$ were considered in order to reduce the matrix size. A final set of 1,127 states were obtained.

In order to place this information into a feature vector, the matrix of each sample was flattened, thus generating a representative vector of k transitions probabilities:

$$x_i = \{tp_i^1, tp_i^2, \ldots, tp_i^k\} \tag{5}$$

At the same time, the frequency of each state were also considered, as it can provide supplementary data able to improve the representation of the sample behaviour. In this case, there is a new representative vector including the frequency of o states for each sample:

$$x_i = \{sf_i^1, sf_i^2, \ldots, sf_i^o\} \tag{6}$$

Both representations can be combined in order to build a new vector containing transition probabilities and frequencies of states:

$$x_i = \{tp_i^1, tp_i^2, \ldots, tp_i^k, sf_i^1, sf_i^2, \ldots, sf_i^m\} \tag{7}$$

These representations were implemented an tested with the same pool of algorithms tested previously with the static features vectors. The results obtained are shown in Table 12, and include the classification accuracy and precision using both representations independently, and a third one combining transitions and states frequencies. Again, a 10-fold cross validation is employed.

Opposite of what one would expect, the use of frequencies states allows to reach higher accuracy values than using the transition probabilities among states, meaning that the distribution of states provides a better description of the samples behaviour. However, a combination of both features allows to achieve the best results, by using a Bagging model composed of Random Forest classifiers or a single Random Forest. Both allow to obtain 78.6% in terms of accuracy and precision.

In comparison to the results achieved through the use of static features, dynamically extracted characteristics experience an important fall of more than 10%. There are several reasons which can be attributed to this fact. On the one side, reports delivered by the dynamic analysis tool used, DroidBox, could be not enough detailed in order to make a division of the space able to differentiate between malicious and benign applications. Besides, DroidBox has not been updated in the last years and could be not able to monitor the most recent malware applications behaviours. However, there are other plausible explanations. For instance, malware samples have proven to be able to detect when they run in a sandbox, thus not deploying the malicious payload [70]. For the classifier perspective, there is space for future users of the OmniDroid dataset for applying techniques aimed at building stronger estimators. For instance, the use of diversity-inducing methods can help in this task [71].

6.3. Static-dynamic fusion method for detecting Android malware

The two previous sections describe how the state-of-the-art ensemble methods works when our set of static and dynamic features are given as a feature vectors for classification. As it has been shown, the use of a classifier with an input based on static features allows to improve the accuracy when it is compared against a dynamic based approach. Despite of this, the growing complexity of malware makes necessary to apply any available techniques, to discern the nature of a given suspicious sample. To this end, a new approach was developed based on fusion of the behaviour information extracted from a hybrid analysis where the static and dynamic features are combined.

This fusion approach is based on joining the classification models that work best in each type of feature (static and dynamic) to build a voting classifier where each model contributes to the final categorisation. It features three different representations to generate the classifier. While it adopts API calls as static features (they obtain the best results in the static comparison), it also uses the three representations analysed in the dynamic analysis approach. Thus, there is a mixture of API calls features which are combined with transition probabilities (equation 8), frequencies states (equation 9) and combination of both (equation 10) to build three vectors:

$$xtp_i = \{sc_i^1, sc_i^2, \ldots, sc_i^m, tp_i^1, tp_i^2, \ldots, tp_i^k\} \tag{8}$$

$$xsf_i = \{sc_i^1, sc_i^2, \ldots, sc_i^m, sf_i^1, sf_i^2, \ldots, sf_i^o\} \tag{9}$$

$$\begin{aligned} xdc_i = \{&sc_i^1, sc_i^2, \ldots, sc_i^m, tp_i^1, tp_i^2, \\ &\ldots, tp_i^k, sf_i^1, sf_i^2, \ldots, sf_i^o\} \end{aligned} \tag{10}$$

The design of the voting classifier implemented is represented in Fig. 5. It includes a Random Forest classifier which is in charge of classifying the static features section of the input vector, while a Bagging classifier (it slightly exceed the results of Random Forest) receives dynamic features. Each classifier contributes to the classification of a given feature according to two weight parameters W_{RF} and W_{BG}. The final categorisation is calculated as follows:

$$Cls(x_i) = \left[\tfrac{W_{RF}S_i + W_{BG}D_i}{2}\right] = l_i, l_i \in \{0, 1\} \tag{11}$$

A grid search was run to decide the best value for W_{RF} and W_{BG} within the range $[0.1, 0.9]$. The results are shown in Table 13. As it could be expected, and according to the results previously seen, when analysing the use dynamic features as the only input, in these experiments the best results are also achieved by combining transition probabilities and frequencies of states. The best result is achieved using $W_{RF} = 0.7$ and $W_{BG} = 0.3$, after a 10-fold cross validation process. This fusion approach achieves 89.7% accuracy, which slightly improves the results when both types of features are used independently.

7. Conclusions

Android conforms a platform that has been selected as the target by many black hats to perform malicious

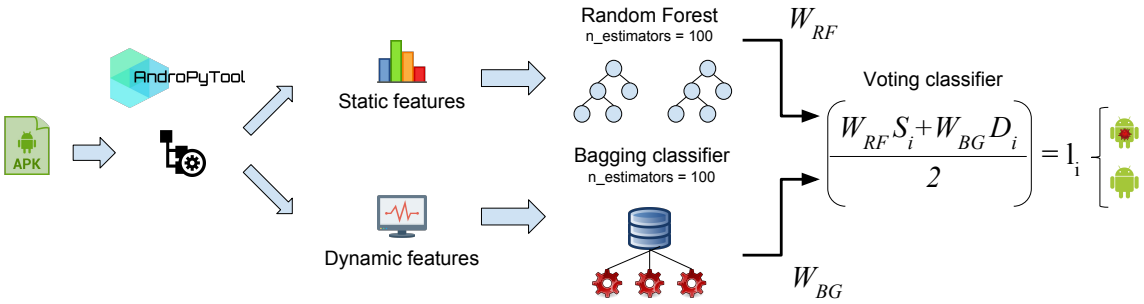


Figure 5: Static-dynamic fusion approach based on a voting classifier which combines static and dynamic classification models.

W_{RF}	W_{BG}	Metric	Transitions	Frequencies	Combination
0.1	**0.9**	**Acc**	0.812 ± 0.01	0.81 ± 0.01	0.815 ± 0.011
		Prec	0.813 ± 0.01	0.81 ± 0.01	0.816 ± 0.011
0.2	**0.8**	**Acc**	0.834 ± 0.01	0.83 ± 0.007	0.837 ± 0.009
		Prec	0.835 ± 0.01	0.83 ± 0.007	0.838 ± 0.009
0.3	**0.7**	**Acc**	0.86 ± 0.008	0.854 ± 0.008	0.861 ± 0.007
		Prec	0.86 ± 0.008	0.855 ± 0.008	0.862 ± 0.007
0.4	**0.6**	**Acc**	0.879 ± 0.006	0.873 ± 0.007	0.88 ± 0.007
		Prec	0.879 ± 0.006	0.873 ± 0.007	0.88 ± 0.007
0.5	**0.5**	**Acc**	0.891 ± 0.007	0.887 ± 0.008	0.892 ± 0.007
		Prec	0.891 ± 0.006	0.887 ± 0.007	0.892 ± 0.007
0.6	**0.4**	**Acc**	0.896 ± 0.01	0.895 ± 0.008	0.894 ± 0.01
		Prec	0.896 ± 0.01	0.896 ± 0.008	0.894 ± 0.01
0.7	**0.3**	**Acc**	0.895 ± 0.008	0.896 ± 0.008	**0.897 ± 0.008**
		Prec	0.895 ± 0.008	0.896 ± 0.008	**0.897 ± 0.007**
0.8	**0.2**	**Acc**	0.895 ± 0.008	0.896 ± 0.008	0.895 ± 0.007
		Prec	0.895 ± 0.008	0.896 ± 0.008	0.896 ± 0.007
0.9	**0.1**	**Acc**	0.893 ± 0.008	0.893 ± 0.008	0.894 ± 0.006
		Prec	0.893 ± 0.008	0.894 ± 0.008	0.894 ± 0.006

Table 13: Results achieved with the static-dynamic fusion approach. W1 and W2 represent the weight given to the Random Forest classifier and to the Bagging classifier in the voting-based approach, responsible of receiving as input the static and dynamic features respectively.

attacks, or to develop applications with non-legal purposes. Many of the efforts made towards counteracting these attacks are based on the training process of an antimalware tool. This process, specially when is based on a machine learning algorithm, requires from a large dataset of samples to be adequately trained. Previously to use this kind of algorithms, it is necessary to extract and analyse an adequate set of features that could be used during the learning process. *OmniDroid* primarily aims to set a benchmark dataset useful for those who are developing antimalware tools. Instead of providing a set of executable files, this dataset provides already analysed samples using different state-of-the-art malware analysis tools in order to facilitate the process. The characteristics of the *OmniDroid* dataset make it also suitable to be used as a general purpose benchmark dataset. For instance, it could be employed to perform algorithms comparison, to test feature selection techniques, or to assess new clustering or classification algorithms among many others possibilities.

Throughout this paper, we have evaluated this dataset, performed different studies and provided results after analysing groups of features individually, in order to deliver interested researchers with a preliminary evaluation of the data, as well as demonstrating the high potential the dataset has and its ease of use. Although the experimental results show a good performance for most of the state-of-the art algorithms analysed, there is still a clear room for improvement over OmniDroid dataset. This could help to a researcher to train, test and evaluate the performance of their algorithms, or simply to compare their malware detection techniques using OmniDroid as a new, clean and correctly pre-processed, data benchmark.

Finally, we are considering to expand the architecture of *AndroPyTool* [10], in order to allow extracting more features for each sample by integrating other feature extraction and reverse engineering tools. At the same time, we will work in the near future with the goal of increasing the number of samples contained in the *OmniDroid* dataset, updating it to introduce recent samples found in the wild. The OmniDroid dataset, and their future updates, will be available for the research community at the *AIDA Datasets Repository*[8].

Acknowledgement

This work has been co-funded by the following grants: Comunidad Autónoma de Madrid under grant S2013/ICE-3095 (CIBERDINE: Cybersecurity, Data and Risks); Spanish Ministry of Science and Education and Competitivity (MINECO) and European Regional Development Fund (FEDER) under grants TIN2014-56494-C4-4-P (EphemeCH), and TIN2017-85727-C4-3-P (DeepBio). We also would like to thank to the Koodous[9] Team for the large dataset of samples that they provided us.

References

[1] Kolias, C., Kambourakis, G., Stavrou, A., Voas, J.M.. DDoS in the IoT: Mirai and other botnets. IEEE Computer 2017;50(7):80–84.

[2] Ehrenfeld, J.M.. Wannacry, cybersecurity and health information technology: A time to act. Journal of Medical Systems 2017;41(7):104:1.

[8] https://aida.ii.uam.es/datasets/

[9] `https://koodous.com/`

[3] IDC, . Worldwide Quarterly Mobile Phone Tracker. Tech. Rep.; International Data Corporation (IDC); Massachusetts, USA; 2017.

[4] Chandrasekar, K., Cleary, G., Cox, C., Lau, H., Nahorney, B., O'Gorman, B., et al. Internet security threat report. Tech. Rep.; Symantec Corporation; California, USA; 2017.

[5] Quinlan, J.R.. C4.5: programs for machine learning. Elsevier; 2014.

[6] Boser, B.E., Guyon, I., Vapnik, V.. A training algorithm for optimal margin classifiers. In: Haussler, D., editor. Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992. ACM; 1992, p. 144–152.

[7] Langley, P., Iba, W., Thompson, K.. An analysis of bayesian classifiers. In: Swartout, W.R., editor. 10th National Conference on Artificial Intelligence. AAAI Press / The MIT Press; 1992, p. 223–228.

[8] Dietterich, T.G.. Ensemble methods in machine learning. In: Multiple Classifier Systems, First International Workshop, MCS 2000; vol. 1857 of *Lecture Notes in Computer Science*. Springer; 2000, p. 1–15.

[9] Martín, A., Lara-Cabrera, R., Camacho, D.. A new tool for static and dynamic android malware analysis. In: Data Science and Knowledge Engineering for Sensing Decision Support. World Scientific; 2018, p. 509–516.

[10] AndroPyTool source code repository. 2017. URL: `https://github.com/alexMyG/AndroPyTool`; Last accessed: 2018-05-04.

[11] Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0). 2013. URL: `https://creativecommons.org/licenses/by-nc-sa/4.0/`; Last accessed: 2018-03-14.

[12] Martín, A., Calleja, A., Menéndez, H.D., Tapiador, J.E., Camacho, D.. ADROIT: android malware detection using meta-information. In: 2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016. IEEE; 2016, p. 1–8.

[13] Jang, J., Kang, H., Woo, J., Mohaisen, A., Kim, H.K.. Andro-dumpsys: Anti-malware system based on the similarity of malware creator and malware centric information. Computers & Security 2016;58:125–138.

[14] Feizollah, A., Anuar, N.B., Salleh, R., Suarez-Tangil, G., Furnell, S.. Androdialysis: Analysis of android intent effectiveness in malware detection. computers & security 2017;65:121–134.

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

[16] Feng, Y., Anand, S., Dillig, I., Aiken, A.. Apposcopy: semantics-based detection of android malware through static analysis. In: Cheung, S., Orso, A., Storey, M.D., editors. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22). ACM; 2014, p. 576–587.

[17] Suarez-Tangil, G., Tapiador, J.E., Peris-Lopez, P., Blasco, J.. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. Expert Systems with Applications 2014;41(4):1104–1117.

[18] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.. DREBIN: effective and explainable detection of android malware in your pocket. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014. The Internet Society; 2014, p. 1–12.

[19] Zheng, M., Sun, M., Lui, J.C.S.. Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. In: 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2013. IEEE Computer Society; 2013, p. 163–171.

[20] Aafer, Y., Du, W., Yin, H.. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In: 9th International ICST Conference, SecureComm 2013; vol. 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Springer; 2013, p. 86–103.

[21] Zhu, H.J., You, Z.H., Zhu, Z.X., Shi, W.L., Chen, X., Cheng, L.. Droiddet: Effective and robust detection of android malware using static analysis along with rotation forest model. Neurocomputing 2018;272:638–646.

[22] Yerima, S.Y., Sezer, S.. Droidfusion: A novel multilevel classifier fusion approach for android malware detection. IEEE Transactions on Cybernetics 2018;doi:`10.1109/TCYB.2017.2777960`.

[23] Deshotels, L., Notani, V., Lakhotia, A.. Droidlegacy: Automated familial classification of android malware. In: Jagannathan, S., Sewell, P., editors. 3rd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, PPREW 2014. ACM; 2014, p. 3:1–3:12.

[24] Wu, D., Mao, C., Wei, T., Lee, H., Wu, K.. DroidMat: Android malware detection through manifest and API calls tracing. In: Seventh Asia Joint Conference on Information Security, AsiaJCIS 2012. IEEE; 2012, p. 62–69.

[25] Yang, C., Xu, Z., Gu, G., Yegneswaran, V., Porras, P.A.. DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in android applications. In: 19th European Symposium on Research in Computer Security, ESORICS 2014; vol. 8712 of *Lecture Notes in Computer Science*. Springer; 2014, p. 163–182.

[26] Dash, S.K., Suarez-Tangil, G., Khan, S.J., Tam, K., Ahmadi, M., Kinder, J., et al. DroidScribe: classifying android malware based on runtime behavior. In: 2016 IEEE Security and Privacy Workshops, SP Workshops 2016. IEEE Computer Society; 2016, p. 252–261.

[27] Suarez-Tangil, G., Dash, S.K., Ahmadi, M., Kinder, J., Giacinto, G., Cavallaro, L.. DroidSieve: fast and accurate classification of obfuscated android malware. In: Ahn, G., Pretschner, A., Ghinita, G., editors. Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017. ACM; 2017, p. 309–320.

[28] Gascon, H., Yamaguchi, F., Arp, D., Rieck, K.. Structural detection of android malware using embedded call graphs. In: Sadeghi, A., Nelson, B., Dimitrakakis, C., Shi, E., editors. 2013 ACM Workshop on Artificial Intelligence and Security, AISec'13. ACM; 2013, p. 45–54.

[29] Xu, K., Li, Y., Deng, R.H.. ICCDetector: ICC-based malware detection on Android. IEEE Transactions on Information Forensics and Security 2016;11(6):1252–1264.

[30] Dini, G., Martinelli, F., Saracino, A., Sgandurra, D.. Madam: a multi-level anomaly detector for android malware. In: International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security. Springer; 2012, p. 240–253.

[31] Feldman, S., Stadther, D., Wang, B.. Manilyzer: Automated android malware detection through manifest analysis. In: 11th IEEE International Conference on Mobile Ad Hoc and Sensor Systems, MASS 2014. IEEE Computer Society; 2014, p. 767–772.

[32] Lindorfer, M., Neugschwandtner, M., Platzer, C.. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In: Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual; vol. 2. IEEE; 2015, p. 422–433.

[33] Chakradeo, S., Reaves, B., Traynor, P., Enck, W.. MAST: triage for market-scale mobile malware analysis. In: Buttyán, L., Sadeghi, A., Gruteser, M., editors. Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks, WISEC'13. ACM; 2013, p. 13–24.

[34] Peiravian, N., Zhu, X.. Machine learning for android malware detection using permission and API calls. In: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence. IEEE Computer Society; 2013, p. 300–305.

[35] Garcia, J., Hammad, M., Pedrood, B., Bagheri-Khaligh, A., Malek, S.. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. Tech. Rep.; George Mason University; Virginia, USA; 2015.

[36] Sheen, S., Anitha, R., Natarajan, V.. Android based malware detection using a multifeature collaborative decision fusion

approach. Neurocomputing 2015;151:905–912.
[37] Wang, W., Li, Y., Wang, X., Liu, J., Zhang, X.. Detecting android malicious apps and categorizing benign apps with ensemble of classifiers. Future Generation Computer Systems 2018;78:987–994.
[38] Yerima, S.Y., Sezer, S., McWilliams, G., Muttik, I.. A new android malware detection approach using bayesian classification. In: Barolli, L., Xhafa, F., Takizawa, M., Enokido, T., Hsu, H., editors. 27th IEEE International Conference on Advanced Information Networking and Applications, AINA 2013. IEEE Computer Society; 2013, p. 121–128.
[39] Yerima, S.Y., Sezer, S., Muttik, I.. Android malware detection using parallel machine learning classifiers. In: Eighth International Conference on Next Generation Mobile Apps, Services and Technologies, NGMAST 2014. IEEE; 2014, p. 37–42.
[40] Zhang, M., Duan, Y., Yin, H., Zhao, Z.. Semantics-aware android malware classification using weighted contextual API dependency graphs. In: Ahn, G., Yung, M., Li, N., editors. 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM. ISBN 978-1-4503-2957-6; 2014, p. 1105–1116.
[41] Canfora, G., Lorenzo, A.D., Medvet, E., Mercaldo, F., Visaggio, C.A.. Effectiveness of Opcode ngrams for detection of multi family android malware. In: 10th International Conference on Availability, Reliability and Security, ARES 2015. IEEE Computer Society; 2015, p. 333–340.
[42] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., et al. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: O'Boyle, M.F.P., Pingali, K., editors. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14. ACM; 2014, p. 259–269.
[43] Martín, A., Menéndez, H.D., Camacho, D.. MOCDroid: multi-objective evolutionary classifier for Android malware detection. Soft Computing 2017;21(24):7405–7415.
[44] Galar, M., Fernandez, A., Barrenechea, E., Bustince, H., Herrera, F.. A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 2012;42(4):463–484.
[45] Galar, M., Fernández, A., Barrenechea, E., Bustince, H., Herrera, F.. An overview of ensemble methods for binary classifiers in multi-class problems: Experimental study on one-vs-one and one-vs-all schemes. Pattern Recognition 2011;44(8):1761–1776.
[46] Zhou, Y., Jiang, X.. Dissecting android malware: Characterization and evolution. In: IEEE Symposium on Security and Privacy, SP 2012. IEEE Computer Society; 2012, p. 95–109.
[47] Contagio malware dumps. 2008. URL: `https://contagiodump.blogspot.com.es`; Last accessed: 2018-03-14.
[48] Allix, K., Bissyandé, T.F., Klein, J., Traon, Y.L.. Androzoo: collecting millions of android apps for the research community. In: Kim, M., Robbes, R., Bird, C., editors. 13th International Conference on Mining Software Repositories, MSR 2016. ACM; 2016, p. 468–471.
[49] Android-malware source code repository. 2016. URL: `https://github.com/ashishb/android-malware`; Last accessed: 2018-03-14.
[50] Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W.. Deep ground truth analysis of current android malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer; 2017, p. 252–276.
[51] Rashidi, B., Fung, C.J.. Xdroid: An android permission control using hidden markov chain and online learning. In: 2016 IEEE Conference on Communications and Network Security, CNS 2016. IEEE; 2016, p. 46–54.
[52] Rashidi, B., Fung, C.J., Bertino, E.. Android resource usage risk assessment using hidden markov model and online learning. Computers & Security 2017;65:90–107.
[53] AndroMalShare dataset. 2013. URL: `http://sanddroid.xjtu.edu.cn:8080/`; Last accessed: 2018-03-14.
[54] Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.. Avclass: A tool for massive malware labeling. In: 19th International Symposium on Research in Attacks, Intrusions, and Defenses, RAID 2016. 2016, p. 230–253.
[55] android-security-awesome. 2018. URL: `https://github.com/ashishb/android-security-awesome`; Last accessed: 2018-11-06.
[56] Androguard. 2011. URL: `https://github.com/androguard/androguard`; Last accessed: 2018-11-06.
[57] smali/backsmali. 2018. URL: `https://github.com/JesusFreke/smali`; Last accessed: 2018-11-06.
[58] Apktool. 2018. URL: `https://ibotpeaches.github.io/Apktool/`; Last accessed: 2018-11-06.
[59] dex2jar. 2018. URL: `https://github.com/pxb1988/dex2jar`; Last accessed: 2018-11-06.
[60] jd-gui. 2018. URL: `https://github.com/java-decompiler/jd-gui`; Last accessed: 2018-11-06.
[61] jadx. 2018. URL: `https://github.com/skylot/jadx`; Last accessed: 2018-11-06.
[62] Droidbox source code repository. 2011. URL: `https://github.com/pjlantz/droidbox`; Last accessed: 2018-03-14.
[63] Cuckoo-droid. 2018. URL: `https://github.com/idanr1986/cuckoo-droid`; Last accessed: 2018-11-06.
[64] Rasthofer, S., Arzt, S., Bodden, E.. A machine-learning approach for classifying and categorizing android sources and sinks. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014. The Internet Society; 2014, p. 1–15.
[65] Gomes, H.M., Barddal, J.P., Enembreck, F., Bifet, A.. A survey on ensemble learning for data stream classification. ACM Computing Surveys (CSUR) 2017;50(2):23.
[66] Calleja, A., Martín, A., Menéndez, H.D., Tapiador, J., Clark, D.. Picking on the family: Disrupting android malware triage by forcing misclassification. Expert Systems with Applications 2018;95:113–126.
[67] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., et al. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 2011;12:2825–2830.
[68] Fink, G.A.. Markov models for pattern recognition: from theory to applications. Springer Science & Business Media; 2014.
[69] Martín, A., Rodríguez-Fernández, V., Camacho, D.. Candyman: Classifying android malware families by modelling dynamic traces with markov chains. Engineering Applications of Artificial Intelligence 2018;74:121–133.
[70] Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.. Rage against the virtual machine: hindering dynamic analysis of android malware. In: Proceedings of the Seventh European Workshop on System Security. ACM; 2014, p. 5.
[71] Lobo, J.L., Laña, I., Ser, J.D., Bilbao, M.N., Kasabov, N.. Evolving spiking neural networks for online learning over drifting data streams. Neural Networks 2018;108:1–19.

PUBLICATION 3

CANDYMAN: Classifying Android malware families by modelling dynamic traces with Markov chains

(IJ-3) **Martín, Alejandro**; Víctor Rodríguez-Fernández & David Camacho: "CANDYMAN: Classifying Android malware families by modelling dynamic traces with Markov chains." *Engineering Applications of Artificial Intelligence*, Volume 74, 2018, Pages 121-133, DOI: 10.1016/j.engappai.2018.06.006
Impact factor = 2.819 (JCR, 2017) [Q1, 33/132, Computer Science, Artificial Intelligence].

- **Contributions of the PhD candidate:**
  - First author of the article.
  - Contributions made in the conception of the presented idea.
  - Contributions made in the design of the classification mechanism presented.
  - Implementation of the experiments using machine learning techniques.
  - Contributions made in the design and execution of the experiments.
  - Co-author of the interpretation and discussion of results provided.
  - Co-author of the manuscript, figures and tables presented.

Engineering Applications of Artificial Intelligence 74 (2018) 121–133

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

CANDYMAN: Classifying Android malware families by modelling dynamic traces with Markov chains

Alejandro Martín, Víctor Rodríguez-Fernández, David Camacho *

Universidad Autónoma de Madrid (UAM), 28049, Madrid, Spain

ARTICLE INFO

Keywords:
Android malware
Dynamic analysis
Classification
Deep Learning
Markov chains

ABSTRACT

Malware writers are usually focused on those platforms which are most used among common users, with the aim of attacking as many devices as possible. Due to this reason, Android has been heavily attacked for years. Efforts dedicated to combat Android malware are mainly concentrated on detection, in order to prevent malicious software to be installed in a target device. However, it is equally important to put effort into an automatic classification of the type, or family, of a malware sample, in order to establish which actions are necessary to mitigate the damage caused. In this paper, we present CANDYMAN, a tool that classifies Android malware families by combining dynamic analysis and Markov chains. A dynamic analysis process allows to extract representative information of a malware sample, in form of a sequence of states, while a Markov chain allows to model the transition probabilities between the states of the sequence, which will be used as features in the classification process. The space of features built is used to train classical Machine Learning, including methods for imbalanced learning, and Deep Learning algorithms, over a dataset of malware samples from different families, in order to evaluate the proposed method. Using a collection of 5,560 malware samples grouped into 179 different families (extracted from the Drebin dataset), and once made a selection based on a minimum number of relevant and valid samples, a final set of 4,442 samples grouped into 24 different malware families was used. The experimental results indicate a precision performance of 81.8% over this dataset.

1. Introduction

It is well known that an important amount of effort is dedicated to combat the huge number of malicious software samples which are detected every day. This causes that new malware samples are incrementally improved with the aim of bypassing the last counter-measures implemented by Anti-Virus engines, anti-malware detectors, and application stores. This situation represents a vicious circle: a step forward by one side forces the other side to go further. Within the scope of malware that is designed to attack mobile platforms, Android represents a target especially elected for implementing new malware, mainly due to some of its attributes, such as the large number of devices running this operating system or the high degree of freedom that offers to the developer when creating new software.

Different techniques have been developed to extract representative behavioural patterns in Android applications, which are later used to build models able to discern between *malware* and *benignware*, or between different families of malware. These techniques are grouped into two broad categories: **static analysis** and **dynamic analysis**. While the formers are focused on those features reported by code inspection, the second ones opt for monitoring the actions performed by the application while it is being executed. Both types of analysis feature some advantages and disadvantages. While static analysis is fast, it is also very vulnerable to code transformations which hinder the extraction of details revealing a malicious payload within the application. On the other hand, dynamic analysis offers a thorough mechanism for analysing applications, but it is much more computationally expensive (Faruki et al., 2015).

Both analysis approaches are often combined with Machine Learning methods to build classification and detection tools that receive as input a feature vector representing a target application and deliver a label, classifying it as benignware or malware. Typically, these Machine Learning methods include Decision Trees (Perdisci et al., 2008), Support Vector Machines (Gorla et al., 2014), Regression based methods (Ham et al., 2013), K-Nearest Neighbours (Chen et al., 2016), or Artificial Neural Networks (Dahl et al., 2013) among others (Shabtai et al., 2009). When dealing with multi-class datasets and when the number of samples representative of each class is not proportional, imbalanced learning algorithms are the most appropriate procedure to take. Under-sampling,

* Corresponding author.
E-mail address: david.camacho@uam.es (D. Camacho).

https://doi.org/10.1016/j.engappai.2018.06.006
Received 31 July 2017; Received in revised form 20 June 2018; Accepted 24 June 2018
Available online 29 June 2018

over-sampling, hybrid sampling or ensemble sampling are techniques employed by these algorithms.

However, in the last decade, Deep Learning techniques are playing an increasingly important role, due to the possibilities offered by the newest high performance computation infrastructures. Deep Learning entails a new paradigm for solving complex problems. Its essence dates back to the well known Artificial Neural Networks (ANN), which were originated after proposing some concepts in the 1940s based on some basic ideas of the biological brain operation (Goodfellow et al., 2016). In the last decade, methods based on Deep Learning are taking a huge repercussion due to their outstanding ability for dealing with problems that have traditionally been performed by humans, such as those related to recognition tasks, image analysis or pattern extraction. The number and complexity reached by the newest malware forces to explore new advanced techniques able of analysing and tackling the different problems caused, since the huge amount of malware make impossible to human engineers to perform traditional software analysis for every sample.

When the dynamic behaviour of any malicious sample is analysed as an ordered *sequence* of states, it is possible to model the malware behaviour using techniques from the field of sequential modelling. In this context, Markov Models (Fink, 2014) are widely used for creating both descriptive and predictive models of sequence of states, assuming that the probability of being in a specific state only depends on the previous states. Although there exist a large variety of Markov Models researched so far (as Hidden Markov Models (Visser, 2011), or Double Chain Markov Models (Berchtold, 2002) among others), this work will be focused on the traditional and well-known Markov chains. By using these models, it is possible to represent each malware sample by taking into account the transitions between consecutive states.

In this paper we present a tool named CANDYMAN, which combines dynamic analysis with a Markov chain-based representation, in order to model the behaviour of individual applications from different malware families. Once all the applications are represented in the same feature space using the proposed model, based on the combination of Markov chains and several malware analysis tools as DroidBox,[1] and monkeyrunner[2] which allow to generate a *state sequence* representation of any app considered, it is possible to apply different Machine Learning classification algorithms, including Deep Learning techniques, to test their ability at classifying accurately malware into their respective families.

The main contributions of this paper can be summarised as follows:

- From a collection of 5560 malware samples grouped in 179 different families extracted from the Drebin dataset (Arp et al., 2014), and once made a preprocessing selection of relevant samples, a final set of **4442** samples grouped into 24 different malware families has been analysed. All these samples have been analysed with the DroidBox tool, which performs a dynamic analysis process where all actions executed are captured. After each execution, a set of timestamped events of different types is obtained.
- Since the number of possible events extracted by DroidBox is too large, we gather similar events into common states. Thus, we can express the result of the dynamic analysis as an ordered sequence of *states*.
- A new model to represent the *sequence of states* for every malware sample, based on Markov Chains that contain the transition probabilities between consecutive states, is proposed. The information of the Markov Chains will be later used as part of the feature space in the classification process.
- A study of Android malware families is performed based on the space defined by the transition probabilities given by the Markov chains. This allows both, to establish conclusions based on the importance of the different kinds of actions, and to evaluate the proximity between families in the feature space.
- Finally, different experiments have been carried out in order to test the performance of several classical Machine Learning algorithms (Random Forest, Bagging, K-Nearest Neighbours, Decision Tree and Support Vector Machines), in combination with different imbalanced learning algorithms (SMOTE, Random Over Sampler, Random Under Sampler or ADASYN among others) and Deep Learning techniques (using the popular Keras framework), to classify samples into the according malware family.

The rest of the paper has been structured as follows: after a brief description of the backgrounds and previous research related to this work shown in Section 2, we describe the proposed methodology in Section 3, it provides a complete description for the whole process, from the data extraction and preprocessing, to the modelling and classification task. The experimental environment and setup for applying the proposed methodology is described in Section 4, and then Section 5 shows the results obtained in the experimentation that have been carried out. Finally, Section 6 summarises the conclusions extracted from this work, and propose some future research lines of work.

2. Background

This paper exploits the benefits of combining dynamic malware analysis together with Markov chains-based behavioural representation for classification purposes. In this section, previous researches conducted on these topics are summarised. Previously, a brief section is in charge of introducing the context of Android malware families, defining the concepts of Markov models and Markov chains and introducing the concept of Deep Learning and its architectures.

2.1. Android malware families

Android malware samples are usually classified into *families*. Samples belonging to the same family show similar behaviour, exploit the same vulnerabilities and have the same objectives. It is to note that classifying malware in families is different from categorising them according to their type (e.g., a dropper, or a trojan) (Massarelli et al., 2017). Below is briefly described the behaviour of some of the most common malware families, all of them used in this work:

- *FakeInstaller*: FakeInstaller is a widespread mobile malware family. It has spoofed the Olympic Games Results App, Skype, Flash Player, Opera and many other top applications. It sends SMS messages to premium rate numbers, without the user's consent, passing itself off as the installer for a legitimate application. There is a large number of variants for this malware, and it is distributed on hundreds of websites and fake markets.
- *Opfake*: In addition to sending out SMS messages to premium-rate numbers, this malware also monitors SMS messages and is capable of deleting/moving messages based on the originating phone numbers and message content.
- *Plankton*: Plankton variants silently forward information about the device to a remote location. In addition, they download an additional file onto the device.
- *GinMaster*: It steals confidential information from the device and sends it to a remote website.
- *Iconosys*: It may leak the information that it requested from users during the registration to other unintended recipients.
- *Kmin*: These malware variants display a message as a decoy, while silently performing multiple malicious routines, including sending SMS messages to a premium-rate number and downloading and installing an additional application onto the device.

[1] https://github.com/pjlantz/droidbox

[2] https://developer.android.com/studio/test/monkeyrunner/index.html

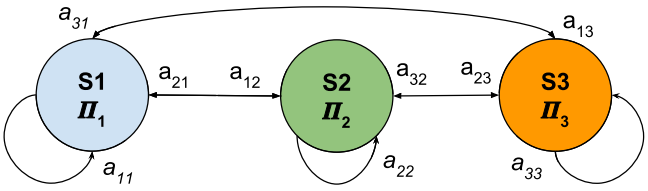


Fig. 1. Example of a 3-state Markov chain. The values of a_{ij} represent the state transition probabilities, and the values of Π_i the initial state probabilities.

- *FakeRun*: It is a malware that deceives users into raising its rating on Google Play. It pretends to be an ad-module stopper but in fact, it includes several ad-modules.

2.2. Markov models & Markov chains

Generally speaking, *Markov models* are stochastic models mainly used for the modelling and prediction of sequences of symbols, and time series in general (Fink, 2014). All of them follow the so-called *Markov property*, i.e, the assumption that future states in a sequence depend only on the current state, not on the events that occurred before it. This type of models have been successfully applied in different topics, such as activity recognition (Duong et al., 2005), modelling of human behavioural patterns (Rodríguez-Fernández et al., 2016a, b) and failure prediction (Salfner and Malek, 2007). There are a wide variety of Markov models that have been researched and applied over decades. We can classify them based on:

1. The representation of time in the sequences, i.e, whether the time is divided into discrete time steps (discrete-time models) or it is considered as a continuous variable (continuous-time models) (Anderson, 2012).
2. The state space of the process we are modelling, whether it is a finite or countable state space (which is most common), or it is a continuous or general state space (Karatzas and Shreve, 2012).
3. Whether every sequential state is fully observable (an observation *is* a state) or it is partially observable, i.e, the observations produced in a state are related to it, but they are insufficient to fully determine it, as it happens in Hidden Markov Models (Visser, 2011).
4. Whether the system or process modelled is *autonomous*, i.e, only dependent on the system state, or it is to be adjusted based on some controls or actions applied to the system, as it happens in Markov Decision Processes (Puterman, 2014).

In this work we are focused on one of the simplest Markov Models, the *Discrete-Time Markov Chains* (or just Markov Chains) for a finite state space. They are characterised by a set of N discrete states $S = \{S_1, S_2, \ldots, S_N\}$ which follow the *first-order Markov property*, namely that the probability of moving to the next state depends only on the present state and not on the previous states. The transition probabilities between the states of a Markov chain are denoted by a square matrix A, with entries:

$$a_{ij}(t) := P(s_{t+1} = S_j | s_t = S_i), \quad 1 \leq i, j \leq N \tag{1}$$

This is a stochastic process, so we have that $\sum_{j=1}^{N} a_{ij} = 0$ for all $1 \leq i \leq N$. Also, we need to specify the set of initial state probabilities, Π, defined as:

$$\Pi_i := P(s_1 = S_i), \quad 1 \leq i \leq N \tag{2}$$

A graphical view of a Markov chain is shown in Fig. 1.

2.3. Imbalanced learning methods

One of the most common problems when performing multiclass classification is the possible lack of a representative amount of samples of a certain class, which causes an unbalanced dataset. When dealing with this kind of datasets, it becomes necessary to use imbalanced learning algorithms (He and Garcia, 2009), which allow to preprocess the training set in order to build a new set of vectors which can later be used to build a classifier that creates a wider separation between samples belonging to different classes. These algorithms apply different techniques which can be grouped into different categories. *Under-sampling* makes reference to those techniques that remove samples from classes with high representation in order to create an homogeneous set of samples representative of all families. Random Under Sampling is one of the most used techniques of this kind. *Over-sampling* techniques follow the opposite direction, they generate new examples by duplicating existing ones, mostly of the minority classes. An example of this technique is the Random Over Sampling algorithm. There can also be found *hybrid* approaches, where the two techniques mentioned above are combined (CSeiffert et al., 2009).

2.4. Deep learning

Deep Learning is one the most promising subfields of Artificial Intelligence. Although its fundamental ideas were posed a few decades ago, under the term *cybernetics* in the 1940's or *connectionism* in the 80's (Goodfellow et al., 2016), the new possibilities offered by the newest hardware make possible to use these models to solve complex and varied problems.

The basic and most important model under the concept of *Deep Learning* is represented by the feedforward networks, where layers of neurons are concatenated in order to build a large set of connections, where the information is only transmitted in one direction and whose initially random weights evolve throughout the training process in order to infer knowledge. A possible improvement consists in introducing Dropout layers (Srivastava et al., 2014), where a set of neurons is disabled with the goal of avoiding overfitting.

Recurrent neural networks offer an interesting procedure when dealing with time sequences (Lipton et al., 2015). The difference lies in introducing cycles, so each neuron keeps certain information. Long short-term memory networks (Gers et al., 2002) are a variant of recurrent neural networks, aimed to learn longer time spaces.

Convolutional neural networks (Krizhevsky et al., 2012) have posed as a powerful mechanism to solve complex problems mainly related to image recognition. Typically these models consist on a sequence of layers with different purposes. For instance, the convolutional layer applies several filters to the features space in order to focus on a reduced set of information. Pooling layers combine the information received from the previous layer in order to build a new representation. Fully connected layers are in charge of inferring knowledge once the input has been processed by the previous layers.

2.5. Related work

Malware designed for the Android platform presents a real and complex problem. To tackle it, multiple approaches have been developed, and they are primarily categorised into two different types of analysis: static and dynamic analysis. Regarding static analysis, various works have studied different methods and different types of features, such as API calls (Martín et al., 2016c), Android Intents (Feizollah et al., 2017) in combination with Android Permissions, showing that both features allow to build fast and reliable detection systems, information reported by the Android Manifest file (Sanz et al., 2013), API calls graphs (Zhang et al., 2014), inter-process communications (Xu et al., 2016) or strings (Martín et al., 2016b).

All these features allow to build deep search spaces where malware can be represented. In order to extract relations from this space and with the goal of developing accurate detection tools, different techniques have been employed, many of them based on Machine Learning techniques. Thus, techniques such as Evolutionary Algorithms (Martín et al., 2016a), Support Vector Machines (Gorla et al., 2014), Regression based methods (Ham et al., 2013) or K-Nearest Neighbours (Chen et al., 2016) among others have been used in this domain.

In contrast, this paper focuses on the second ones, which involve greater complexity, but allow to generate a deeper model representing the behaviour and intentions of a target application. There are different tools focused on dynamic analysis, such as Crowdroid (Burguera et al., 2011), which monitors API calls at the kernel level or DroidScope (Yan and Yin, 2012), which monitors at hardware, OS and Dalvik Virtual Machine level.

A dynamic analysis typically involves applying a representation technique to convert a sequence of events into a model which characterises the behaviour of the app. Stochastic models have been widely applied for this purpose, as it is the case of Markov models. For instance, they have been used as a statistical approach for distinguish between user initiated applications and malicious ones. Markov models are used to generate behaviour graphs based on GUI interactions (Xie et al., 2010). This approach is, however, focused on some old linux-based and symbian devices.

With a similar goal, these models have been applied to model malicious and benign Android Intents patterns, captured through the ADB tool. The stochastic models are used to calculate the proximity of new samples to a previously generated model representing malware samples (Chen et al., 2014). Markov models have also been used in combination with behaviour traces extracted with the DroidBox framework with the goal of studying how the malicious code of a malware is triggered (Suarez-Tangil et al., 2014), which sometimes only occurs when a restrictive set of conditions are fulfilled. In this case, Markov models allow to minimise the search space, employing an effective representation to model system events.

Recently, Hidden Markov models (HMMs) and also structural entropy based methods have been applied for modelling the sequence of instructions that better fit the behaviour of an application using opcodes as states (Canfora et al., 2016). The HMM allows to compare new sequences of opcodes to deliver a label (benignware or malware) as result. The authors show the results for different Android malware families. In another research, sequences of API calls are extracted using static analysis, building a behaviour representation by using a Markov model for each app. These models allow to train Machine Learning algorithms for building classification tools (Mariconti et al., 2016).

Therefore, the literature has demonstrated how Markov models are able to generate useful representations of the behaviour of malicious, or benign, software. Multiple times have been combined with classical Machine Learning classification algorithms, such as those based on decision trees, to build accurate classification tools, as it is the case of MaMaDroid (Mariconti et al., 2016). But newer techniques like Deep Learning have also been explored, as it is the case of DroidDetector (Yuan et al., 2016), which employs Deep Learning to build a detection tool based on a combination of static and dynamic features, involving permissions, API calls and dynamic traces extracted with DroidBox. DroidDelver (Hou et al., 2016b) uses Deep Belief Networks over blocks of API calls to categorise applications. Deep4MalDroid employs Stacked AutoEncoders (SAEs) to detect malicious patterns based on Linux kernel calls (Hou et al., 2016a).

In this paper we draw a novel combination of dynamic analysis and Markov chains to model the sequences of states incurred during the execution of a malware sample and thus obtaining a deep representation for each sample. Later, both classical machine and Deep Learning algorithms are tested over a malware dataset divided by families to test their ability to discriminate between the different behaviours found. This combination of dynamic traces modelled as Markov chains and Deep Learning allows to accurately infer the rules that divide the space where the different malware families are defined.

3. Modelling Android malware dynamic behaviours

In this section, we describe the whole process for representing a malicious sample as a Markov chain, and how to apply this technique for classifying a whole dataset of malicious samples into a set of families. A general and graphical view of this process can be seen in Fig. 2.

3.1. Extracting dynamic behaviours

Although static malware analysis techniques offer a fast and useful mechanism to extract significant information from a suspicious application, they do not allow to model the real behaviour of an application when running in a real environment. Different techniques, which fall within the scope of the so-called obfuscation techniques, can be deliberately employed with the aim of shaping the code into a new different scheme, with different classes and methods names or including sections of code which are never reached. For example, a series of useless system calls can be included into a section of the code which is executed based on a condition, whose evaluation is always false when is executed. In contrast, a dynamic analysis allows to model the behaviour of a suspicious application based on the actions actually exhibited in a controlled execution environment. One of the major advantages of this kind of analysis lies in that it is possible to capture events which are triggered after certain conditions are met.

In this work, the DroidBox tool has been used to extract different information when an Android application is executed in an emulator provided by the Android studio framework. This tool converts an Android emulator into a mobile sandbox which monitor a series of actions and events which occur while executing the application. All these events are captured and sent to the *logcat* register, which serves as a pipe to monitor all the events by an external application, independent from the emulator.

In order to control the application from outside of the emulator, DroidBox makes use of the *monkeyrunner* tool. This tool enables to send actions such as launching applications or sending keystrokes. At the same time, this is useful to force the application to trigger certain events which usually only become visible when the app is really being used by a human. Thus, *monkeyrunner* is executed simultaneously with the application, and a set of random actions over the emulator is launched, including varied movements simulating the movement of a finger on the screen in different directions, touching different buttons, both physical and those displayed by the app, or introducing text in a text field if present.

The information gathered by this framework includes incoming and outgoing network data, file read and write operations, started services and loaded classes, information leaks via the network, file and SMS services, circumvented permissions, cryptographic operations, broadcast receivers and sent SMS and phone calls. All this data is reported in a JSON file which is delivered after an Android application has been analysed with DroidBox. This file is organised in multiple fields according to the different categories of events. Each event includes specific details such as the name of an started service or the timestamp of the event, measured in milliseconds from the beginning of the execution.

3.2. Modelling dynamic behaviour with Markov chains

Once the dynamic behaviour of a malware sample has been processed into a JavaScript Object Notation (JSON) file, the next step consists in expressing that behaviour in terms of a Markov model, more specifically, in terms of a Markov chain. A graphical overview of this process is given in Fig. 3.

The key issue here is that every action made by a malware sample is recorded into the JSON file along with a timestamp, which allows us to sort them and express the behaviour of the app in terms of a *state sequence*. Then, the state sequence is modelled as a *first-order Markov process*, i.e, we make the assumption that the value of a state in the

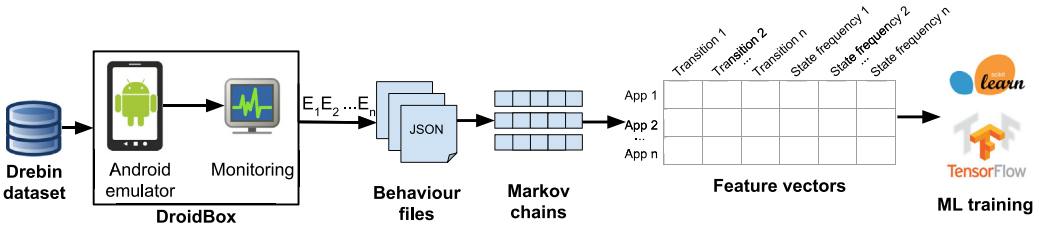


Fig. 2. Workflow of CANDYMAN: a dataset of applications received as input is analysed with DroidBox. Then the sequence of events delivered are modelled through a Markov chains that are finally used as feature vectors of the applications, and serve as input to a set of Machine Learning classification algorithms.

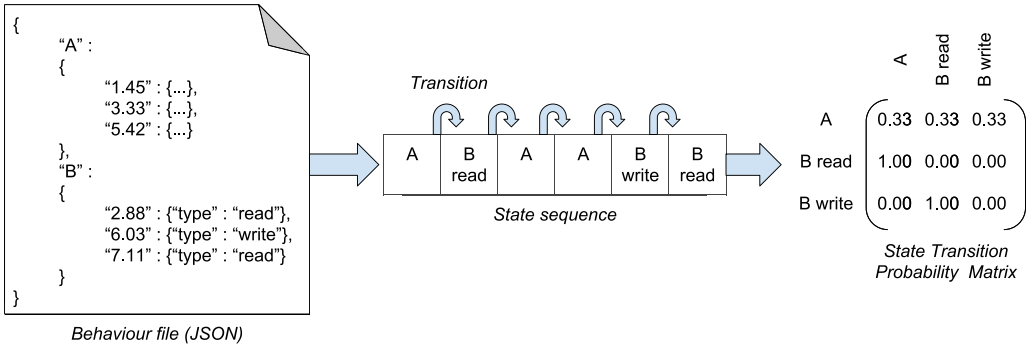


Fig. 3. Transforming a behaviour file into a Markov transition matrix. Every record under a field of the behaviour file (A or B in this figure) is identified with a timestamp, which can be used to sort the information contained in the records in terms of a *state sequence*. The transitions between states in this sequence can in turn be modelled as a transition probability matrix, or Markov transition matrix.

sequence depends only on the value of the previous state. Thus, we can analyse the transitions between adjacent states and create a *state transition probability matrix*, A, where a_{ij} represents the probability of going from state i to state j.

The set of states visited by a malware sample depends on the events recorded in the behaviour file. In this work, we define the following states, gathering the different fields contained in that JSON file into a common set of states:

- `fdaccess`: This field describes every file access made by the malware sample. Every event recorded under this field will be mapped to the state `fdaccess [base-path-`δ`]`, where `base-path-`δ is the base path of that file access, trimmed to depth δ. For instance, a file access to the path `/dev/input/event0` will be mapped to the state `fdaccess /dev/`, providing that $\delta = 2$.
- `dataleaks`: This field describes leaks of sensitive information stored on the phone by the user. This involves contacts, calendar entries, emails, call history and SMS data. Additionally, phone-specific information like IMEI, phone number, installed applications and GPS coordinates is seen as sensitive data. Every record under this field will be mapped to the state `dataleaks [type] [tags]`, where `type` identifies the context of the data leak (outgoing network data, write operations to files or outgoing SMS), and `tags` contains labels with the type of information that has been stored in the data leak. As an example, a network data leak storing the IMEI of the phone would be mapped to the state `dataleaks Network TAINT_IMEI`.
- `dexclass`: This field contains information about the external calls made by the malware sample, in order to execute code that is not installed as part of the application. Every record under this field will be mapped to the state `dexclass [base-path-`δ`]`, where `base-path-`δ is the path of the *.apk* executed, trimmed to depth δ. For instance, an execution of the APK file `/system/app/QuickSearchBox.apk` will be mapped to the state `dexclass /system/`, in case that $\delta = 2$.
- `opennet`: This field describes every network connection opened, giving information about the remote destination, such as the host address and the port. Every record under this field will be mapped to the state `opennet`, i.e, we gather every action of opening a connection into the same state, regardless the specific connection details.
- `sendnet`: This field contains one record for every chunk of outgoing data transferred between the malware sample and a remote network destination, except the case when the data contains leaks (in that case the traffic is saved under the `dataleaks` field). Every record under this field will be mapped to the state `sendnet net write [tags]`, where `tags` contains labels with the type of information sent.
- `recvnet`: This field contains one record for every chunk of incoming data transferred between a remote network destination and the malware sample, except the case when the data contains leaks (in that case the traffic is saved under the `dataleaks` field). Every record under this field will be mapped to the state `recvnet net read`, i.e, we gather every instance of incoming network traffic into the same state, regardless the specific traffic details.
- `servicestart`: This field monitors every API call executed by the malware sample. Every record under this field will be mapped to the state `servicestart [base-name-`δ`]`, where `base-name-`δ is the name of the API call, up to the namespace of depth δ. As an example, an API call to `com.android.music.MediaPlaybackService` would be gathered into the state `servicestart com.android`, in case that $\delta = 2$.
- `sendsms`: Sending SMS messages is a common issue for malware due to SMS messages can be performed unnoticed when the device is not used, and even in the background without any indication of this action. This fields records information about every sent SMS, including the destination number and the contents of the message. Every record under this field will be ampped into the state `sendsms`, i.e, we gather every action of sending a SMS into the same state, regardless the specific SMS details.

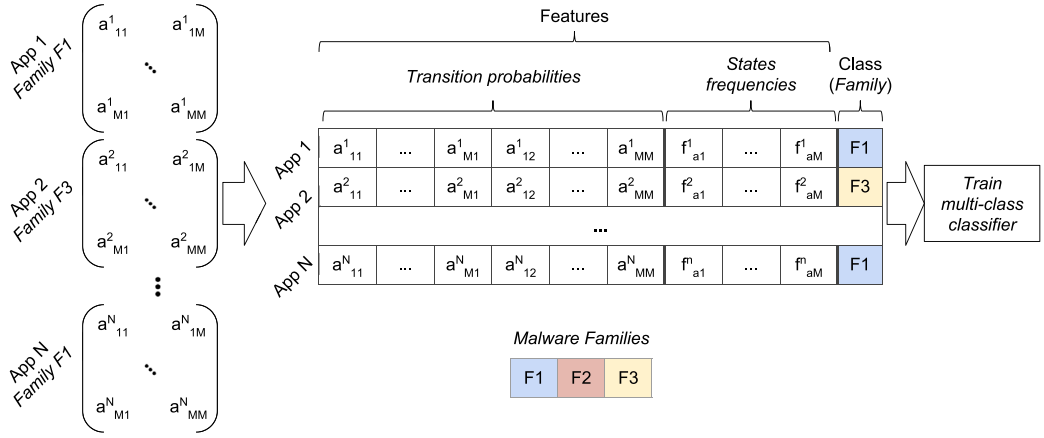


Fig. 4. Malware family classification using Markov chains. This example shows how a set of N Markov chains, each of them represented by a $M \times M$ transition probability matrix and the frequency of each different state, are gathered into a $N \times (M^2 + M)$ feature matrix. This feature matrix is used to train a multi-class classifier that predicts the family of a malware sample.

3.3. Using Markov chains to perform malware family classification

Although the creation of a Markov chain for modelling the behaviour of a malware sample is useful by its own in a descriptive way (i.e, one can analyse the values of the transition probability matrix to extract some behavioural patterns), the aim of this work consists in using those Markov chains in a classification process in which the family of a malware sample is predicted.

One important issue to consider is the fact that, given N different malware samples, the resulting Markov transition matrices for each of them will possibly contain different states, and even different number of states, which makes more difficult to compare the samples from a classification point of view. To solve this issue, first we compute the state sequence for each malware sample (Fig. 3, step 1), and then, we define the *common state space* as the union of the state space of each sequence. Thus, the rows and columns of the Markov transition matrices will be identified by this common state space, and will be the same for every malware sample in the dataset.

Although Markov chains allow to build a useful representation of the behaviour of each sample, where the transitions between different events are modelled, they do not consider the individual importance of each state. For instance, a particular family can be characterised by a considerable number of file operations and SMS sent. That is why the frequency of each state has also been included into the model, ensuring that every application is not only represented by the transitions probabilities between each possible pair of states but also by the frequencies itself of the states.

In Fig. 4 we show graphically the process to format the Markov chains of N malware samples into a format suitable for a classification process, i.e., into a set of feature vectors. As it can be seen, what we do is to *vectorise*, by columns, the contents of the transition probability matrix of each Markov chain. Let M be the size of the common state space for all our malware samples, each transition probability matrix will be $M \times M$. Adding also the frequency of each state, the total size of each feature vector will be $M^2 + M$. Normally, when gathering all the feature vectors, we will find many zero-features, i.e, many transitions with zero-probability in every sample. In order to avoid useless features in the classification process, columns with less than ϵ non-zero values are removed from the feature space. The value of the hyperparameter ϵ will be fixed in the experimentation section.

Once each malicious sample has been represented as a feature vector, it is possible to train different Machine Learning classification algorithms. We have employed both classical algorithms, imbalanced learning techniques and Deep Learning architectures. Regarding classical Machine Learning algorithms, we used a Decision Tree based algorithm, Random Forests, K-nearest Neighbours, a Bagging classifier composed of Random Forest classifiers and Support Vector Machines with linear, rbf and sigmoid kernels. In the case of imbalanced learning algorithms, we employed a wide set of over-sampling, under-sampling and hybrid techniques.

Deep Learning has also been explored testing different combinations of fully connected layers with different number of neurons, Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) and Convolutional Neural Networks (CNNs). Since the problem being addressed presents a multi-class and unbalanced scenario, when using fully connected layers, these have been disposed in cooperation with dropout layers, which have proven to be very effective at improving the performance, specially when dealing with overfitting effects (Srivastava et al., 2014).

3.4. Applicability to other platforms

Although the methodology presented here has been proposed specifically for Android and the *Droidbox* tool, the underlying concepts can be applied straightforwardly to any other OS, such as *Windows* or *Mac OS*, as long as a tool for extracting dynamic behaviour is available. At the very least, the tool must extract, for any behavioural aspect of an application, the timestamp and the name of the event extracted. This information will allow the creation of state sequences, as seen in the previous sections.

There are currently numerous examples of dynamic analysis tools that has been used in other situations and platforms, such as *Cuckoo Sandbox* (Qiao et al., 2013), *ProcMon* (Wagner et al., 2015), or *MALHEUR* (Rieck et al., 2011). All these tools are applicable within the proposed methodology, as long as a set of states and a set of classes to predict are defined.

4. Experimental setup

This section describes the experimental setup for validating the proposed methodology. This setup includes different components, such as the dataset used, the details related to the dynamic analysis, the Markov chains construction process, and the parametrisation of the Machine Learning algorithms trained.

4.1. Dataset

In order to test the capability of the method designed to classify malware into their respective Android malware families, several experiments have been performed using the Drebin dataset (Arp et al., 2014), a collection of **5560** malware samples grouped in **179** different families where initially gathered. Only those families with at least 20

Table 1
Distribution of the malware families and samples from the Drebin dataset finally used in the experiments.

Family	No. samples	Family	No. samples	Family	No. samples
Adrd	79	FakeRun	60	Jifake	28
BaseBridge	311	Gappusin	46	Kmin	95
Boxer	25	Geinimi	80	MobileTx	68
DroidDream	77	GinMaster	334	Opfake	597
DroidKungFu	658	Glodream	68	Plankton	478
ExploitLinuxLotoor	67	Hamob	26	SMSreg	38
FakeDoc	132	Iconosys	135	SendPay	58
FakeInstaller	904	Imlog	42	Yzhc	36

samples are kept, resulting in a final dataset of **24** different malware families. We have selected 20 as the minimum number of samples per family due to it is used in similar experiments with the Drebin dataset found in the literature (Dash et al., 2016). Furthermore, we removed invalid applications by using the Androguard (Desnos et al., 2011) tool and discarding those that could not be later executed in the Android emulator. The final dataset employed in the experiments stores a collection of **4442** samples.

4.2. Dynamic traces extraction

Each app of the dataset was executed in an emulated Android device, managed with the DroidBox tool, to extract sequences for the different events occurred in the system. A maximum time of 300 s was fixed to execute each application in which 10,000 pseudo-random events were triggered using the *Monkey* tool aiming to simulate different interactions with the application. Once the app has been running for the defined period, the emulator is terminated and a report of all the events is saved. Each time a new app is executed, the emulator starts from a clean Android image, thus always starting from the same conditions. Several applications were removed in this step due to some errors during their execution. As it was described above, the final dataset of samples executed contains **4442** samples grouped into **24** different malware families, according to the distribution shown in Table 1. The emulated device elected was a Nexus 4 running Android 4.1.1, which is the last version supported by DroidBox.

4.3. Parameter tuning for building Markov chains

As it was mentioned in Section 3, there are some fields in the behaviour file of a malicious sample, such as `fdaccess` or `dexclass`, that contain information about a specific path to which the sample is accessing. In order to avoid too many states in the Markov chains, those paths are considered only up to a depth level given by the parameter δ. In this experiment, we set $\delta = 1$, so, as an example, the records `fdaccess read /proc/001/` and `fdaccess read /proc/002/` will belong to the same aggregated state: `fdaccess read /`.

On the other hand, the hyperparameter ϵ, which regulates the minimum number of samples in which a transition must occur in order to be included in the feature space, was experimentally set to 10.

4.4. Machine learning algorithms parametrisation

The *scikit-learn* library[3] was used to train four classical Machine Learning algorithms (Random Forest, Decision Trees, Bagging classifier, KNN and SVM). Random Forest was executed using 100 internal trees, as it is also the case of the Random Forest used in the Bagging classifier. Decision Tree based algorithm and K-Nearest Neighbours were executed with the default parameters, where a total of 10 executions were run for each algorithm. SVMs were executed with linear, rbf and sigmoid as kernel functions. The *scikit-learn* library was executed in combination with the *imbalance-learn* library[4] in order to run the imbalanced learning algorithms.

Table 2
Parameters tested for each Deep Learning model trained.

Deep Learning model	Range of parameters
Fully connected + Dropout	No. Neurons = [50, 100, 300, 500] No. Layers = [2,3,4,6]
CNN	No. Filters = [10, 30, 50] Filter length = [5, 10, 15] Pooling size = [2, 5, 10]
RNN LSTM	No. units = [2, 5, 10, 20] No. units = [2, 6, 10]

On the other hand, the *Keras* framework was used[5] to train different deep neural architectures. Keras is a framework written in Python that serves as a high level API to define architectures, which can be trained using TensorFlow (Abadi et al., 2016) or Theano,[6] two of the most popular Deep Learning libraries. All the experiments have been executed using the TensorFlow library, due to the large amount of documentation available and its high performance.

When fully connected layers were used, two different types of layers were used in Keras, the *Dense* layer, which is a classical fully connected layer with n neurons and the *Dropout* layer, where a set of neurons are dropped during the training step, based on a probability p, with the aim of reducing the overfitting effect. In the experiments, this probability was fixed to $p = 0.2$. For building a Convolutional Neural Network, a combination of *Convolution1D* (for dealing with one dimensional features space), *MaxPooling1D*, *Flatten* and a final *Dense* layer were used. In the case of the Long Short-Term Memory model, a *LSTM* and a *Dense* layers were involved. For a simple Recurrent Neural Network model, only the *SimpleRNN* Keras layer was necessary. Table 2 shows the range of parameters tested for each possible network model.

The remaining parameters were configured as follows: *Adam* was used as the optimiser for the neural network; every activation function was a sigmoid function (except in the Recurrent Neural Network model, where different functions were tested due to its importance in this model); a normal function was used for the initialisation process; and the training instances were grouped in packages of 10 examples (the batch size parameter of Keras).

4.5. Design of experiments

The different components comprising the feature space described in Fig. 4 allow to design several experiments, the three types of features used, transitions probabilities, states frequencies, and aggregated state frequencies grouped, have been used to define five different kind of experiments:

- **Experiment 1: Transition probabilities.** This experiment aims to study whether the Markov chain-based representation of malware samples is able, by its own, to define a feature space that effectively discriminates malware families.

[3] http://scikit-learn.org/stable/

[4] http://contrib.scikit-learn.org/imbalanced-learn/stable/index.html

[5] https://keras.io/

[6] http://deeplearning.net/software/theano/

- **Experiment 2: State frequencies.** This experiment tries to show whether the distribution of state frequencies is able, by its own, to discern between families of malware and how it performs in comparison to the previous experiment.
- **Experiment 3: Aggregated state frequencies.** This experiment is similar to the previous one, but instead of using the whole state space of the dataset, we aggregate those states belonging to the same "superstate". As an example, close states such as `fdaccess read /proc/` and `fdaccess read /dev/` will be aggregated to the superstate `fdaccess`. This experiment allow us to determine whether taking into account a smaller number of states provides better classification results.
- **Experiment 4: Transition probabilities & state frequencies.** Here, a combination of transitions probabilities and states frequencies is evaluated, which allows to study how the different Machine Learning algorithms behave when a bigger feature space is employed.
- **Experiment 5: Transitions probabilities & aggregated state frequencies.** Finally, a combination of transitions probabilities and aggregated state frequencies is also tested.

4.6. Data-related settings

For both, the classical Machine Learning algorithms and Deep Learning configurations, 70% of the data was dedicated to training and 30% for testing. In the case of Deep Learning, 10% of the training dataset was used to monitor the learning curve, and stop the process when the accuracy is not improved during 50 iterations.

5. Experimental results

In this section, the space built based on the combination of all the possible transitions between events and states frequencies for all the samples is analysed through the application of different Machine Learning algorithms, with the aim of evaluating the degree of discrimination and the performance at allocating samples to their malware family. Furthermore, states frequencies have also been studied grouping the events into the eight kinds of events described in Section 3.2.

5.1. Families space analysis

Modelling malware samples as sequences of events provides a useful way of understanding their behaviour at runtime. This allows to analyse not only how different malware families differ from each other, but also to extract the common patterns shared across all of them. Before studying the different malware families represented in the dataset used, a high-level analysis of the most frequent state transitions has been made.

Fig. 5 shows the distribution of transitions between "superstates". As it can be seen, most of the states related to file read operations (superstate `fdaccess`) are followed by the same type of operation. The same applies to write operations, but with less number of occurrences. There are also important transitions between different events. For instance, a considerable amount of write operations are followed by read operations, and vice versa. Another point to note is high symmetry of the graph, which means that for every states A and B, the probability of going from A to B is very similar to the probability of going from B to A.

Deepening into the space of transitions, Table 3 shows the top 10 most frequent state transitions. As it can be seen, a repeated file read operation (`fdaccess read`) leads this ranking. It can be assumed that this transition represents a common pattern between applications, meaning that is not relevant for distinguishing between families. The second transition is also a self transition, consisting in a repeated write operation. These top two transitions accumulate more frequency than the rest of transitions together.

Table 3
Top 10 count of state transitions in the Drebin dataset, considering $\delta = 1$ for the creation of the state sequences.

Position	Source	Destiny	Count
1	fdaccess read	fdaccess read	288 568
2	fdaccess write	fdaccess write	271 215
3	fdaccess read pipe	fdaccess read pipe	48 716
4	fdaccess write	fdaccess read	25 541
5	servicestart com	servicestart com	23 926
6	fdaccess read	fdaccess write	23 425
7	recvnet net read	recvnet net read	10 703
8	recvnet net read	fdaccess write	9 045
9	servicestart com	fdaccess read	8 989
10	fdaccess read	servicestart com	7 886

In order to evaluate how the malware families are distributed in this space of transitions, Fig. 6 shows a two dimensional projection of all the transitions after applying a Principal Component Analysis (PCA) process. Each family is displayed as a point representing the mean in the projected space, together with lines representing the standard in the two dimensions. This plot is aimed to show how particular families present closer behaviour than others. For instance, it can be seen that there are some families that are represented by a large variety of transitions, such as Hamob, while some other ones are defined in a small region, such as Fake Installer, Opfake or Plankton, which are some of the most prevalent families in the wild. In general terms, families are well distributed over the definition space, although some of them, like Yzhc and SMSreg (represented by X and W in the figure respectively) are located very close to each other. This makes sense due to these families are characterised by a high frequency of accesses to sensitive data and information related to the network state.

5.2. Applying machine learning algorithms

Table 4 shows the average accuracy, F1, recall and precision measures for the five classical classification algorithms trained in the five experiments drawn in the previous section. As it can be seen, grouping the states frequencies by the "superstate" they belong to (Experiment 3) causes a drop in all the measures used by more than 19%. When comparing Experiments 1 and 2, which involve isolated transition probabilities and isolated states frequencies respectively, it can be seen that isolated states allow a better discrimination, reaching a 80.3% in terms of F1 measure. However, the best values are obtained when these two feature spaces are combined (Experiment 4) and a Random Forest classifier is used. If we combine state transitions with aggregated states frequencies (Experiment 5), the four measures tested are slightly worse. SVMs are only competitive when a linear kernel is employed, but it cannot improve the results achieved by the Random Forest algorithm.

Results obtained applying Deep Learning techniques are present in Table 5. The results provided are determined by the best combination of parameters found for each architecture (these combinations can be found in Table 2). For all the experiments, Deep Learning is not able to surpass the results obtained with classical Machine Learning algorithms. It is remarkable that the best results for Deep jLearning are achieved in Experiment 4 using a fully connected model. Regarding the use of Deep Learning with fully connected layers in Experiment 5, an architecture of 2 layers of 300 neurons turned out to be best. In the case of the experiment 4, where the best F1 and Precision measures are obtained, a Convolutional Neural Network with 10 filters of size 15 and with a pooling layer with size 5. The poor results achieved with the RNN and LSTM models are mainly due to the transformations performed when modelling the dynamic traces using Markov Chains, where the original time sequences are no longer present.

The results achieved by Random Forest in comparison to Deep Learning indicates that a logical rule-based approach is able to discriminate better the feature space rather than using a connectionist approach. However, the imbalance among the samples belonging to the different

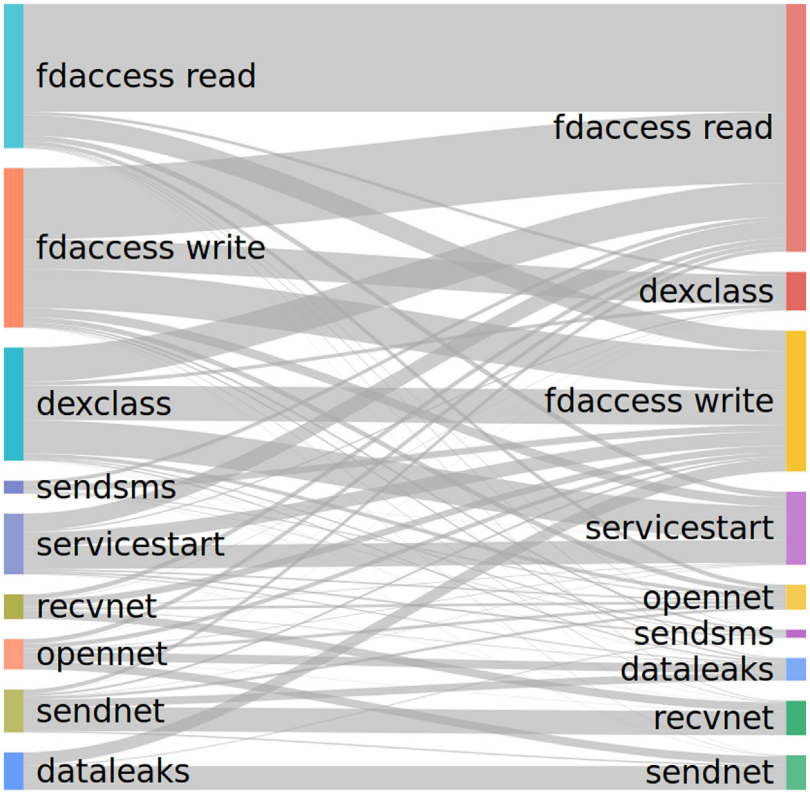


Fig. 5. Flow diagram representing the transition frequencies between the main states defined in this work for the dynamic behaviour of malware samples. For a sake of clarity in the visualisation, close states such as `fdaccess read /proc/` and `fdaccess read /dev/` have been gathered into the same "superstate" `fdaccess`.

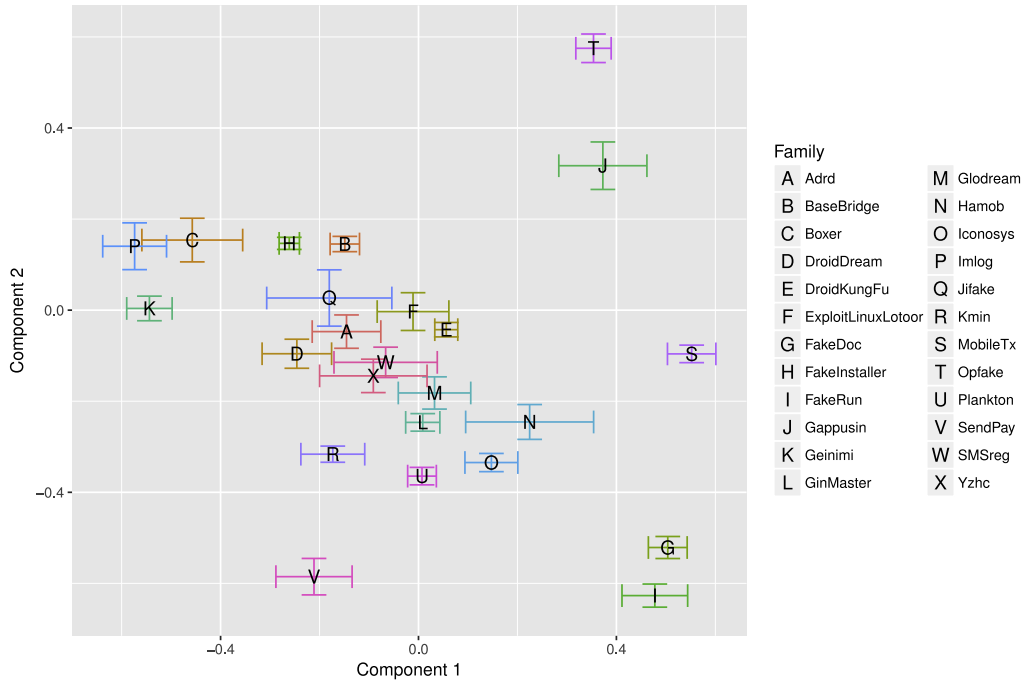


Fig. 6. Distribution of state transitions by family, after applying Principal Component Analysis (PCA) to reduce to 2 the dimensionality of the state transition.

families of malware makes it necessary to study appropriate algorithms to deal with this problem as well. For this reason, we have tested different imbalanced learning algorithms in order to build new datasets able to reach higher precision. Table 6 shows the results in terms of precision (which allows to compare how the different approaches behave when taking into account each family in an individual manner). The best result is achieved in Experiment 4 (combination of transition probabilities and states weights) and using the SMOTE Tomek algorithm in combination with the Random Forest classification algorithm, reaching 81.8% in terms of precision. This algorithm performs an over-sampling process

Table 4
Summary of the results achieved with Bagging, Decision Tree, K-Nearest Neighbours, Random Forest and SVM with different kernel functions classification algorithms for the different experiments.

Experiment	Metric	ML algorithm						
		Bagging	Decision tree	Nearest neighbours	Random forest	SVM linear	SVM rbf	SVM sigmoid
Experiment 1: Transition probabilities	Accuracy	0.799	0.679	0.739	0.801	0.748	0.512	0.426
	F1	0.779	0.678	0.728	0.784	0.728	0.462	0.359
	Precision	0.787	0.681	0.729	0.792	0.738	0.529	0.516
	Recall	0.799	0.679	0.739	0.801	0.748	0.512	0.426
Experiment 2: States frequencies	Accuracy	0.805	0.741	0.708	0.813	0.454	0.288	0.257
	F1	0.793	0.739	0.697	**0.803**	0.414	0.194	0.148
	Precision	0.797	0.74	0.697	0.804	0.614	0.412	0.178
	Recall	0.805	0.741	0.708	0.813	0.454	0.288	0.257
Experiment 3: States frequencies grouped	Accuracy	0.666	0.626	0.563	0.667	0.269	0.23	0.221
	F1	0.654	0.62	0.541	0.659	0.156	0.114	0.098
	Precision	0.655	0.622	0.557	0.658	0.146	0.162	0.137
	Recall	0.666	0.626	0.563	0.667	0.269	0.23	0.221
Experiment 4: Transitions probabilities & states frequencies	Accuracy	0.811	0.723	0.719	**0.818**	0.752	0.489	0.333
	F1	0.792	0.721	0.709	0.802	0.734	0.439	0.255
	Precision	0.801	0.723	0.712	**0.807**	0.745	0.53	0.431
	Recall	0.811	0.723	0.719	**0.818**	0.752	0.489	0.333
Experiment 5: Transitions probabilities & states frequencies grouped	Accuracy	0.808	0.713	0.742	0.815	0.748	0.509	0.422
	F1	0.789	0.71	0.729	0.799	0.728	0.458	0.355
	Precision	0.797	0.71	0.734	0.806	0.736	0.529	0.509
	Recall	0.808	0.713	0.742	0.815	0.748	0.509	0.422

Table 5
Summary of the results achieved with different Deep Learning architectures for the different experiments.

Experiment	Metric	Deep learning architecture			
		CNN	Fully Connected + Dropout	LSTM	RNN
Experiment 1: Transition probabilities	Accuracy	0.76	0.773	0.204	0.204
	F1	0.755	0.767	0.069	0.069
	Precision	0.757	**0.769**	0.041	0.041
	Recall	0.76	0.773	0.204	0.204
Experiment 2: States frequencies	Accuracy	0.752	0.698	0.204	0.204
	F1	0.739	0.672	0.069	0.069
	Precision	0.744	0.699	0.041	0.041
	Recall	0.752	0.698	0.204	0.204
Experiment 3: States frequencies grouped	Accuracy	0.445	0.512	0.204	0.204
	F1	0.39	0.471	0.069	0.069
	Precision	0.403	0.502	0.041	0.041
	Recall	0.445	0.512	0.204	0.204
Experiment 4: Transitions probabilities & states frequencies	Accuracy	0.768	**0.778**	0.204	0.204
	F1	0.764	**0.768**	0.069	0.069
	Precision	0.766	0.768	0.041	0.041
	Recall	0.768	**0.778**	0.204	0.204
Experiment 5: Transitions probabilities & states frequencies grouped	Accuracy	0.762	0.771	0.204	0.204
	F1	0.758	0.757	0.069	0.069
	Precision	0.761	0.755	0.041	0.041
	Recall	0.762	0.771	0.204	0.204

where majority classes are sequentially removed. In comparison to the results achieved when applying solely the Random Forest classifier, this one reduced this value to 80.7% precision.

The confusion matrix obtained after training the Random Forest algorithm is shown in Fig. 7. This figures reveals the most complicated families to discern and how their samples are allocated to other families. For instance, SMSreg presents a behaviour which is difficult to categorise. More than 40% of its samples are wrongly entirely assigned to the Fake Installer family. A similar fact can also be observed in the Yzhc family. For the rest of families, Random Forest achieves excellent results, reaching close to 100% of correctly allocated samples for most of the families. On the other hand, Deep Learning achieves high performance values, although a bit worse than Random Forest.

The results gathered in Tables 4–6 do not show individual execution results, but averages over a total of 10 executions. It is important to analyse the statistical significance between those averages, specially between the different experiments proposed, in order to provide a more robust comparison between them. Thus, a pairwise *Mann–Whitney U test* (Corder and Foreman, 2014) between experiments has been performed over the results of the best classifier found (Random Forest + SMOTE Tomek). The performance metric used in the test is the F1 measure. The results of this pairwise comparison are gathered into a matrix of p-values, shown in Table 7. As it can be seen, Experiment 4, which achieves the best results in the previous tables, reaches statistical significant differences with respect to the rest of the experiments, excepting Experiment 5. With this, we can conclude that the inclusion of transition probabilities in combination with state frequencies, as proposed in this work, improves the performance of the classification process with respect to classic and simplistic approaches where only state frequencies are considered (Experiments 2 and 3).

6. Conclusions

Dynamic analysis offers thorough information of the behaviour of a malware sample. As a result of modelling this information with a Markov chain, samples can be represented by the transition probabilities

Table 6
Summary of the results achieved by different imbalanced learning algorithms in combination with the Random Forest classification algorithm for the different experiments.

Imbalanced learning algorithm	Experiment 1	Experiment 2	Experiment 3	Experiment 4	Experiment 5
ADASYN	0.808	0.803	0.654	0.816	0.806
AllKNN	0.76	0.757	0.6	0.774	0.768
Cluster centroids	0.77	0.703	0.593	0.788	0.781
Condensed nearest neighbour	0.611	0.617	0.453	0.643	0.609
Edited nearest neighbours	0.71	0.691	0.544	0.723	0.714
Instance hardness threshold	0.74	0.75	0.637	0.761	0.741
Near miss	0.45	0.454	0.334	0.404	0.459
One sided selection	0.711	0.717	0.593	0.742	0.698
Random over sampler	0.799	0.796	0.655	0.806	0.804
Random under sampler	0.704	0.718	0.553	0.724	0.725
Repeated edited nearest neighbours	0.708	0.683	0.516	0.722	0.715
SMOTE	0.807	0.805	0.658	0.815	0.813
SMOTE borderline 1	0.804	0.797	0.648	0.808	0.809
SMOTE borderline 2	0.797	0.799	0.647	0.815	0.806
SMOTE svm	0.801	0.8	0.654	0.813	0.812
SMOTE ENN	0.808	0.792	0.654	0.816	0.812
SMOTE tomek	0.809	0.805	0.66	**0.818**	0.813
Tomek links	0.784	0.796	0.656	0.796	0.803

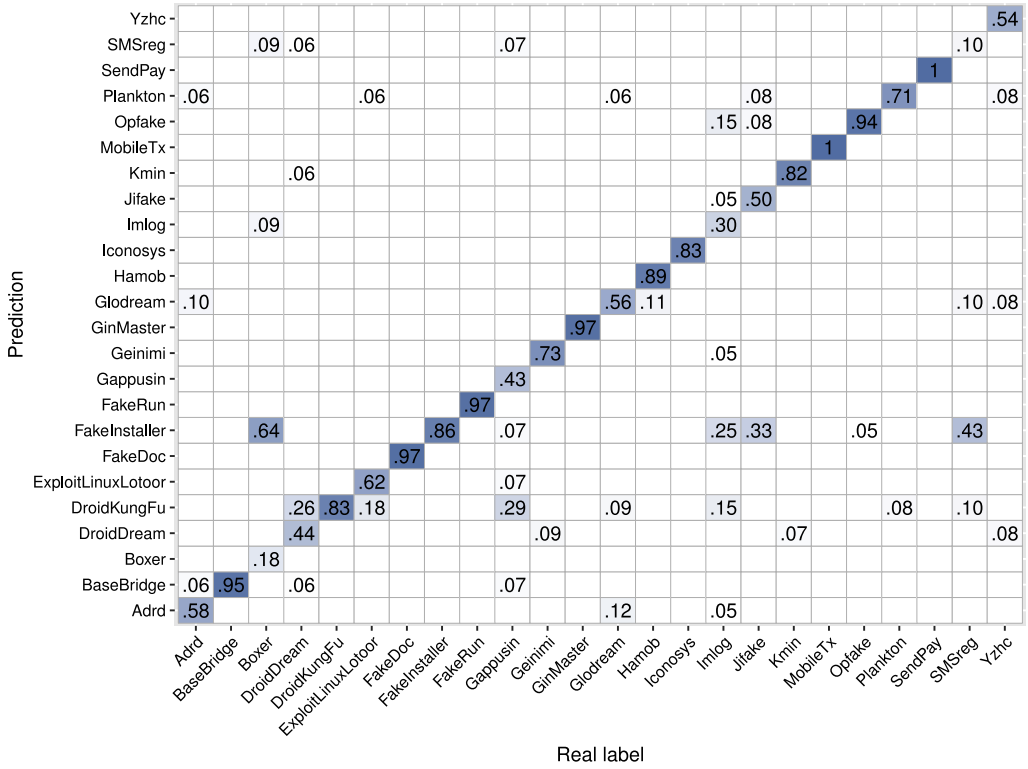


Fig. 7. Confusion matrix of the best execution performed by Random Forest algorithm. Only values greater than 0.05 are shown for a better visualisation.

Table 7
p-values for the pairwise Wilcoxon rank sum statistical tests performed between the different type of experiments, using the results of the F1 measure for the best found classifier (Random Forest + SMOTE Tomek). p-values under 0.05 represent significant differences.

	Ex. 1	Ex. 2	Ex. 3	Ex. 4
Experiment 2	4.35×10^{-1}			
Experiment 3	1.08×10^{-4}	1.08×10^{-4}		
Experiment 4	1.94×10^{-2}	6.30×10^{-3}	1.08×10^{-4}	
Experiment 5	1.06×10^{-1}	3.57×10^{-2}	1.08×10^{-4}	4.35×10^{-1}

between pairs of consecutive states. Several Machine Learning algorithms have been applied to classify malware using this representation, including Deep Learning approaches. The results show that it is possible to discriminate malware families with high accuracy, specially when combining the information from the state transitions with the state frequency distribution. To the best of our knowledge, no previous research has been previously focused on studying the combination of both dynamic analysis and Deep Learning for family classification. As future lines of work, the work presented here will be extended to other varieties of Markov models, such as higher order Markov chains or Hidden Markov Models, to develop a formal comparison among them in terms of predictive performance. At the same time, we would like to explore the possibilities of RNNs and LSTM models when applied directly to the dynamic traces, instead of employing the Markov chains based representation. Finally, other possible interesting research could be to study how to hybridise the features derived from the dynamic behaviour with those derived from static analysis, in order to improve the accuracy of the classifier.

Acknowledgements

This work has been co-funded by the following research projects: Comunidad Autónoma de Madrid under grant S2013/ICE-3095 (CIBERDINE: Cybersecurity, Data and Risks); Spanish Ministry of Science and Education and Competitivity (MINECO) and European Regional Development Fund (FEDER) under grants TIN2014-56494-C4-4-P (EphemeCH), and TIN2017-85727-C4-3-P (DeepBio); and Justice Programme of the European Union (2014–2020) 723180 – RiskTrack – JUST-2015-JCOO-AG/JUST-2015-JCOO-AG-1. The contents of this publication are the sole responsibility of their authors and can in no way be taken to reflect the views of the European Commission.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al., 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, arXiv preprint arXiv:1603.04467.

Anderson, W.J., 2012. Continuous-time Markov Chains: An Applications-Oriented Approach. Springer Science & Business Media.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., 2014. Drebin: effective and explainable detection of android malware in your pocket. In: NDSS.

Berchtold, A., 2002. High-order extensions of the double chain Markov model. Stoch. Models.

Burguera, I., Zurutuza, U., Nadjm-Tehrani, S., 2011. Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. ACM, pp. 15–26.

Canfora, G., Mercaldo, F., Visaggio, C.A., 2016. An hmm and structural entropy based detector for Android malware: An empirical study. Comput. Secur. 61, 1–18.

Chen, Y., Ghorbanzadeh, M., Ma, K., Clancy, C., McGwier, R., 2014. A hidden markov model detection of malicious android applications at runtime. In: Wireless and Optical Communication Conference (WOCC), 2014 23rd. IEEE, pp. 1–6.

Chen, S., Xue, M., Tang, Z., Xu, L., Zhu, H., 2016. Stormdroid: A streaminglized machine learning-based system for detecting android malware. In: Proceedings of the 11th ACM on.

Corder, G.W., Foreman, D.I., 2014. Nonparametric Statistics: A Step-by-Step Approach. John Wiley & Sons.

CSeiffert, C., Khoshgoftaar, T., Van Hulse, J., 2009. Hybrid sampling for imbalanced data. Integr. Comput.-Aided Eng. 16 (3), 193–210.

Dahl, G.E., Stokes, J.W., Deng, L., Yu, D., 2013. Large-scale malware classification using random projections and neural networks. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 3422–3426.

Dash, S.K., Suarez-Tangil, G., Khan, S., Tam, K., Ahmadi, M., Kinder, J., Cavallaro, L., 2016. Droidscribe: Classifying android malware based on runtime behavior. In: Security and Privacy Workshops (SPW), 2016 IEEE. IEEE, pp. 252–261.

Desnos, A., et al., 2011. Androguard, URL: https://github.com/androguard/androguard.

Duong, T.V., Bui, H.H., Phung, D.Q., Venkatesh, S., 2005. Activity recognition and abnormality detection with the switching hidden semi-markov model. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2005, Vol. 1. CVPR 2005, IEEE, pp. 838–845.

Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M.S., Conti, M., Rajarajan, M., 2015. Android security: a survey of issues, malware penetration, and defenses. IEEE Commun. Surv. Tutor. 17 (2), 998–1022.

Feizollah, A., Anuar, N., Salleh, R., Suarez-Tangil, G., 2017. Androdialysis: Analysis of android intent effectiveness in malware detection. Computers.

Fink, G., 2014. Markov models for pattern recognition: from theory to applications.

Gers, F.A., Schraudolph, N.N., Schmidhuber, J., 2002. Learning precise timing with lstm recurrent networks. J. Mach. Learn. Res. 3 (Aug), 115–143.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press.

Gorla, A., Tavecchia, I., Gross, F., Zeller, A., 2014. Checking app behavior against app descriptions. In: Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. ACM Press, New York, New York, USA, pp. 1025–1035.

Ham, H., Choi, M., Hyo-Sik Ham, H.-S., Mi-Jung Choi, M.-J., 2013. Analysis of Android Malware Detection Performance using Machine Learning Classifiers. IEEE, pp. 490–495.

He, H., Garcia, E.A., 2009. Learning from imbalanced data. IEEE Trans. Knowl. Data Eng. 21 (9), 1263–1284.

Hou, S., Saas, A., Chen, L., Ye, Y., 2016a. Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. In: IEEE/WIC/ACM International Conference on Web Intelligence Workshops. (WIW), IEEE, pp. 104–111.

Hou, S., Saas, A., Ye, Y., Chen, L., 2016b. DroidDelver: An android malware detection system using deep belief network based on api call blocks. In: International Conference on Web-Age Information Management. Springer, pp. 54–66.

Karatzas, I., Shreve, S., 2012. Brownian Motion and Stochastic Calculus, Vol. 113. Springer Science & Business Media.

Krizhevsky, A., Sutskever, I., Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (Eds.), Advances in Neural Information Processing Systems 25. Curran Associates, Inc., pp. 1097–1105.

Lipton, Z.C., Berkowitz, J., Elkan, C., 2015. A critical review of recurrent neural networks for sequence learning, arXiv preprint arXiv:1506.00019.

Mariconti, E., Onwuzurike, L., Andriotis, P., De Cristofaro, E., Ross, G., Stringhini, G., 2016. Mamadroid: Detecting android malware by building markov chains of behavioral models, arXiv preprint arXiv:1612.04433.

Martín, A., Menéndez, H.D., Camacho, D., 2016a. Mocdroid: multi-objective evolutionary classifier for Android malware detection. Soft Comput. 1–11.

Martín, A., Menéndez, H.D., Camacho, D., 2016b. String-based malware detection for android environments. In: International Symposium on Intelligent and Distributed Computing. Springer International Publishing, pp. 99–108.

Martín, A., Menéndez, H.D., Camacho, D., 2016c. Studying the influence of static api calls for hiding malware. In: Conference of the Spanish Association for Artificial Intelligence. Springer, pp. 363–372.

Massarelli, L., Aniello, L., Ciccotelli, C., Querzoni, L., Ucci, D., Baldoni, R., 2017. Android malware family classification based on resource consumption over time, CoRR abs/1709.00875.

Perdisci, R., Lanzi, A., Lee, W., 2008. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In: ACSAC. IEEE Computer Society, pp. 301–310.

Puterman, M.L., 2014. Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons.

Qiao, Y., He, J., Yang, Y., Ji, L., 2013. Analyzing malware by abstracting the frequent itemsets in API call sequences. In: 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications. (TrustCom), IEEE, pp. 265–270.

Rieck, K., Trinius, P., Willems, C., Holz, T., 2011. Automatic analysis of malware behavior using machine learning. J. Comput. Secur. 19 (4), 639–668.

Rodríguez-Fernández, V., Gonzalez-Pardo, A., Camacho, D., 2016a. A method for building predictive hsmms in interactive environments. In: 2016 IEEE Congress on Evolutionary Computation. (CEC), IEEE, IEEE, pp. 3146–3153.

Rodríguez-Fernández, V., Gonzalez-Pardo, A., Camacho, D., 2016b. Finding behavioral patterns of UAV operators using multichannel hidden markov models. In: 2016 IEEE Symposium Series on Computational Intelligence, {SSCI} 2016, Athens, Greece, December 6–9, 2016. IEEE, pp. 1–8.

Salfner, F., Malek, M., 2007. Using hidden semi-markov models for effective online failure prediction. In: 26th IEEE International Symposium on Reliable Distributed Systems, 2007. SRDS 2007, IEEE, pp. 161–174.

Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Nieves, J., Bringas, P.G., Álvarez Marañón, G., 2013. MAMA: Manifest analysis for malware detection in Android, Vol. 44, pp. 469–488.

Shabtai, A., Moskovitch, R., Elovici, Y., Glezer, C., 2009. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. Inf. Sec. Tech. Rep. 14 (1), 16–29.

Srivastava, N., Hinton, G.E., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 15 (1), 1929–1958.

Suarez-Tangil, G., Conti, M., Tapiador, J.E., Peris-Lopez, P., 2014. Detecting targeted smartphone malware with behavior-triggering stochastic models. In: European Symposium on Research in Computer Security. Springer, pp. 183–201.

Visser, I., 2011. Seven things to remember about hidden Markov models: A tutorial on Markovian models for time series. J. Math. Psych. 55 (6), 403–415.

Wagner, M., Fischer, F., Luh, R., Haberson, A., Rind, A., Keim, D.A., Aigner, W., Borgo, R., Ganovelli, F., Viola, I., 2015. A survey of visualization systems for malware analysis. In: EG Conference on Visualization (EuroVis)-STARs. pp. 105–125.

Xie, L., Zhang, X., Seifert, J.-P., Zhu, S., 2010. pbmds: a behavior-based malware detection system for cellphone devices. In: Proceedings of the Third ACM Conference on Wireless Network Security. ACM, pp. 37–48.

Xu, K., Li, Y., Deng, R.R.H., 2016. ICCDetector: ICC-Based Malware Detection on Android, Vol. 11, pp. 1252–1264.

Yan, L.-K., Yin, H., 2012. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In: USENIX Security Symposium. pp. 569–584.

Yuan, Z., Lu, Y., Xue, Y., 2016. Droiddetector: android malware characterization and detection using deep learning. Tsinghua Sci. Technol. 21 (1), 114–123.

Zhang, M., Duan, Y., Yin, H., Zhao, Z., 2014. Semantics-aware android malware classification using weighted contextual api dependency graphs. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14. ACM Press, New York, New York, USA, pp. 1105–1116.

Alejandro Martín is a teaching assistant in Universidad Autónoma de Madrid. He has a B.Sc. in Computer Science from Universidad Carlos III de Madrid, and a M.Sc. in Computer Science from the mentioned university. Nowadays, he is a Ph.D. candidate at Universidad Autónoma de Madrid (UAM). He is involved with Applied Intelligence and Data Analysis (AIDA) interest research group at EPS-UAM, his main research interests are related to Malware analysis, Classification, Evolutionary Computation and Machine Learning.

Víctor Rodríguez-Fernández is a researcher in Universidad Autónoma de Madrid. He has a B.Sc. in Computer Science and in Mathematics from Universidad Autonoma de Madrid, and a M.Sc. in Computer Science from the mentioned university. Nowadays, he is a Ph.D. candidate at Universidad Autónoma de Madrid (UAM). He is involved with Applied Intelligence and Data Analysis (AIDA) interest research group at EPS-UAM, his main research interests are related to Unsupervised learning, Hidden Markov Models, Pattern recognition, Data mining, Unmanned Aerial Vehicles.

David Camacho is currently working as Associate Professor in the Computer Science Department at Universidad Autonoma de Madrid (Spain) and Head of the Applied Intelligence & Data Analysis (AIDA) interest research group. He received a Ph.D. in Computer Science (2001) from Universidad Carlos III de Madrid, and a B.S. in Physics (1994) from Universidad Complutense de Madrid. He has published more than 200 journals, books, and conference papers. His research interests include Data Mining (Clustering), Evolutionary Computation (GA & GP), Multi-Agent Systems and Swarm Intelligence (Ant colonies), Automated Planning and Machine Learning, or Video games among others.

PUBLICATION 4

Picking on the family: Disrupting android malware triage by forcing misclassification

(IJ-4) Calleja, Alejandro; **Alejandro Martín**, Héctor D. Menéndez, Juan Tapiador & David Clark: "Picking on the family: Disrupting android malware triage by forcing misclassification." *Expert Systems with Applications*, 95 (2018): 113-126, DOI: `10.1016/j.eswa.2017.11.032`
Impact factor = 3.768 (JCR, 2017) [Q1, 20/132, Computer Science, Artificial Intelligence].

- **Contributions of the PhD candidate:**
  - Second author of the article.
  - Contributions made in the conception of the presented ideas.
  - Contributions made in the design and implementation of the genetic algorithm exposed.
  - Implementation of the countermeasure.
  - Co-author of the interpretation and discussion of results provided.
  - Co-author of the manuscript, figures and tables presented.

Expert Systems With Applications 95 (2018) 113–126

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Picking on the family: Disrupting android malware triage by forcing misclassification

Alejandro Calleja[a], Alejandro Martín[b], Héctor D. Menéndez[c,*], Juan Tapiador[a], David Clark[c]

[a] Department of Computer Science, Universidad Carlos III de Madrid, Madrid, Spain
[b] Departamento de Informática, Universidad Autónoma de Madrid, Madrid, Spain
[c] University College London (UCL), Gower Street, London WC1E 6BT, United Kingdom

ARTICLE INFO

Article history:
Received 31 March 2017
Revised 29 October 2017
Accepted 14 November 2017

Keywords:
Malware classification
Adversarial learning
Genetic algorithms
Iagodroid

ABSTRACT

Machine learning classification algorithms are widely applied to different malware analysis problems because of their proven abilities to learn from examples and perform relatively well with little human input. Use cases include the labelling of malicious samples according to families during triage of suspected malware. However, automated algorithms are vulnerable to attacks. An attacker could carefully manipulate the sample to force the algorithm to produce a particular output. In this paper we discuss one such attack on Android malware classifiers. We design and implement a prototype tool, called IagoDroid, that takes as input a malware sample and a target family, and modifies the sample to cause it to be classified as belonging to this family while preserving its original semantics. Our technique relies on a search process that generates variants of the original sample without modifying their semantics. We tested IagoDroid against RevealDroid, a recent, open source, Android malware classifier based on a variety of static features. IagoDroid successfully forces misclassification for 28 of the 29 representative malware families present in the DREBIN dataset. Remarkably, it does so by modifying just a single feature of the original malware. On average, it finds the first evasive sample in the first search iteration, and converges to a 100% evasive population within 4 iterations. Finally, we introduce RevealDroid*, a more robust classifier that implements several techniques proposed in other adversarial learning domains. Our experiments suggest that RevealDroid* can correctly detect up to 99% of the variants generated by IagoDroid.

1. Introduction

Detecting and classifying malware is a challenge that has steadily increased over time. Not only has the rate of production of distinct files been increasing but the methods used to evade detection have become more sophisticated. For instance, malicious apps have been observed colluding to achieve their desired outcomes (Labs, 2016; Zhou & Jiang, 2012). The quantity of malware targeting mobile devices doubled in the year to July 2016 (Labs, 2016), with a clear trend towards the reuse of source code instead of developing new variants from scratch (Zhou & Jiang, 2012). Mobile malware variants are produced through component reuse and also via obfuscation. Considering the advances in machine learning techniques in the last decades, there is widespread interest in applying these to the malware triage problem. Contemporary machine learning algorithms provide the potential to improve scalability and offer high flexibility regarding the features employed during the classification of malware into families (Dash et al., 2016; Gandotra, Bansal, & Sofat, 2014). However, an informed adversary can deliberately alter the decision process of an automated classifier by different means. The problem of employing machine learning algorithms in adversarial environments has previously been studied in security related contexts such as spam, intrusion detection, or malware classification (Biggio, Rieck et al., 2014; Dalvi, Domingos, Sanghai, & Verma, 2004; Lowd & Meek, 2005). In the same way, different countermeasures have been proposed (Biggio, Corona, Fumera, Giacinto, & Roli, 2011; Chinavle, Kolari, Oates, & Finin, 2009).

This paper investigates the automated disruption of Android malware triage, the process by which decisions are made in regard to the further analysis steps for a suspicious file (Chakradeo, Reaves, Traynor, & Enck, 2013). A critical step during this process, that may affect the choice of subsequent analysis techniques, is the identification of the malware family of a highly suspicious file. Our attack is that a malware writer, in deploying

* Corresponding author.
E-mail addresses: accortin@inf.uc3m.es (A. Calleja), alejandro.martin@uam.es (A. Martín), h.menendez@ucl.ac.uk (H.D. Menéndez), jestevez@inf.uc3m.es (J. Tapiador), david.clark@ucl.ac.uk (D. Clark).

https://doi.org/10.1016/j.eswa.2017.11.032

variants from a relatively novel family, attempts to disguise them as a different family, one that is less likely to attract intensive scrutiny. This may hide novel indicators of compromise such as DNS records, malicious URLs, or exploits (Lakhotia, Walenstein, Miles, & Singh, 2013).

In this scenario, the power of the malware writer or adversary is as follows: she has control over her malware sample and is able to extract static features such as intents-actions, API calls, and information flows. In addition, she knows the feature space used by the targeted classifier and has access to the classification/misclassification probability. This is a relatively strong assumption, yet the attacker still has the limitation of not knowing the underlying classification algorithm and she needs to preserve the semantics of the executable. Besides, she wants to automate the process. Our solution to this problem, a tool called IagoDroid, uses evolutionary algorithms to perform a search that identifies a minimal number of changes to the features in order to effect a family misclassification. IagoDroid can randomly choose a family or target a specific family.

Assuming further knowledge about the classifier is unrealistic in practice. Since the mapping (from vectors to labels) implemented by the classifier is unknown, there is no other option but to treat it as a black box that can be repeatedly queried during search. Even when this is not the case and the details about the classifier are fully known, obtaining an actionable analytical description of such a mapping might not be always possible, particularly for non-linear classifiers that capture complex interactions among features to produce the output label. Population-based search mechanisms such as genetic algorithms have proven to perform remarkably well in challenging domains where more traditional search algorithms have not succeeded (Sivanandam & Deepa, 2007).

Attacks against classifiers have been discussed before, both from a theoretical point of view and in particular security domains such as spam or intrusion detection. In this paper we study the impact of an attack against multiclass Android malware classifiers. Android apps are extremely easy to decompile, manipulate and repackage again into a new app. This makes it easy to introduce new artefacts (e.g., components, API calls, intents, information flows) in the app that will affect its associated feature vector and, therefore, the label given by a classifier. If carefully introduced (for instance, in if-then blocks only accessible through an opaque predicate that always evaluates to false), such modifications will not affect the app's execution semantics.

To demonstrate our approach, IagoDroid attacks family classification by RevealDroid (Garcia, Hammad, Pedrood, Bagheri-Khaligh, & Malek, 2015), a recently proposed malware classifier employing existing static analysis features. Our choice of RevealDroid is for convenience (it is open source and ready to use) and because it incorporates most of the static features discussed in the literature (API calls, information flows, and so on). However, IagoDroid is agnostic with respect to the classifier used and can be applied to different classifiers. Moreover, we have subsequently designed a countermeasure that can detect when a potential evasion has been performed and can recover a set of potential original families.

The main contributions of this paper are summarized as follows:

- We propose a novel classification evasion attack against any triage process where the family classification relies on static analysis. We demonstrate, in particular, that IagoDroid can evade an open source classifier named RevealDroid, a freely available multi-class malware classifier which combines several different features. To do so, we employ evolutionary algorithms, a technique which has been previously employed in the context of evading classifiers for security applications (Pastrana, Orfila, & Ribagorda, 2011; Xu, Qi, & Evans, 2016) (see Section 2).
- We train RevealDroid using 1919 malware samples from the DREBIN (Arp, Spreitzenbarth, Hubner, Gascon, & Rieck, 2014) dataset divided into 29 different malware families. IagoDroid successfully forces misclassification of 28 of the 29 families, in the process modifying only a single feature of the original malware feature vector. On average, IagoDroid is able to find the first evasive file within the first generation and converges on a 100% evasive population within 4 generations (see Section 4). It was able to find approximately 14,000 evasive variants from more than 290 initial malware samples within 2 min.
- The countermeasure, named by us as RevealDroid*, detects potential evasions in between 90% and 99% of the output of IagoDroid, depending on the number of modifications introduced, and can identify potential original families for the malware (see Section 5).

The rest of this paper is organized as follows: In Section 2, we present our approach, introducing issues related to our contribution such as the adopted adversarial model, the target classifier, and the parameters of the genetic algorithm component. Section 3 describes the experiments and our configuration of them. In Section 4 we analyse and discuss the results while Section 5 describes the countermeasure proposed. Section 6 introduces the most relevant, related contributions found in the literature and finally Section 7 concludes the paper.

2. IagoDroid

This section describes IagoDroid, a prototype tool that induces mislabelling of malware families during the triaging process for potential malware samples. Given the importance of automated systems to detect and classify malware, to understand how these systems can fail (and how can they be strengthened) when attacks are directed against their integrity is an important task. IagoDroid's main goal is to demonstrate that an attack on an Android malware classification tool is feasible, by forcing it to produce a family misclassification as the result of some minor changes in the original sample and without modifying its semantics.

Following the taxonomy of attacks on machine learning developed by Barreno, Nelson, Sears, Joseph, and Tygar (2006), our approach can be positioned as follows:

- **Exploratory Attacks:** The attack described in this paper is *exploratory* since it does not aim at altering the training process but the classification itself, offline.
- **Targeted Attacks:** Regarding specificity, the proposed attack is focused on misleading the label given by the classifier to a particular sample. Nevertheless, the use of evolutionary search to find a proper mutation strategy can be used to fool the detection of any sample in the dataset as demonstrated in the following sections of the paper.
- **Integrity Attacks:** In contrast to attacks against the availability of the classifier, we do not seek to induce random classification errors. We aim to coerce an intended family misclassification for specific input samples.

The basic idea behind the IagoDroid attack is that the feature vector of a malicious application can be transformed by injecting new specific, incremental values, and this can eventually result in the assignment of an incorrect family label. These changes in the feature vector require modifications in the app's code and resources, in order to build a new sample corresponding to the desired feature vector. For instance, it may be necessary to include a new API call. Moreover, these changes are made while simultaneously keeping the semantics of the app invariant. The

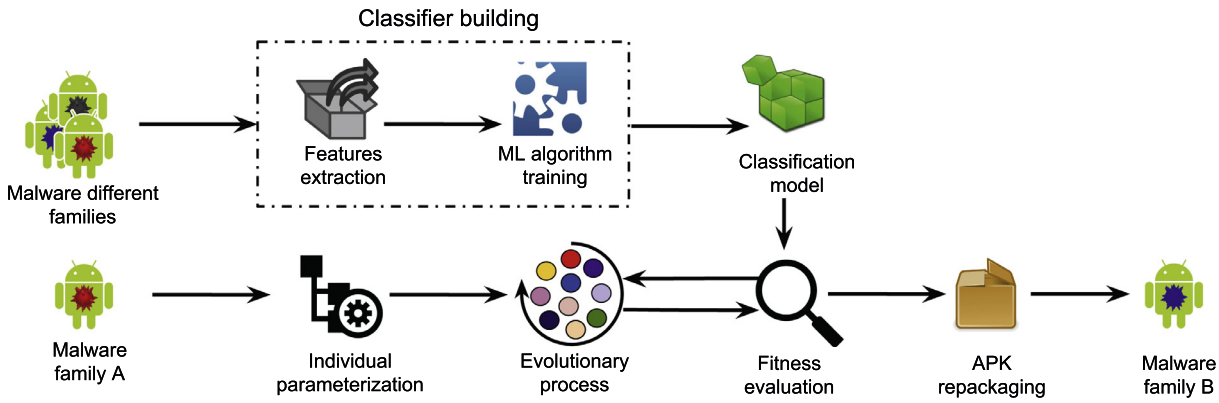


Fig. 1. General scheme of IagoDroid.

key to achieving semantic invariance is to only consider small, *incremental* changes in the app (i.e., adding new API calls or new permissions) each of which does not alter the original semantics.

Obtaining the list of transformations to apply to the feature vector can be seen as a search problem in which a search heuristic finds a solution (i.e., a new feature vector) based on the proximity of the current feature vector to one associated with a label different from the current one. This proximity can be calculated based on the output of a malware classifier, by measuring the probability of the feature vector being classified as the original label or as a different label.

Fig. 1 shows the general architecture of IagoDroid. There are two main pipelines, one depicted above the other. The upper pipeline shows the process of building the classification algorithm used to drive the heuristic search. This algorithm takes as input a set of samples placed in a feature space. These samples are employed to train the classifier and obtain a classification model. On the other hand, the process of performing the attack is shown in the bottom pipeline. In this case the process starts by picking a malware sample whose family label we wish to alter. Additionally, the attack pipeline can also take a target family (see Section 2.1). IagoDroid employs a genetic algorithm to perform the heuristic search, as these algorithms to adapt to problems of high complexity. The search is guided by a fitness function which uses the classification model previously trained to find the solutions that induce misclassification. Finally, the application is modified in order to adapt it to its new feature vector and it is repackaged to obtain a new app which is able to evade a correct family classification.

The following subsections present the context for IagoDroid, starting from a description of the adversarial model, a specification of the target classifier and finishing with the problem formalisation.

2.1. Adversarial model

In our scenario, we consider an adversary who aims to evade the correct classification of a sample belonging to family *A* by misclassifying it as family *B*.

The goal of the adversary is to ensure that it is possible to miss the identification of the correct family. We consider two cases. In the first scenario, the selection of the target family is delegated to the evolutionary algorithm which will merely try to change the label of the input feature vector with the minimum number of changes. In the second scenario, the target family is also an input and the search will attempt to find the feature changes that attain this specific misclassification.

As introduced in Section 1, the adversary seeks to thwart the deployment of proper countermeasures. To appreciate how the attacker achieves this, it is useful to specify what the adversary knows about the classifier. Given that IagoDroid is based on a well known classification algorithm whose source code is publicly available, we allow the feature set employed by the classifier to be known to the attacker. We assume the attacker is able to create new feature vectors and submit them directly to the classifier without any constraint. We assume that the classifier interacts with the submitted feature vectors as if they had been extracted from applications created or modified by the adversary. In other words, the search is conducted at the feature vector level, without directly modifying the malware sample until a solution is found.

Regarding the classifier output, the attacker receives two values: the label assigned to the input vector and a classification score, indicating the trust/reliability of the classification. Since the adversary is able to deploy her own implementation of the classifier, we do not consider any limitation in the number of feature vectors that can be submitted, hence the attacker has an unbounded number of attempts to lead the classifier to a compromised verdict.

This scenario for the adversarial capabilities is realistic since the target classifier can be well documented (i.e., no security through obscurity) or else reversed.

2.2. Target classifier

We decided to use an already proposed and documented classifier in our work. Our selection criteria for choosing a target classifier included good classifier precision and high diversity in the features it uses. While there are several classifiers discussed in the literature, few of them consider an important and representative set of features and are freely available to download. Table 1 compares the use of different features by the most important classifiers described in the literature and notes whether they can be downloaded to be used for our purposes. From the nine analysed proposals, only the authors of three of them have released the source code of their solutions, RevealDroid, Dendroid and DroidLegacy. Of these, RevealDroid is the most appropriate one since it uses the widest set of features. In addition, it was designed and tested for malware family classification.

RevealDroid classifier building

RevealDroid consists of a series of components that enable the extraction of three different kinds of data from Android apps: API calls, intent actions, and streams and flows. These features can be used to build a dataset and then to train a machine learning algorithm to perform a classification task that predicts the family label of previously unseen samples. Each group of features is extracted separately and is sequentially added to the feature vector for each application, with the objective of controlling and

Table 1
Android malware classification methods using machine learning approaches.

Classifier	Code structures	Permissions	Api Calls	Intent-actions	Flow analysis	Tested for families classification	Freely available to download
RevealDroid (Garcia et al., 2015)	✗	✗	√	√	√	√	√
DroidSIFT (Zhang et al., 2014)	✗	√	√	√	√	✗	✗
Dendroid (Suarez-Tangil et al., 2014)	√	✗	✗	✗	✗	√	√
Drebin (Arp et al., 2014)	✗	√	√	√	✗	√	✗
DroidMiner (Yang et al., 2014)	✗	✗	√	√	✗	√	✗
DroidAPIMiner (Aafer et al., 2013)	✗	✗	√	✗	✗	✗	✗
VILO (Lakhotia et al., 2013)	√	✗	✗	✗	✗	√	✗
DroidLegacy (Deshotels et al., 2014)	✗	✗	√	✗	✗	√	√
MAST (Chakradeo et al., 2013)	√	√	✗	√	✗	✗	✗

supervising the whole process. We used the original code of RevealDroid, downloaded from its public repository.[1]

The first feature extracted from each application is a list of the API calls found in the code, which allows one to obtain a high level description of the expected behaviour of the application. These API calls can be included in the feature vector of an app in two ways: grouping the calls by using the 30 security-sensitive API categories defined by Rasthofer, Arzt, and Bodden (2014), or grouping the calls by using the Android package in which they are defined. Revealdroid follows this second approach.

The second step of the dataset building process consists of including intent actions data. Intent actions are identifiers of different events that happen within the lifecycle of an application such as launching a new activity or a new service. This is also a useful information source for detecting and classifying malicious applications (Chin, Felt, Greenwood, & Wagner, 2011).

Thirdly, RevealDroid uses information flows to characterise the samples. An information flow can be seen as the path followed by a piece of sensitive data through the flow graph of a program. In this case, an information flow is represented as a pair consisting of a *source* (i.e. an API call providing data to the app) and a *sink* (i.e. the app providing data as input for another API call).

The final step involves the training process of a machine learning classification algorithm. The authors of RevealDroid use a decision tree based algorithm, C4.5, and the 1-nearest neighbour algorithm. Nevertheless, any other machine learning algorithm might be used instead.

2.3. Problem formalisation

In this subsection we provide a formal description of the attack.

Our experimental dataset can be formalised as the set X, containing samples of different malware families. However, since we are solely interested in the feature vectors describing different properties of each sample, X can be represented as the set of n feature vectors:

$$X = \{x_1, x_2, \ldots, x_n\}. \tag{1}$$

Each feature vector x_i is composed of k different features, extracted directly from the original application:

$$x_i = \{x_i^1, x_i^2, \ldots, x_i^k\}. \tag{2}$$

Initially, each sample in the dataset is labelled with the name of the family it belongs to. We name the set of all the possible labels in the dataset as Y. Thus, the classifier C can be defined as a function mapping a feature vector $x_i \in X$ to the most likely label $y_j \in Y$, paired with its probability of being the correct label:

$$C(x_i) = (p(y_j), y_j),\ \ y_j \in Y, \tag{3}$$

where $p(y_j)$ is the probability of y_j being the true label of x_i as estimated by the classifier.

[1] https://bitbucket.org/joshuaga/revealdroid.

Finally, we formalise our search approach at a high level of abstraction as a function accepting two arguments: a feature vector which is to be misclassified, obtained from the app, and the original label that we want to avoid. The output of this function (x_i') will be the original vector with a set of changes (e.g., increment the value of a feature) to be applied to the original feature vector x_i. Once this new vector has been created, the classifier C will assign a new label y_j' to this modified vector:

$$\text{IagoDroid}(x_i, y_j) = x_i' :\ \ y_j' \in Y,\ \ y_j' \neq y_j. \tag{4}$$

We consider the changes as a Δ vector satisfying: $x_i + \Delta = x_i'$.

2.4. Genetic approach

This section describes the design details of the genetic algorithm that is at the core of IagoDroid.

2.4.1. Encoding

Each individual I_i present in the evolutionary process is designed to represent a possible new feature vector x_i' containing k different features or genes. Since the goal of IagoDroid is to introduce modifications in the feature vector so that the associated app gets misclassified while preserving its semantics, the individual's encoding is designed to only allow incremental changes in each feature. Thus, the individual starts with the same feature vector as the sample received as input x_i. Once the minimum value of each gene I_i^j of the individuals is established, it is also necessary to fix a maximum threshold MT to limit the number of changes and facilitate their implementation. Then, $[x_i, x_i + MT]$ is the range for each gene in each individual I_i. This restriction on the values of each individual will be present through the entire evolutionary process.

2.4.2. Genetic operators

Four operators are in charge of driving the evolutionary process across a number of generations. The **selection** operator is elitist, picking the n best individuals in each generation to be part of the next generation. **Reproduction** is performed by means of a standard tournament operator. For **crossover** we opt for a uniform operator and, lastly, a random **mutation** operator is used to introduce diversity in the population by changing the value of some genes randomly (within the ranges specified above).

2.4.3. Fitness function

The fitness function uses the gradient of the classifier output (score) to guide the genetic search. Specifically it uses the probability of the class that the algorithm wants to avoid. This can be formally defined as:

$$f(x_i, y_j) = \begin{cases} 1 - p(y_j) & \text{if } (p(y_j), y_j) = C(x_i) \\ 1 & otherwise \end{cases} \tag{5}$$

where x_i is the feature vector of the application, y_j indicates its family and p represents the probability assigned to the classification.

2.5. Targeting specific families

The approach described so far addresses a genetic search seeking to reach different malware families, providing an effective technique to hide the real family of a malicious application. However, the search has no control over the final family label that will be assigned to the modified sample. This represents an interesting issue, since an attacker might well wish to target specific families with different purposes (for instance, to force defenders to deploy specific incorrect countermeasures). To address this issue, the fitness function can be easily modified to guide the search to individuals representing feature vectors classified as a given target family. The new fitness function is as follows:

$$f(x_i, y_k) = \begin{cases} p(y_k) & \text{if } C(x_i) \neq (p(y_k), y_k) \\ 1 & otherwise \end{cases} \quad (6)$$

where y_k represents the target family label.

3. Experimentation

We next discuss the experiments that we have performed to validate our proposal. The experiments address the following research questions:

- **RQ1:** *How much effort does it take to find the modifications needed to misclassify a particular sample?*
- **RQ2:** *Which features are more often involved when modifying a sample?*
- **RQ3:** *Given a malware family, is the cost of forcing misclassification errors in its samples constant for all possible target families or are some families easier to target than others?*

Our main goal is to provide evidence that our approach can induce a misclassification error in a targeted malware classifier. Accordingly, the experiments discussed in this section have been executed using only the first fitness function presented in Section 2.4.3 and we did not direct the genetic search towards a particular family classification. The rest of this section describes the experimental setting, including the dataset, classifiers and parameters used. To facilitate the reproducibility of our experiments, we have created open source versions of our implementation, dataset and scripts used throughout this work[2].

3.1. Dataset

We tested our approach using the DREBIN dataset (Arp et al., 2014). This dataset contains 5560 malicious Android apps classified into 179 different families. Unfortunately, the number of samples per family is not balanced, resulting in some families with a low number of samples (e.g., 47 families contain just 1 sample). We therefore removed all classes containing less than 10 samples, resulting in a final dataset composed of 5198 samples distributed in 54 different families.

We then leveraged a number of existing tools to extract the features from each sample in the dataset. API calls and intent actions were obtained using Androguard[3], a fairly well known static analysis tool. To extract information flows we used FlowDroid (Arzt et al., 2014), a taint analysis tool that finds *source-sink* connections. FlowDroid can be tuned through different parameters to maximise either performance or precision. We set parameters to achieve as much precision as possible. This approach differs slightly from the procedure followed by other works that have generally aimed at maximising performance by compromising precision (e.g., Garcia et al., 2015). Using FlowDroid to extract information flows introduces two important issues. First, the time it takes to analyse a single app ranges from a few minutes to several hours in the worst case. Furthermore, it unexpectedly crashes for many apps. These two issues (scalability and stability) forced us to dramatically reduce the number of samples actually used in the experiments. Thus, from the original set of 5189 samples, only 1919 samples belonging to 29 different families were successfully processed by FlowDroid.

Finally, once the final dataset was built, we carried out a basic covariance analysis among the features to remove those that did not provide any additional information.

3.2. Target classifier

To demonstrate our approach, we relied on RevealDroid, an Android malware classifier that uses various static features and allows the use of different machine learning classification algorithms. While the original authors used C4.5 and 1-NN, we restricted ourselves to C4.5 since it showed better accuracy and precision. Nevertheless, our approach is not limited to a particular classification algorithm and should work with any other classification approach. The C4.5 algorithm was trained using the RWeka package for R, keeping its default parameters. We use 2/3 of the data for training combined with 10 cross-fold validation and the remaining 1/3 for testing. The testing accuracy is 88% averaged over 50 runs.

3.3. Genetic search

The genetic algorithm was configured using the following parameters: a mutation probability of 0.1; a crossover probability of 0.8; population size equal to 50; maximum number of generations equal to 20; elitism parameter of 3; and a maximum number of transformations per allele of 1 (though we set an increment that provides a transformation probability per allele ranging from 0.6 to 1).

3.4. Attack steps

This subsection discusses the sequence of steps followed by an attacker to force the missclassification of a particular sample. Recall (Section 2.1) that we assume an adversary with full knowledge and unlimited access to the classifier.

The first step is to extract the features from the malicious samples that will be eventually mutated. Androguard and FlowDroid extract these features and generate the feature vector x_i. This feature vector provides a basis for the genetic search. Since the aim of the attacker is to change the final label of the sample without altering its functionality, the way in which the components of this vector may be modified during the search is restricted. For instance, if a particular API call is used in the original sample, the mutated sample must keep this feature (i.e., if the component of this API call is set to 1 in the original vector it cannot be set to 0). Otherwise the semantics of the application will be altered and the malicious behaviour will not be preserved. The genetic algorithm takes this into account and only mutates these features by adding additional intent actions, API calls or information flows, *without removing any of the original values*. Under this premise, the search process generates new individuals by evolving the previous generation. On each iteration, the fitness function evaluates for every single individual whether the correct classification has been evaded.

Once a solution is found, the attacker applies the mutation strategy found by the genetic search to the original malware sample. This will require adding a combination of new intent actions,

[2] The dataset is available at https://data.mendeley.com/datasets/4sksrpm5vj/1 and the code at https://github.com/hdg7/lagoDroid.

[3] https://github.com/androguard/androguard

API calls, and/or new information flows. To alter the original APK file, the attacker first decompresses it to access the files packed inside, such as the manifest or the DEX file(s). Adding a new intent action, API call or information flow requires disassembling the original DEX file, which contains the bytecode responsible for the app's functionality and is generated at compilation time from the original Java source code. There are several tools to carry out this process. Smali and Backsmali[4] are well known tools for translating the Dalvik bytecode contained the DEX file into human readable (smali) code. The result of disassembling a DEX file using these tools is a set of files related to the original Java sources. These files can easily be modified by the attacker to add a new call to an Android API method or a new intent action. To avoid introducing undesirable extra functionality into the app, the attacker can put the newly added code blocks within conditional sentences (i.e., if-then) driven by opaque predicates that always evaluate to false. This would prevent optimizers from removing them while achieving the two-fold goals of having those features in the code but not executing them. Once the new elements have been added to the code, the process can be reversed using Backsmali to repackage the APK file.

Unlike API calls or intent actions, information flows are related to the execution paths of the program. This means that a particular information flow will only be detected if it happens as part of the instructions that are actually executed when the app runs. This is a consequence of the way in which taint analysis tools based on symbolic execution, such as FlowDroid, explore the application to find possible data flows, building the application flow graph and following all the possible paths within the application. To insert a new information flow the attacker needs to place it within a method that will be eventually called. Android apps implement several callbacks (such as those used for managing the life-cycle of activities and services) to interact with different events taking place in the operating system. Thus, finding pieces of code that will certainly be executed is not difficult. To add a new information flow, the attacker can follow the procedure described above for intent actions and API calls.

We have manually tested the attacks with one of the samples in our dataset. Specifically, we modified an app labelled as a member of the Plankton family and, after altering it according to the found mutation strategy, the classifier misclassified it as a member of the BaseBridge family. Achieving misclassification only required the addition of a single intent action (ACTION INPUT METHOD CHANGED). After following the previously described steps, we examined the app and extracted the new feature vector. This new vector contains the feature ACTION INPUT METHOD CHANGED along with the original features of the app, showing that the modification step worked as expected while keeping the original features unchanged. Finally, we ran the classifier over this sample and obtained the wrong label (BaseBridge) as expected.

4. Results

We next discuss our experimental results. The experiments aim to provide answers to the three research questions introduced in the previous section. All the experiments were executed on a cluster of 6 nodes, each node equipped with 24 cores and 128Gb of RAM memory.

We took a random subset, selected uniformly across families, of samples from our main dataset for the experiments. This subset was composed of 290 samples, taking 10 samples per family from 29 different families. As we mentioned above, there are families that only have 10 samples, hence the need to pick at most 10 apps per family to balance the final sample.

4.1. Evasion effort

The first research question aims to measure the effort required by the attacker to find a mutation strategy that induces a classification error. We attempt to answer this question from three different points of view: (i) the number of generations required by the search to achieve evasion (i.e., to find a single individual evading the correct classification) and convergence (a whole generation evading correct classification); (ii) the number of modifications in the feature vector required; and (iii) the number of queries to the classifier (this is correlated with the first perspective but it is a standard metric in evasion environments (Biggio et al., 2013)).

Table 3 summarizes the results for the experiments carried out. It shows how many generations were enough to achieve evasion and at which point the genetic search converges (i.e., all individuals being misclassified). Remarkably, a solution is found in the first generation for all families but BaseBridge. This means that a single iteration of the genetic algorithm is required to evade the correct classification of a single sample. This achievement suggests that the search effort is low and the search might be replaced by a simple analytical process consisting on adding changes to the features (i.e., adding API calls, or intents among others). As a sanity check, we analysed this possibility considering transformations from a single sample of a specific family to another (in this case, from GinMaster to DroidKungFu). The analytical process can only add changes. However, all possible transformations from the vectors of GinMaster to vectors of DroidKungFu require subtractions. This would change the app semantics. Considering only those features that can be added, the analytical process requires between 500 and 16,000 changes from the original to the target vector. Using the same samples, the GA found solutions with only one change.

The average number of generations required to achieve convergence is around 4 for all families. Notable deviations include DroidKungFu, whose samples require around 7 generations, and SMSreg with less than 2 generations. This demonstrates that the evasion technique is extremely efficient against the classifier for the families tested. The only family whose samples cannot be successfully mutated so as to be classified as some other family is BaseBridge. A careful analysis of the results and the classifier's inner working for this family shows that samples with this label have a strong correlation with the ACTION_INPUT_METHOD_CHANGE feature. Every time this feature is present, the sample is classified as belonging to BaseBridge. Since the semantics preserving rules prevent us from removing any features, this poses a clear limitation on the attack.

The total amount of time taken for these experiments using the sample subset of 290 individuals is around 2 min. Within this time span, the search found 14,000 mutation strategies able to evade the classifier. This number can be broken down into 50 different mutation strategies for 280 individuals (omitting the ten individuals that belong to the BaseBridge family). This gives us interesting information about the performance of the attack and demonstrates how easy it is to evade a malware classifier such as RevealDroid.

To discover the minimum number of modifications needed to achieve misclassification, we set the change probability to the minimum value (0.6). The results are shown in Table 3. In this case, we selected malware samples uniformly from the whole dataset, considering a realistic scenario in which an attacker would employ different malware samples without any previous knowledge about their classifications. In this scenario, some samples were then misclassified by RevealDroid, showing that no modification is needed to evade it.

In almost all cases the average number of modifications is close to 1. This means that the evasion technique only needs to modify

[4] https://github.com/JesusFreke/smali

a single feature in order to evade classification, revealing how fragile malware classifiers such as RevealDroid are. In order to explore this case in more depth, Section 4.2 studies the impact of the different features during the misclassification attack.

Finally, the number of queries to the classifier during the evolution depends on the number of generations and the population size. In each generation a single query is executed for each sample in the population. Since our approach only needs one generation to succeed, 50 queries are needed per sample. This number is higher if we require the convergence of the whole population, as needs up to 350 queries (7×50) in the worst case.

RQ1. Our results show that misclassification can be achieved in just one generation of the genetic search. This translates to a number of queries to the classifier ranging from 50 to 350 per sample. Furthermore, only one mutation is needed to induce a misclassification error for most samples.

4.2. Relevant features for the attack

In order to understand which features are more related to a particular family, we performed an analysis of the most relevant features affected during the mutation process. We followed the same approach as in the experiments discussed in Section 4.1, where only one feature is needed to change the family classification. Table 3 shows the feature that is changed most often and the overall probability for this feature to be modified during the search.

The feature most frequently added is ACTION_INPUT_METHOD_CHANGED. As we mentioned above, this feature is tightly coupled with the BaseBridge family, in such a way that whenever it is added to the feature vector, the sample is classified as belonging to BaseBridge.

ACTION_USER_PRESENT is another feature that is present in the modifications, especially for families such as Kmin, Steek, Yzhc and Fatakr. These families are closely related to remote server connections (Kmin and Yzhc) and sending SMS messages (Steek and Fatakr) containing private information, so they do not necessarily focus on user actions.

RQ2. The feature ACTION_INPUT_METHOD_CHANGED is used most often due to its close relationship with the BaseBridge family. ACTION_USER_PRESENT is used next often, appearing in four families with a common behaviour (leaking information from the device).

4.3. Transition between families during evasion

The final experiment attempts to measure the difficulty of mutating samples from each family to each potential target family. To do this, we measured the most commonly changing patterns among the different families during the search. Fig. 2 depicts a probabilistic representation of the most frequent changes between families. Unsurprisingly, BaseBridge is the family to which samples are most commonly reclassified. This is related to our previous analysis in Section 4.2, which showed that any sample with the feature ACTION_INPUT_METHOD_CHANGED set to one is classified as belonging to BaseBridge regardless of any other features.

Some interesting relationships can be found, such as the one between Plankton and Nyleaker, which share almost the same intent actions, Plankton having a couple of actions more than Nyleaker. Kmin and GinMaster have a close relationship with

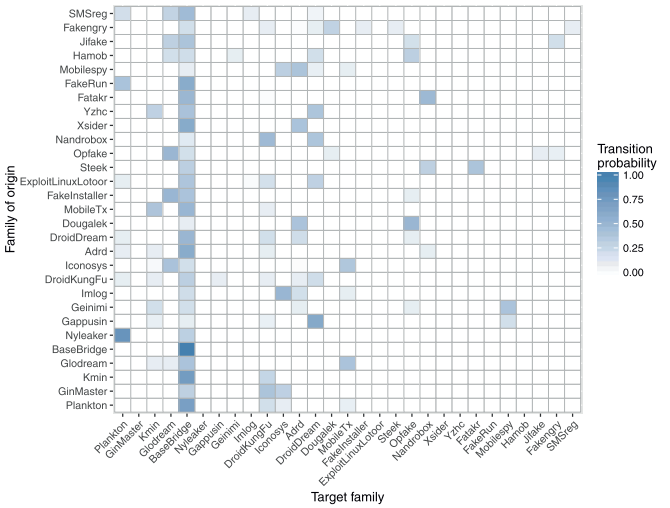


Fig. 2. Most frequent classification errors between families induced during the search.

DroidKungFu: a single modification of a flow based on MMS (Kmin) or the ACTION_USER_PRESENT feature (GinMaster) causes the original sample to be classified as belonging to DroidKungFu. A similar case happens with Fatakr and Nandrobox, in which a single modification of the ACTION_USER_PRESENT feature causes a misclassification.

Interestingly, the matrix shown in Fig. 2 is asymmetric. This means that samples from family A can be mutated into samples of family B but the inverse process was not found possible during the search. The only cases in which both mutations are possible are Plankton and DroidKungFu, DroidKungFu and Kmin, and Adrd and DroidKungFu. This suggests that DroidKungFu is a heterogeneous family. Finally, we note that there are 9 families that can never be targets: GinMaster, Nyleaker, Geinimi, Imlog, ExploitLinuxLotoor, Xsider, Yzhc, FakeRun and Hamob. This is a consequence of how the classifier builds the classification model, keeping some families bounded to specific feature ranges that are modified during the evolution process.

RQ3. The effort required to mutate a sample from an original classification to classification as a target family depends on both families, with some mutations being impossible.

5. A countermeasure

The results discussed in the previous section demonstrate that it is generally possible (and in fact easy) to cause a misclassification error in a typical Android malware classifier. This is ultimately accomplished by injecting additional artefacts into the sample, such as new API calls or intents, that will affect the feature vector associated with the app.

We next discuss how such attacks can be countered through the use of a more robust classifier. Our proposal first aims at detecting potential attack cases (i.e., samples deliberately modified so as to induce a classification error) and then at backtracking the changes to identify potential source families. Both strategies constitute variations of ideas proposed before in the field of adversarial machine learning (Chinavle et al., 2009). However, this is the first countermeasure discussing the ability to backtrack the attack.

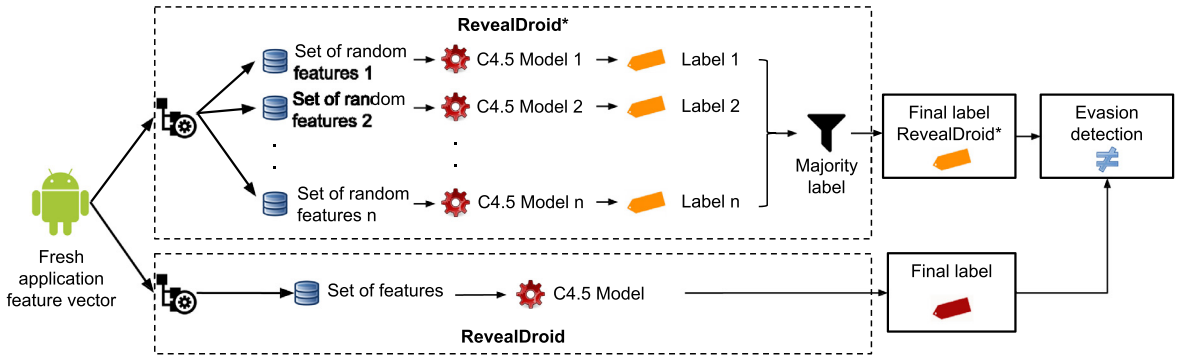


Fig. 3. Countermeasure schema including RevealDroid*.

5.1. Detecting potential misclassifications

The result of an attack, such as the one shown in this paper, is an app modified in a way that will deceive a classification system, causing it to return an incorrect family label. The underlying causes for such an error are related to the manner in which the algorithm at the core of the classifier works. In the case of RevealDroid, each alteration introduced in the application translates into features that will change the path followed along the decision tree, thus driving the output to a different leaf and, therefore, a different label.

In order to detect potential attempts to evade the classifier, we propose an extension of the target classifier: Revealdroid*. This enhanced version of RevealDroid employs a pool of C4.5 trees instead of relying on just one instance. Each classifier in the pool makes decisions based on different subsets of features present in the feature vector, making it more robust against deliberate modifications. Thus, each classifier chooses its own subset of features randomly at runtime. Therefore, a potential attacker has no evident way of modifying the vector in order to evade all the classifiers at once. The final label assigned to a sample by this enhanced version of RevealDroid results from majority voting. Our countermeasure is inspired by those proposed by different authors in the literature. The *bagging* (boosting and aggregating) approach has proven to be effective in enhancing the robustness of classifiers in various related problems (Biggio et al., 2011; Perdisci, Gu, & Lee, 2006).

Fig. 3 shows the architecture proposed for RevealDroid* and the whole schema for the countermeasure. The countermeasure consists of measuring the level of agreement between RevealDroid and RevealDroid*. When these two tools disagree, we consider that an attacker achieved a potential evasion. RevealDroid* must keep the same classifier, training data and parameters as RevealDroid, in order to generate similar outputs and reduce the false alarm (or false positive) rate. However, for the triage process, the priority is to reduce false negatives in order to guarantee that an important sample is not misclassified as irrelevant.

The feature extraction process of RevealDroid* remains unchanged, using the same feature vector for each app with a list of API calls, intent actions and information flows. Once the feature vector is generated for each app, features are randomly partitioned into a number of groups. That is, each feature is randomly assigned to one (and just one) group, guaranteeing that all groups have the same number of features. The number of groups can be manually tuned and also equals the number of classifiers (C4.5 in our case) used in the ensemble. Each classifier is then trained with all the instances using the subset of features allocated for it, seeking to maximise the separation among labels in this reduced feature space.

The classification process for a new malware sample with RevealDroid* is also outlined in Fig. 3. Once again, the feature vector is generated following the rules of RevealDroid. In a second step, the list of features is divided into groups depending on the split previously performed when training the models. Each instance of the C4.5 algorithm delivers a label according to its portion of the feature space and a majority rule is applied to obtain the final label for the input sample.

The strength of RevealDroid* lies in reducing the fragility of a single-classifier structure such as that of RevealDroid, in which just a simple change in the feature vector may lead to a classification error. When using multiple classifiers, the effort required to achieve a successful evasion becomes considerably more complex since the attacker needs to evade the majority of the classifiers in the ensemble. As a sanity check on RevealDroid*'s classification ability, we calculated its accuracy (see the plot at the bottom of Fig. 6). The accuracy (88%) is similar to that of RevealDroid (75–91%).

To evaluate the ability of the countermeasure to detect when a sample has been altered so as to evade a correct classification, we have used our attack to generate a representative set of apps successfully mutated, departing from, and trying to reach, all possible families following the approach described in Section 2.5. With this procedure, a subset of more than 10,000 individuals were successfully mutated. All these individuals were classified using RevealDroid*, yielding the results showed in Fig. 4. Each series in the figure is related to a specific configuration of the genetic algorithm, where an increment of 1 means that it is possible to generate individuals with up to 30 changes in the feature vector, whereas an increment of 0.6 reduces the number of changes to around 1. The reasons for this relationship between the increment parameter and the number of possible changes lies in the probability used internally by the genetic algorithm. Since every change must be manually injected into the application by the attacker, we may assume that in most cases the attacker would be interested in applying the minimum number of changes needed to achieve misclassification as a different family. This situation is represented by an increment equal to 0.6. In contrast, if the number of changes is not an issue for the attacker, a higher value of this parameter can be considered. As Fig. 4 shows, the label delivered by RevealDroid* differs considerably from the fake label pursued by the attacker, thereby notifying of a potential classification attack. With a maximum increment of 0.6 (around 1, 2 or 3 changes injected), using 14 different classifiers RevealDroid* will fail to detect the attack in 0.9% of the cases (false negatives), which means that the evasion will be detected in 99.1% of the evaluations.

The false positives of the countermeasure are computed by evaluating RevealDroid* with RevealDroid's test data (this data has non-mutated fresh samples for RevealDroid and RevealDroid*). The level of disagreement is calculated using the test output. Fig. 6 (top) shows that the false positive rate of our countermeasure ranges between 8% and 13.5%. Because the focus

Fig. 4. False negative rates for the countermeasure with respect to the number of classifiers used in RevealDroid*.

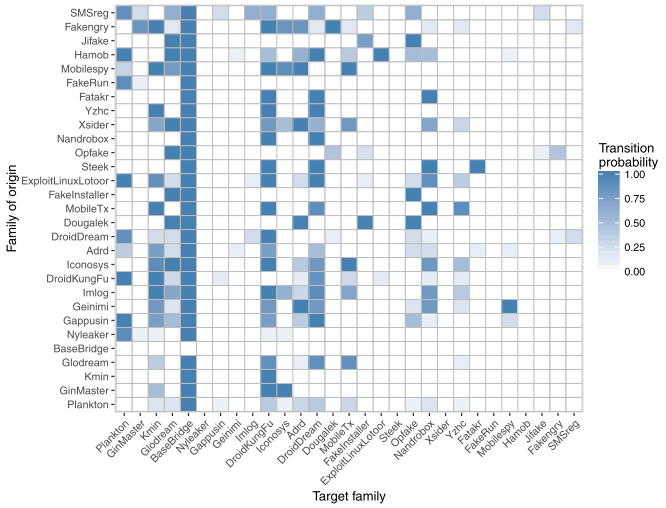


Fig. 5. Transition matrix to target families.

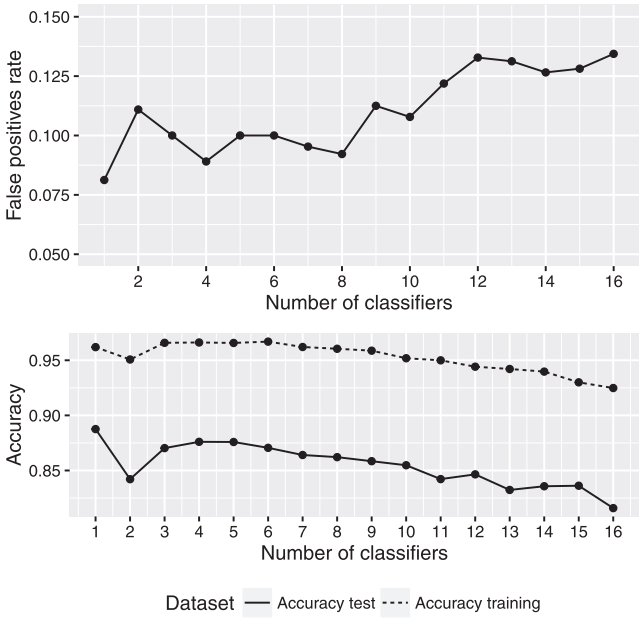


Fig. 6. False positive rate for the countermeasure (top) and accuracy of RevealDroid* depending on the number of classifiers used in RevealDroid* construction.

of the triage process is avoiding false negatives, we consider this result reasonable.

5.2. Reversing the attack

Once a sample has been suspected as the result of a misclassification attack, determining its original family is the next natural step. Reversing the transformation process that the attacker may have implemented is a complex task, particularly because of the difficulty of differentiating between the original app behaviour and the actions deliberately injected to cause the classification error.

However, the search process used during the attack offers the means to evaluate a number of possible original family classification candidates. The search was used between each possible pair of families in order to evaluate the transition probabilities between them (as the number of individuals belonging to a specific family able to reach a target family divided by the total number of individuals in the original family). The results of this experiment are shown in Fig. 5. The fitness function used here is the one described in Section 2.5, which allows one to target specific families. Since this matrix represents all possible transitions between original malware samples of different families and mutated samples, it is also possible to use this artefact to reveal the possible source families of an application detected as misclassified. Furthermore, it is also possible to order these candidate families by the transition probabilities.

For instance, Fig. 7 shows the probabilities of being the original family of a malware sample classified as Kmin family, according to the corresponding row of Fig. 5. In this example, there are 6 potential source families in which all the individuals were successfully mutated to be classified as Kmin, and these form a set of 6 prospective original families.

6. Related work

In this section we discuss the context for our work as it relates to Android static analysis, adversarial machine learning and countermeasures.

6.1. Android static analysis

Our work is focused on attacking a machine learning algorithm which operates on a space generated by static analysis

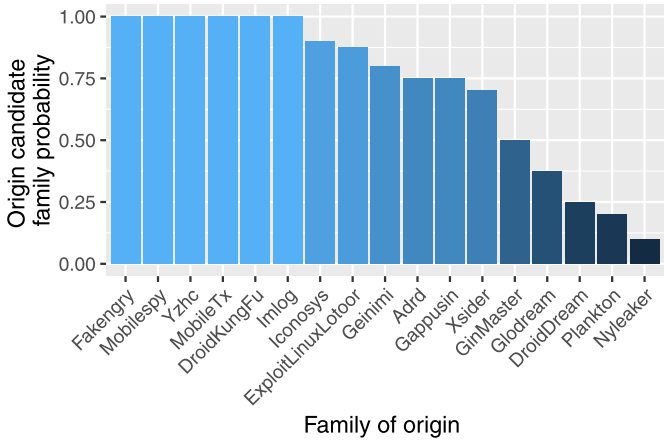


Fig. 7. Probabilities of being the origin family of a malware sample mutated to Kmin family.

features. We discuss the most relevant static analyses for our work: permissions, API calls, intent actions and flows.

Permissions have been identified as potential signifiers of malicious intentions. Tools like Kirim (Enck, Ongtang, & McDaniel, 2009) were used to detect anomalous settings containing malicious behaviour and tools like DroidRanger (Zhou, Wang, Zhou, & Jiang, 2012) leverage heuristics to perform the same task. API calls can be used to detect malware and generate signatures, which is the case for DroidLegacy (Deshotels, Notani, & Lakhotia, 2014) and DroidAPIMiner (Aafer, Du, & Yin, 2013). Current trends use both flow analysis, i.e. information leaks between data sources and potentially malicious sinks, and intent actions, remote procedures where one application can use the privileges of another one to perform malicious activities. Flows have been studied using tools such as FlowDroid (Arzt et al., 2014) and DroidSafe (Gordon et al., 2015), while intents have been studied in different ways: from the detection of communication vulnerabilities using ComDroid (Chin et al., 2011); to validation of the interaction between components with Epicc (Octeau et al., 2013); to points to communication between objects in different applications using Amandroid (Wei, Roy, & Ou, 2014); and to hybridization these methods, as seen in DidFail (Klieber, Flynn, Bhosale, Jia, & Bauer, 2014) which hybridizes Epicc and FlowDroid to improve detection through aggregated information.

Other work, out of the scope of our analysis but also related to static analysis for Android, uses a description language to identify semantic-based signatures, such as Apposcopy (Feng, Anand, Dillig, & Aiken, 2014), or aims to detect the context that triggers the malicious behaviour, such as AppContext (Yang et al., 2015) and TriggerScope (Fratantonio et al., 2016).

In our work, we target techniques that use static analysis features and leverage machine learning algorithms to detect or classify malware. These techniques, provided in Table 1, use the previously discussed tools to extract feature vectors that feed a machine learning algorithm. Tools like DroidSIFT (Zhang, Duan, Yin, & Zhao, 2014) and DroidAPIMiner (Aafer et al., 2013) have only been used for the detection problem, in which malware and goodware must be discriminated, while tools like Dendroid (Suarez-Tangil, Tapiador, Peris-Lopez, & Blasco, 2014), DroidLegacy (Deshotels et al., 2014), Drebin (Arp et al., 2014), DroidMiner (Yang, Xu, Gu, Yegneswaran, & Porras, 2014) and RevealDroid (Garcia et al., 2015) have also been used for family classification, with RevealDroid covering the largest spectrum in the feature space. This was the main reason for choosing RevealDroid as the targeted classifier in our work. We also targeted the triage problem, which is closely related to the family classification problem as Lakhotia et al. state during the description of their tool VILO (Lakhotia et al., 2013). This problem has also been examined from a detection perspective using ranking based algorithms in MAST (Chakradeo et al., 2013). Our goal here was to attack the triage process using adversarial machine learning.

6.2. Adversarial machine learning

Evasion and Adversarial Learning (Huang, Joseph, Nelson, Rubinstein, & Tygar, 2011) are widely studied topics in both the machine learning and computer security areas (Barreno et al., 2006; Lowd & Meek, 2005; Ptacek & Newsham, 1998). Given the success of machine learning techniques for addressing security related problems such as malware analysis, spam identification, or intrusion detection, testing the resilience and robustness of these approaches against an informed adversary is a necessary activity.

There is a wide spectrum of applications of machine learning algorithms in classification problems. Their reliability is closely linked to the reliability of the systems that depend on them. Adversarial learning is then an important problem that must be addressed. According to Barreno et al., the main weaknesses of machine learning algorithms lie precisely in their adaptation ability, which can be exploited by attackers to cause deliberate errors (Barreno, Nelson, Joseph, & Tygar, 2010). This presents a complex issue, since machine learning theory takes as its basis that the training dataset used in a learning process remains representative of the problem domain and assumes intentionally harmful modifications of the data do not happen (Laskov & Lippmann, 2010).

The problem of learning in hostile environments was first considered by Kearns and Li (1993). In this work, the authors developed an extension to Valiant's *Probably Approximately Correct* (PAC) framework (Valiant, 1984; 1985). The extension allows the algorithm to learn even when a dataset has been polluted with erroneous data, introduced by an active adversary. This adversarial behaviour is modelled following a worst-case approach (i.e., unbounded computational power and access to the classification history are assumed). The main contribution of this work was to provide methods to limit the maximum portion of the dataset polluted by the adversary without having a negative effect on the classification result.

The proliferation of classification and detection tools relying on machine learning techniques has promoted an increased interest in attacking these tools, taking advantage of the weaknesses in classification algorithms. These attacks are very varied and depend mainly on the adversarial model considered, since the capabilities of the attacker and her knowledge about the classifier define the impact of the attack.

All these attacks against machine learning can be categorised by point of view. From a coarse perspective, the attacks can be classified in two categories: poisoning attacks (Biggio, Nelson, & Laskov, 2012) and evasion attacks (Xu et al., 2016). In the former case, the attack is performed during the training stage. In this scenario the adversary introduces fake or malformed data into the training set. This will lead the classifier to learn an inaccurate model and then classify further instances incorrectly. In the latter case, the attack is performed during the classification stage. The feature vector belonging to a particular sample is modified so as to force the classifier to produce a wrong label. The proposed attack in this paper falls in the evasion category as we try to fool an already trained model by distorting the feature vector of a particular sample.

Barreno et al. (2006) provide an extended taxonomy of the different attacks against machine learning applications. They model attack spaces using three key concepts: influence (*whether it affects the training stage or the classification itself*), specificity (*whether the*

attack tries to misdirect the classification of data belonging to a particular class or, alternatively, causes no discrimination to happen) and the security property violated by the attacker (*whether the attack is against the classifier's availability or against the result's integrity*).

A practical example of how classification algorithms can be successfully evaded is the classifier-agnostic attack strategy described by Biggio et al. for assessing the security of machine learning applications (Biggio et al., 2013). They propose an adversarial strategy based on gradient descent attacks. They consider different threat models depending on how much information the attacker has regarding the attacked classifier. The authors demonstrate how their strategy could be employed to evade classifiers such as SVM and neural networks trained for detecting malicious PDF files.

In an approach similar to our own work but applied to a different problem, Vigna, Robertson, and Balzarotti (2004) developed a framework for measuring the resilience of a signature-based Network intrusion detection system (NIDS) against an adversary. The authors employed mutation strategies for modifying known exploits. Mutations were applied at network, application and code level, and included modifying the shape of network packets, injecting malformed data, and hiding malicious code using polymorphic engines. Ten real world exploits were mutated using different strategies and used to measure the resilience of two NIDS products. The experiment confirmed that evading the NIDS signatures is feasible, especially when combining different mutation techniques. Although this research has the injection of specific information to provoke a malfunction in a detection system in common with the attack that we describe in this paper, the problem varies significantly, since we are focused on malware classification.

Other research focused on NIDS was presented by Pastrana et al. (2011). It also takes advantage of genetic programming, in this case as a search heuristic for finding modification routines capable of evading a particular NIDS. These modification routines take a malicious network packet as input and apply different adjustments (e.g.: changing a particular value within the data payload, altering the TCP header, etc.). The authors tested the framework against C4.5 and Naïve–Bayes. By using this genetic search, the authors obtained individuals able of inducing non-negligible error rates in both classifiers, attaining a 37% classification error rate in the Naïve–Bayes classifier. In our work, by contrast, we are finding modifications to evade a correct family malware classification, rather than evading its detection (thus assuming that the sample will still be detected as malware with high probability).

Genetic programming has also been successfully employed in fooling the detection of malicious code. In particular Xu et al. presented EvadeML, a framework for automatically evading PDF malware classifiers (Xu et al., 2016). PDF files have been frequently used by attackers as hosts for embedded malware. The authors of that paper employed genetic search for finding the best modification strategy leading to evasion of detection by two PDF malware detection systems (PDFrate and Hidost) built on top of machine learning solutions. Up to 500 malicious payloads were successfully evaded using the discovered strategies. Again, an evolutionary algorithm is used to evade detection rather than to evade a correct classification between malware families.

Another example of adversarial learning to evade the detection of malicious PDF files is the *mimicry attack* (Maiorca, Corona, & Giacinto, 2013) that injects malicious code into a benign file using 3 different strategies: injecting an EXE (EXEembed), a PDF (PDFembed) or a Javascript (JSinject) payload. The evaluation of these tools shows high detection evasion effectiveness (100% for EXEembed and PDFembed and 80% for JSinject) on the 6 variants generated by the authors. Again, in contrast to our attack, this research is not focused on re-shaping malware for evading a correct classification and the domain is different. The evasion of PDF detection has been extensively analysed in Laskov (2014), demonstrating the vulnerabilities of a known online PDF analyser to this kind of attacks. The interaction between malware families has indeed been studied but from an unsupervised learning perspective, using clustering algorithms (Biggio, Rieck et al., 2014). Here, the authors inject new samples into the training process with the aim of disrupting the result.

On the Android side, there are new evasion strategies which aim to attack machine learning (Grosse, Papernot, Manoharan, Backes, & McDaniel, 2016; Meng et al., 2016) and antivirus systems (Aydogan & Sen, 2015; Meng et al., 2016; Xue et al., 2017; Zheng, Lee, & Lui, 2012). The first technique in this area was ADAM (Zheng et al., 2012), which manipulates malware via re-packing and obfuscation. ADAM was created to audit antivirus systems and it showed good effectiveness against VirusTotal, reaching an evasion rate close to a 50%. In a similar line, Aydogan and Sen (2015) include a genetic programming framework to the obfuscation process, reporting an evasion effectiveness up to 33% against 8 antivirus systems. The effectiveness of evasion strategies was extended to new machine learning techniques, such as deep learning, by Grosse et al. (2016), who were able to reach up to an 80% evasion rate by adding perturbations to the malware variants through junk code. This effective strategy was also followed by Mystique (Meng et al., 2016), and its extension, Mystique-S (Xue et al., 2017). The former uses a multi-objective genetic algorithm to reduce the classification rate of machine learning algorithms and anti-viruses, while it maximizes the attack behaviour; the later generates the code dynamically to reach the same goal. They are able to evade the detectors up to 80% of the time for Mystique and 94% of the time for Mystique-S. An interesting case for evasion, out-of-the-box from the previous techniques, was introduced by Vidas and Christin (2014) who generated an attack based on red pills, i.e., detecting environmental conditions. This strategy combines the detection of behaviour, performance, hardware and software components. They were able to reach an 86% evasion rate. Our tool, IagoDroid, is focused on attacking the static analysis features of a machine learning classifier, and it is able to reach a 97% evasion rate. Compared with the other related tools of the state of the art, it is a competitive result (for a summary comparing all the above mentioned tools, see Table 2.

6.3. Counteracting adversarial learning techniques

The security community has worked both on testing classification systems built on top of machine learning techniques against different kinds of attacks and on designing countermeasures to deal with this problem. For instance, Chinavle et al. studied the effect of employing learning ensembles for combatting adversaries in a spam detection scenario (Chinavle et al., 2009). Their approach demonstrated that through the use of different classifiers, it is possible to detect performance degradation (due to evasion attacks on behalf of a motivated adversary) and automatically repair this condition. Their approach allows the system to maintain a high degree of accuracy through time while reducing the number of re-training stages.

Barreno et al. elaborated on the security and reliability of machine learning (Barreno et al., 2006), proposing a framework to evaluate the security of a particular machine learning application. In the same line, Biggio, Fumera, and Roli (2014) proposed a framework to introduce countermeasures against attackers while designing the classifier, instead of applying them later during training or test stages. The main contribution of this work is the lack of bounds for a particular classifier, making the framework flexible.

Table 2
A comparison among different evasion methodologies related to IagoDroid, separated in general techniques and Android specific.

Method	Target	Type of attack	Evasion rate
PDF and Network			
Pastrana (Pastrana et al., 2011)	C4.5 & Naïve Bayes	Network injection	37%
Vigna (Vigna et al., 2004)	Network intrusion detectors	Mutation of exploits	90%
Biggio (Biggio et al., 2013)	SVM & Nearest Neighbour	Noise injection & Gradient Descent	up to 100%
EvadeMl (Xu et al., 2016)	PDFRate & Hidost	Genetic Programming	90%
EXEembed (Maiorca et al., 2013)	PDF malware detectors	EXE payload embedding	up to 100%
PDFembed (Maiorca et al., 2013)	PDF malware detectors	PDF embedding	up to 100%
JSinject (Maiorca et al., 2013)	PDF malware detectors	Javascript embedding	up to 80%
Laskov (Laskov, 2014)	PDFRate	Noise injection	up to 72%
Biggio (Biggio, Rieck et al., 2014)	Behavioural Clustering	Data poisoning	76%
Android			
Mystique (Meng et al., 2016)	Anti-virus & Machine Learning	Code injection & Genetic Algorithms	up to 80%
Mystique-S (Xue et al., 2017)	Anti-virus	Dynamic code generation	94%
Vidas (Vidas & Christin, 2014)	Dynamic Analysis tools	Red Pills	86%
Grosse (Grosse et al., 2016)	Deep Learning	Perturbation	up to 80%
ADAM (Zheng et al., 2012)	Anti-virus	Re-packing and obfuscation	up to 50%
Aydogan (Aydogan & Sen, 2015)	Anti-virus	Genetic Programming and Obfuscation	up to 33%
IagoDroid	RevealDroid	Code injection & Genetic Algorithms	97%

Table 3
Summary of experimental results. The table shows, for each malware family, the number of generations required to find a first solution, the average number of generations required to achieve convergence, the average number of modifications, and the feature that is most frequently changed.

Family	First sol.	Avg. conv.	Avg. mod.	Feature
Plankton	1	3.3	1.0	ACTION_INPUT_METHOD_CHANGED (0.7)
GinMaster	1	3.7	1.0	SMS_MMS (0.6)
Kmin	1	4.3	1.0	ACTION_USER_PRESENT (0.6)
Glodream	1	4.7	0.8	ACTION_INPUT_METHOD_CHANGED (0.4)
BaseBridge	Inf	Inf	–	–
Nyleaker	1	3.6	1.0	NETWORK__LOG (0.4)
Gappusin	1	3.4	0.9	ACTION_INPUT_METHOD_CHANGED (0.3)
Geinimi	1	3.9	1.0	NETWORK_INFORMATION (0.5)
Imlog	1	4.7	1.2	ACTION_INPUT_METHOD_CHANGED (0.7)
DroidKungFu	1	7.2	0.7	IPC__NETWORK (0.2)
Iconosys	1	3.5	1.1	NETWORK__LOG (0.3)
Adrd	1	3.6	0.8	ACTION_INPUT_METHOD_CHANGED (0.5)
DroidDream	1	4.1	0.8	ACTION_INPUT_METHOD_CHANGED (0.4)
Dougalek	1	3.5	1.0	ACTION_INPUT_METHOD_CHANGED (0.4)
MobileTx	1	3.2	1.0	FILE (0.5)
FakeInstaller	1	3.5	1.0	ACTION_INPUT_METHOD_CHANGED (0.5)
ExploitLinuxLotoor	1	2.1	0.8	ACTION_INPUT_METHOD_CHANGED (0.4)
Steek	1	3.9	1.0	ACTION_USER_PRESENT (0.4)
Opfake	1	4.8	0.9	ACTION_INPUT_METHOD_CHANGED (0.5)
Nandrobox	1	3.2	1.0	ACTION_INPUT_METHOD_CHANGED (0.4)
Xsider	1	3.1	1.0	ACTION_INPUT_METHOD_CHANGED (0.6)
Yzhc	1	4.5	0.8	ACTION_USER_PRESENT (0.4)
Fatakr	1	3.2	1.0	ACTION_USER_PRESENT (0.7)
FakeRun	1	4.4	1.0	ACTION_INPUT_METHOD_CHANGED (0.4)
Mobilespy	1	3.1	0.9	ACTION_MAIN (0.4)
Hamob	1	3.4	1.0	ACTION_INPUT_METHOD_CHANGED (0.3)
Jifake	1	2.6	0.8	android.net (0.3)
Fakengry	1	2.6	0.6	UNIQUE_IDENTIFIER_DB_INFORMATION (0.2)
SMSreg	1	1.6	0.9	ACTION_INPUT_METHOD_CHANGED (0.3)

Dalvi et al. (2004) studied the development of robust classifiers. They addressed the problem as a game between the attacker and the target classifier. In their approach, the attacker's strategy is used as input for generating a classifier resilient to particular adversarial behaviour. Addressing the problem from a game theoretical perspective, the authors improved the working of vanilla Naïve–Bayes classifier in a spam detection case, dramatically reducing the number of errors.

From a more general point of view, the effectiveness of different strategies that deal with evasion attacks has been studied elsewhere. For instance, Support Vector Machines have been evaluated (Russu, Demontis, Biggio, Fumera, & Roli, 2016), concluding that the selection of the kernel function is crucial. Feature selection based countermeasures have been studied (Budhraja & Oates, 2015), showing that this can be counterproductive since it reduces the accuracy of the classifier in some cases. There is also a framework focused on evaluating the potential attack scenarios due to the use of feature selection methods (Xiao et al., 2015).

Although the above countermeasures are able to successfully narrow the effects produced by attacks on machine learning classifiers, they are mainly focused on detection problems: a binary classification between benign and malicious software. However, a classification task into different malware families constitutes a different scenario in which there can be an important number of classes closely located in the search space.

7. Conclusions

IagoDroid demonstrates that any Android malware classification scheme that relies exclusively on static analysis during triage is a sensitive process that can easily be destabilised. IagoDroid is able to fool the RevealDroid classifier into misclassifying the family for

28 out of 29 families in the dataset by modifying a single feature of the original malware. In the process, this attack generates up to 14,000 new variants for 290 malware samples in just 2 min.

As a countermeasure, we split the feature space into different overlapping sets where different classifiers work together to detect potential evasions. This method, named RevealDroid*, is demonstrably effective for a small number of modifications but less useful when the number of modifications is high. In the latter case, it is able to reduce the number of evasive variants that IagoDroid generates, but cannot prevent it from generating at least some. In consequence, RevealDroid* forces producers of malware variants to find techniques to modify a higher number of features during the variants generation process. In the case where an evasive file is detected, our countermeasure is also able to track the original malware family, providing an opportunity to reconsider the malware priority during the triage process.

This countermeasure shows some of the limitations of the evasion method. The first limitation is related to the adversarial environment. IagoDroid has full knowledge of the underlying classifier. This limits the possibility of having strong results in different scenarios, even when the technique may still be applicable. Another significant limitation of the technique is in the final generation of the variants, which in the current version requires human intervention to transform the suggested vector of changes into the actual variant.

These limitations inspire several, possible lines of future work, starting from measuring the ability of IagoDroid to cause misclassification in commercial tools, such as antivirus engines. This line of research would require an extension to the tool's capabilities such as providing automatic injection of changes within opaque predicates. From a research perspective, IagoDroid is useful for studying the limitations on the robustness of machine learning classifiers and, indeed, our future work will focus on defining sound measures based on evasion abilities. This will help to understand which classifications algorithms are stronger than others when faced with adversaries for algorithms based on both static and dynamic analysis. Finally, the backtracking ability of our countermeasure can be understood as a Markov model among families and transitions. This knowledge can be used to study more deeply the different relationships among Android malware families.

Acknowledgements

This work has been supported by the following grants: EphemeCH (MINECO TIN2014-56494-C4-4-P) and CIBERDINE (CM S2013/ICE-3095), both under the European Regional Development Fund FEDER; SeMaMatch EP/K032623/1 and InfoTestSS EP/P006116/1 from EPSRC; SPINY (MINECO TIN2013-46469-R) and SMOG-DEV (MINECO TIN2016-79095-C2-2-R) and Justice Programme of the European Union (2014-2020) 723180 – RiskTrack – JUST-2015-JCOO-AG/JUST-2015-JCOO-AG-1. The contents of this publication are the sole responsibility of their authors and can in no way be taken to reflect the views of the European Commission.

References

Aafer, Y., Du, W., & Yin, H. (2013). Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems* (pp. 86–103). Springer.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., & Rieck, K. (2014). Drebin: Effective and explainable detection of android malware in your pocket.. *Ndss*.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., et al. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices, 49*(6), 259–269.

Aydogan, E., & Sen, S. (2015). Automatic generation of mobile malwares using genetic programming. In *European conference on the applications of evolutionary computation* (pp. 745–756). Springer.

Barreno, M., Nelson, B., Joseph, A. D., & Tygar, J. D. (2010). The security of machine learning. *Machine Learning, 81*(2), 121–148. doi:10.1007/s10994-010-5188-5.

Barreno, M., Nelson, B., Sears, R., Joseph, A. D., & Tygar, J. D. (2006). Can machine learning be secure? In *Proceedings of the 2006 ACM symposium on information, computer and communications security* (pp. 16–25). ACM.

Biggio, B., Corona, I., Fumera, G., Giacinto, G., & Roli, F. (2011). Bagging classifiers for fighting poisoning attacks in adversarial classification tasks. In *International workshop on multiple classifier systems* (pp. 350–359). Springer.

Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., et al. (2013). Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases* (pp. 387–402). Springer.

Biggio, B., Fumera, G., & Roli, F. (2014). Security evaluation of pattern classifiers under attack. *IEEE Transactions on Knowledge and Data Engineering, 26*(4), 984–996.

Biggio, B., Nelson, B., & Laskov, P. (2012). Poisoning attacks against support vector machines. In *Proceedings of the 29th international conference on machine learning, ICML 2012, Edinburgh, Scotland, Uk, June 26 - July 1, 2012*.

Biggio, B., Rieck, K., Ariu, D., Wressnegger, C., Corona, I., Giacinto, G., et al. (2014). Poisoning behavioral malware clustering. In *Proceedings of the 2014 workshop on artificial intelligent and security workshop* (pp. 27–36). ACM.

Budhraja, K. K., & Oates, T. (2015). Adversarial feature selection. In *2015 IEEE international conference on data mining workshop (ICDMW)* (pp. 288–294). IEEE.

Chakradeo, S., Reaves, B., Traynor, P., & Enck, W. (2013). Mast: triage for market-scale mobile malware analysis. In *Proceedings of the sixth ACM conference on security and privacy in wireless and mobile networks* (pp. 13–24). ACM.

Chin, E., Felt, A. P., Greenwood, K., & Wagner, D. (2011). Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on mobile systems, applications, and services* (pp. 239–252). ACM.

Chinavle, D., Kolari, P., Oates, T., & Finin, T. (2009). Ensembles in adversarial classification for spam. In *Proceedings of the 18th ACM conference on information and knowledge management* (pp. 2015–2018). ACM.

Dalvi, N., Domingos, P., Sanghai, S., & Verma, D. (2004). Adversarial classification. In *Proceedings of the tenth ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 99–108). ACM.

Dash, S. K., Suarez-Tangil, G., Khan, S., Tam, K., Ahmadi, M., Kinder, J., et al. (2016). Droidscribe: Classifying android malware based on runtime behavior. In *Mobile security technologies (MoST 2016)* (pp. 1–12). 7kearns1993learning148

Deshotels, L., Notani, V., & Lakhotia, A. (2014). Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on program protection and reverse engineering workshop 2014* (p. 3). ACM.

Enck, W., Ongtang, M., & McDaniel, P. (2009). On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on computer and communications security* (pp. 235–245). ACM.

Feng, Y., Anand, S., Dillig, I., & Aiken, A. (2014). Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering* (pp. 576–587). ACM.

Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., & Vigna, G. (2016). Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)* (pp. 377–396). doi:10.1109/SP.2016.30.

Gandotra, E., Bansal, D., & Sofat, S. (2014). Malware analysis and classification: A survey. *Journal of Information Security, 5*(02), 56.

Garcia, J., Hammad, M., Pedrood, B., Bagheri-Khaligh, A., & Malek, S. (2015). Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. *Technical Report*. Department of Computer Science, George Mason University.

Gordon, M. I., Kim, D., Perkins, J. H., Gilham, L., Nguyen, N., & Rinard, M. C. (2015). Information flow analysis of android applications in droidsafe.. *NDSS*. Citeseer.

Grosse, K., Papernot, N., Manoharan, P., Backes, M., & McDaniel, P. (2016). Adversarial perturbations against deep neural networks for malware classification. arXiv preprint arXiv:1606.04435

Huang, L., Joseph, A. D., Nelson, B., Rubinstein, B. I., & Tygar, J. (2011). Adversarial machine learning. In *Proceedings of the 4th ACM workshop on security and artificial intelligence* (pp. 43–58). ACM.

Kearns, M., & Li, M. (1993). Learning in the presence of malicious errors. *SIAM Journal on Computing, 22*(4), 807–837.

Klieber, W., Flynn, L., Bhosale, A., Jia, L., & Bauer, L. (2014). Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN international workshop on the state of the art in java program analysis* (pp. 1–6). ACM.

Labs, M. (2016). McAfee labs threats report. http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-may-2016.pdf. [Online; Accessed 19.07.2016].

Lakhotia, A., Walenstein, A., Miles, C., & Singh, A. (2013). Vilo: A rapid learning nearest-neighbor classifier for malware triage. *Journal of Computer Virology and Hacking Techniques, 9*(3), 109–123.

Laskov, P., & Lippmann, R. (2010). Machine learning in adversarial environments. *Machine Learning, 81*(2), 115–119. doi:10.1007/s10994-010-5207-6.

Laskov, P. (2014). Practical evasion of a learning-based classifier: A case study. In *2014 IEEE symposium on security and privacy* (pp. 197–211). IEEE.

Lowd, D., & Meek, C. (2005). Adversarial learning. In *Proceedings of the eleventh ACM SIGKDD international conference on knowledge discovery in data mining* (pp. 641–647). ACM.

Maiorca, D., Corona, I., & Giacinto, G. (2013). Looking at the bag is not enough to find the bomb: An evasion of structural methods for malicious pdf files detection. In *Proceedings of the 8th ACM SIGSAC symposium on information, computer and communications security* (pp. 119–130). ACM.

Meng, G., Xue, Y., Mahinthan, C., Narayanan, A., Liu, Y., Zhang, J., et al. (2016). Mystique: Evolving android malware for auditing anti-malware tools. In *Proceedings of the 11th ACM on Asia conference on computer and communications security* (pp. 365–376). ACM.

Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., et al. (2013). Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX security symposium (USENIX security 13)* (pp. 543–558).

Pastrana, S., Orfila, A., & Ribagorda, A. (2011). A functional framework to evade network ids. In *System sciences (HICSS), 2011 44th Hawaii international conference on* (pp. 1–10). IEEE.

Perdisci, R., Gu, G., & Lee, W. (2006). Using an ensemble of one-class SVM classifiers to harden payload-based anomaly detection systems. In *Sixth international conference on data mining (ICDM'06)* (pp. 488–498). IEEE.

Ptacek, T. H., & Newsham, T. N. (1998). Insertion, evasion, and denial of service: Eluding network intrusion detection. *Technical Report*. DTIC Document.

Rasthofer, S., Arzt, S., & Bodden, E. (2014). A machine-learning approach for classifying and categorizing android sources and sinks.. *NDSS*.

Russu, P., Demontis, A., Biggio, B., Fumera, G., & Roli, F. (2016). Secure kernel machines against evasion attacks. In *Proceedings of the 2016 ACM workshop on artificial intelligence and security* (pp. 59–69). ACM.

Sivanandam, S., & Deepa, S. (2007). *Introduction to genetic algorithms*. Springer Science & Business Media.

Suarez-Tangil, G., Tapiador, J. E., Peris-Lopez, P., & Blasco, J. (2014). Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications, 41*(4), 1104–1117.

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM, 27*(11), 1134–1142.

Valiant, L. G. (1985). Learning disjunction of conjunctions.. In *IJCAI* (pp. 560–566).

Vidas, T., & Christin, N. (2014). Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on information, computer and communications security* (pp. 447–458). ACM.

Vigna, G., Robertson, W., & Balzarotti, D. (2004). Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the 11th ACM conference on computer and communications security* (pp. 21–30). ACM.

Wei, F., Roy, S., & Ou, X. (2014). Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security* (pp. 1329–1341). ACM.

Xiao, H., Biggio, B., Brown, G., Fumera, G., Eckert, C., & Roli, F. (2015). Is feature selection secure against training data poisoning?. In F. Bach, & D. Blei (Eds.), *JMLR W&CP-proceedings of the 32nd international conference on international conference on machine learning (ICML): Vol. 37* (pp. 1689–1698).

Xu, W., Qi, Y., & Evans, D. (2016). Automatically evading classifiers. In *Proceedings of the 2016 network and distributed systems symposium*.

Xue, Y., Meng, G., Liu, Y., Tan, T. H., Chen, H., Sun, J., & Zhang, J. (2017). Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Transactions on Information Forensics and Security, 12*(7), 1529–1544.

Yang, C., Xu, Z., Gu, G., Yegneswaran, V., & Porras, P. (2014). Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *European symposium on research in computer security* (pp. 163–182). Springer.

Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., & Enck, W. (2015). Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE international conference on software engineering: Vol. 1* (pp. 303–313). IEEE.

Zhang, M., Duan, Y., Yin, H., & Zhao, Z. (2014). Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security* (pp. 1105–1116). ACM.

Zheng, M., Lee, P. P., & Lui, J. C. (2012). Adam: an automatic and extensible platform to stress test android anti-virus systems. In *International conference on detection of intrusions and malware, and vulnerability assessment* (pp. 82–101). Springer.

Zhou, Y., & Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy* (pp. 95–109). IEEE.

Zhou, Y., Wang, Z., Zhou, W., & Jiang, X. (2012). Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS: Vol. 25* (pp. 50–52).

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[ABKT16] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: collecting millions of android apps for the research community. In *13th International Conference on Mining Software Repositories, MSR 2016*, pages 468–471. ACM, 2016.

[ADY13] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International Conference on Security and Privacy in Communication Systems*, pages 86–103. Springer, 2013.

[Alt92] Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[And13] AndroMalShare dataset. `http://sanddroid.xjtu.edu.cn:8080/`, 2013. Last accessed: 2018-03-14.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[ASH+14] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: effective and explainable detection of android malware in your pocket. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014*, pages 1–12, 2014.

[BNS+06] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D Joseph, and J Doug Tygar. Can machine learning be secure? In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 16–25. ACM, 2006.

[Bre96] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[Bre99] Leo Breiman. Pasting small votes for classification in large databases and on-line. *Machine learning*, 36(1-2):85–103, 1999.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[C+15] François Chollet et al. Keras. `https://keras.io`, 2015. Last accessed: 2018-11-13.

[cam18] Cambridge english dictionary, 2018.

[CBHK02] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[cca13] Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0), 2013. Last accessed: 2018-03-14.

[CDLM+15] Gerardo Canfora, Andrea De Lorenzo, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Effectiveness of opcode ngrams for detection of multi family android malware. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 333–340. IEEE, 2015.

[CKOF09] Deepak Chinavle, Pranam Kolari, Tim Oates, and Tim Finin. Ensembles in adversarial classification for spam. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 2015–2018. ACM, 2009.

[CMM+18] Alejandro Calleja, Alejandro Martín, Héctor D. Menéndez, Juan Tapiador, and David Clark. Picking on the family: Disrupting android malware triage by forcing misclassification. *Expert Systems with Applications*, 95:113–126, 2018.

[Cor17] F-Secure Corporation. F-secure state of cyber security 2017. https://fsecurepressglobal.files.wordpress.com/2017/02/cyber-security-report-2017.pdf, Jan 2017.

[CRTE13] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. Mast: Triage for market-scale mobile malware analysis. In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, pages 13–24. ACM, 2013.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[DG13] Antony Desnos and Geoffroy Gueguen. Androguard. `https://github.com/androguard/androguard`, 2013. Last accessed: 2018-11-13.

[DHQ+14] Ken Dunham, Shane Hartman, Manu Quintans, Jose Andre Morales, and Tim Strazzere. *Android Malware and Analysis*. Auerbach Publications, 2014.

[DNL14] Luke Deshotels, Vivek Notani, and Arun Lakhotia. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, PPREW'14, pages 3:1–3:12, 2014.

[DSTK+16] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. Droidscribe: Classifying android malware based on runtime behavior. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 252–261. IEEE, 2016.

[Ehr17] Jesse M Ehrenfeld. Wannacry, cybersecurity and health information technology: A time to act. *Journal of medical systems*, 41(7):104, 2017.

[F-S] F-Secure Labs. Trojan: Android/SLocker Description | F-Secure Labs. Technical report, F-Secure. Last accessed: 2018-12-04.

[FAS+17] Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, Guillermo Suarez-Tangil, and Steven Furnell. Androdialysis: Analysis of android intent effectiveness in malware detection. *Computers & security*, 65:121–134, 2017.

[Fin14] Gernot A Fink. *Markov models for pattern recognition: from theory to applications*. Springer Science & Business Media, 2014.

[Fri01] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 29(5):1189–1232, 2001.

[FSW14] Stephen Feldman, Dillon Stadther, and Bing Wang. Manilyzer: automated android malware detection through manifest analysis. In *Mobile Ad Hoc and Sensor Systems (MASS), 2014 IEEE 11th International Conference on*, pages 767–772. IEEE, 2014.

[gar17] Gartner says worldwide information security spending will grow 7 percent to reach $86.4 billion in 2017. https://www.gartner.com/en/newsroom/press-releases/2017-08-16-gartner-says-worldwide-information-security-spending-will-grow-7-percent-to-reach-86-billion-in-2017, August 2017. Last accessed: 2018-12-04.

[GBCB16] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[GEW06] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[GHP+15] Joshua Garcia, Mahmoud Hammad, Bahman Pedrood, Ali Bagheri-Khaligh, and Sam Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. *Department of Computer Science, George Mason University, Tech. Rep*, 2015.

[GPM+16] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*, 2016.

[Gru18] Ben Gruver. smali/baksmali. `https://github.com/JesusFreke/smali`, 2018.

[GYAR13] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

[HPK11] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[HRZZ09] Trevor Hastie, Saharon Rosset, Ji Zhu, and Hui Zou. Multi-class adaboost. *Statistics and its Interface*, 2(3):349–360, 2009.

[iso12] Iso/iec 27000:2009, Dec 2012.

[JKW+16] Jae-wook Jang, Hyunjae Kang, Jiyoung Woo, Aziz Mohaisen, and Huy Kang Kim. Andro-dumpsys: anti-malware system based on the similarity of malware creator and malware centric information. *Computers & security*, 58:125–138, 2016.

[JWHT13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[Lan11] Patrik Lantz. An android application sandbox for dynamic analysis. *Master's Thesis, Department of Electrical and Information Technology, Lund university, Lund, Sweden*, 2011.

[Lan15] Patrik Lantz. Droidbox: Dynamic analysis of android apps. `https://github.com/pjlantz/droidbox`, 2015.

[Lue18] Christian Lueg. 3.002.482 new android malware samples in 2017. https://www.gdatasoftware.com/blog/2018/02/30491-some-343-new-android-malware-samples-every-hour-in-2017, Feb 2018.

[LWMS13] Arun Lakhotia, Andrew Walenstein, Craig Miles, and Anshuman Singh. Vilo: a rapid learning nearest-neighbor classifier for malware triage. *Journal of Computer Virology and Hacking Techniques*, 9(3):109–123, 2013.

[Mal05] Marcus A. Maloof. *Machine Learning and Data Mining for Computer Security: Methods and Applications (Advanced Information and Knowledge Processing)*. Springer-Verlag, 2005.

[MBSZ11] Federico Maggi, Andrea Bellini, Guido Salvaneschi, and Stefano Zanero. Finding non-trivial malware naming inconsistencies. In *International Conference on Information Systems Security*, pages 144–159. Springer, 2011.

[MCC17] Alejandro Martín, Raúl Cabrera, and David Camacho. Droidbox-andropytool. `https://github.com/alexMyG/DroidBox_AndroPyTool`, 2017.

[MCC18] Alejandro Martín, Raúl Cabrera, and David Camacho. Andropytool: A framework for automated extraction of static and dynamic features from android applications. `https://github.com/alexMyG/AndroPyTool`, 2018.

[MCM+16] Alejandro Martín, Alejandro Calleja, Héctor D. Menéndez, Juan Tapiador, and David Camacho. Adroit: Android malware detection using meta-information. In *Computational Intelligence (SSCI), 2016 IEEE Symposium Series on*, pages 1–8. IEEE, 2016.

[MFHNC17] Alejandro Martín, Félix Fuentes-Hurtado, Valery Naranjo, and David Camacho. Evolving deep neural networks architectures for android malware classification. In *Evolutionary Computation (CEC), 2017 IEEE Congress on*, pages 1659–1666. IEEE, 2017.

[MGF+15] Misael Mongiovì, G. Giannone, Andrea Fornaia, Giuseppe Pappalardo, and Emiliano Tramontana. Combining static and dynamic data flow analysis: A hybrid approach for detecting data leaks in java applications. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1573–1579. ACM, 2015.

[MHCC18] Alejandro Martín, Julio Hernández-Castro, and David Camacho. An in-depth study of the jisut family of android ransomware. *IEEE Access*, pages 1–1, 2018.

[Mit97] Thomas M. Mitchell. *Machine learning*. McGraw-Hill, Inc., 1997.

[MLCC18] Alejandro Martín, Raúl Lara-Cabrera, and David Camacho. Android malware detection through hybrid features fusion and ensemble classifiers: the andropytool framework and the omnidroid dataset. *Information Fusion*, DOI: 10.1016/j.inffus.2018.12.006, 2018.

[MLCFH+17] Alejandro Martín, Raúl Lara-Cabrera, Félix Fuentes-Hurtado, Valery Naranjo, and David Camacho. Evodeep: a new evolutionary approach for automatic deep neural networks parametrisation. *Journal of Parallel and Distributed Computing*, 2017.

[MM00] Gary McGraw and Greg Morrisett. Attacking malicious code: A report to the infosec research council. *IEEE software*, (5):33–41, 2000.

[MMC16a] Alejandro Martin, Héctor D. Menéndez, and David Camacho. Genetic boosting classification for malware detection. In *Evolutionary Computation (CEC), 2016 IEEE Congress on*, pages 1030–1037. IEEE, 2016.

[MMC16b] Alejandro Martín, Héctor D. Menéndez, and David Camacho. String-based malware detection for android environments. In *International Symposium on Intelligent and Distributed Computing*, pages 99–108. Springer, Cham, 2016.

[MMC16c] Alejandro Martín, Héctor D. Menéndez, and David Camacho. Studying the influence of static api calls for hiding malware. In *Conference of the Spanish Association for Artificial Intelligence*, pages 363–372. Springer, Cham, 2016.

[MMC17] Alejandro Martín, Héctor D Menéndez, and David Camacho. Mocdroid: multi-objective evolutionary classifier for android malware detection. *Soft Computing*, 21(24):7405–7415, 2017.

[MRC18] Alejandro Martín, Lara-Cabrera Raúl, and David Camacho. A new tool for static and dynamic android malware analysis. In *Data Science and Knowledge Engineering for Sensing Decision Support*, pages 509–516, 2018.

[MRFC18] Alejandro Martín, Víctor Rodríguez-Fernández, and David Camacho. Candyman: Classifying android malware families by modelling dynamic traces with markov chains. *Engineering Applications of Artificial Intelligence*, 74:121–133, 2018.

[MXM+16] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen. Mystique: Evolving android malware for auditing anti-malware tools. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 365–376. ACM, 2016.

[OM99] David Opitz and Richard Maclin. Popular ensemble methods: An empirical study. *Journal of artificial intelligence research*, 11:169–198, 1999.

[PVA+14] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[PZ13] Naser Peiravian and Xingquan Zhu. Machine learning for android malware detection using permission and api calls. In *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*, pages 300–305. IEEE, 2013.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[Qui14] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[RF16] B. Rashidi and C. Fung. Xdroid: An android permission control using hidden markov chain and online learning. In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 46–54, Oct 2016.

[RFB17] Bahman Rashidi, Carol Fung, and Elisa Bertino. Android resource usage risk assessment using hidden markov model and online learning. *Computers & Security*, 65:90–107, 2017.

[RLMM09] Rick Rogers, John Lombardo, Zigurd Mednieks, and Blake Meike. *Android application development: Programming with the Google SDK*. O'Reilly Media, Inc., 2009.

[Rob14] Christian Robert. Machine learning, a probabilistic perspective, 2014.

[SAN15] Shina Sheen, R Anitha, and V Natarajan. Android based malware detection using a multifeature collaborative decision fusion approach. *Neurocomputing*, 151:905–912, 2015.

[Sax18] Joshua Saxe. *Malware Data Science: attack detection and attribution*. No Starch Press, 2018.

[SH12] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. No Starch Press, 2012.

[SKE+12] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. "andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[SRKC16] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[STDA+17] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 309–320. ACM, 2017.

[STTPLB14] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Jorge Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(4):1104–1117, 2014.

[Vir] Virustotal. `https://www.virustotal.com/`. Accessed: 2018-09-30.

[VM02] Giorgio Valentini and Francesco Masulli. Ensembles of learning machines. In *Italian Workshop on Neural Nets*, pages 3–20. Springer, 2002.

[Win12] R Winsniewski. Android–apktool: A tool for reverse engineering android apk files, 2012.

[WLR+17] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 252–276. Springer, 2017.

[WLW+18] Wei Wang, Yuanyuan Li, Xing Wang, Jiqiang Liu, and Xiangliang Zhang. Detecting android malicious apps and categorizing benign apps with ensemble of classifiers. *Future Generation Computer Systems*, 78:987–994, 2018.

[WMW+12] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. DroidMat: Android malware detection through manifest and API calls tracing. In *Seventh Asia Joint Conference on Information Security, AsiaJCIS 2012*, pages 62–69. IEEE, 2012.

[XLD16] Ke Xu, Yingjiu Li, and Robert H Deng. Iccdetector: Icc-based malware detection on android. *IEEE Transactions on Information Forensics and Security*, 11(6):1252–1264, 2016.

[XQE16] Weilin Xu, Yanjun Qi, and David Evans. Automatically evading classifiers. In *Proceedings of the 2016 Network and Distributed Systems Symposium*, 2016.

[YLWX14] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 371–372. ACM, 2014.

[YS18] Suleiman Y Yerima and Sakir Sezer. Droidfusion: A novel multilevel classifier fusion approach for android malware detection. *IEEE Transactions on Cybernetics*, 2018.

[YSM14] Suleiman Y Yerima, Sakir Sezer, and Igor Muttik. Android malware detection using parallel machine learning classifiers. In *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, pages 37–42. IEEE, 2014.

[YSMM13] Suleiman Y Yerima, Sakir Sezer, Gavin McWilliams, and Igor Muttik. A new android malware detection approach using bayesian classification. In *2013 IEEE 27th international conference on advanced information networking and applications (AINA)*, pages 121–128. IEEE, 2013.

[Yu13] Rowland Yu. Ginmaster: a case study in android malware. In *Virus bulletin conference*, pages 92–104, 2013.

[YXG+14] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *European Symposium on Research in Computer Security*, pages 163–182. Springer, 2014.

[ZDYZ14] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.

[ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy, SP 2012*, pages 95–109. IEEE Computer Society, 2012.

[ZJY+17] Dali Zhu, Hao Jin, Ying Yang, Di Wu, and Weiyi Chen. Deepflow: Deep learning-based malware detection by mining android application for abnormal usage of sensitive data. In *Computers and Communications (ISCC), 2017 IEEE Symposium on*, pages 438–443. IEEE, 2017.

[ZYZ+18] Hui-Juan Zhu, Zhu-Hong You, Ze-Xuan Zhu, Wei-Lei Shi, Xing Chen, and Li Cheng. Droiddet: Effective and robust detection of android malware using static analysis along with rotation forest model. *Neurocomputing*, 272:638–646, 2018.