Universitat Oberta de Catalunya

uoc.edu

Inteligencia Artificial para la detección de binarios maliciosos.

Autor: Jorge Díaz Navarro
Máster Universitario en Ingeniería Informática
Área de Inteligencia Artificial

Joan M. Nuñez do Rio
Carles Ventura Royo

Enero 2022

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Inteligencia Artificial para la detección de binarios maliciosos*
Nombre del autor:	*Jorge Díaz Navarro*
Nombre del consultor/a:	*Joan M. Nuñez do Rio*
Nombre del PRA:	*Carles Ventura Royo*
Fecha de entrega (mm/aaaa):	01/2022
Titulación:	*Máster Universitario en Ingeniería Informática*
Área del Trabajo Final:	*Inteligencia Artificial*
Idioma del trabajo:	*Castellano*
Palabras clave	*Ciberseguridad, Inteligencia Artificial, Reversing*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados y conclusiones del trabajo.*

La ejecución de *malware* en los sistemas informáticos es un problema que afecta a toda la sociedad y cuyas consecuencias económicas no dejan de aumentar año tras año. La mejora de las capacidades técnicas de los binarios maliciosos para camuflarse y no ser detectados limitan la capacidad de protección de los antivirus tradicionales, lo que supone un riesgo muy elevado para la seguridad de organizaciones y particulares.

Ante esta situación, la Inteligencia Artificial ofrece técnicas efectivas que mejoran las capacidades defensivas actuales en materia de detección de *malware*. Sin embargo, las investigaciones actuales no aportan herramientas o referencias detalladas que permitan a los profesionales de la Ciberseguridad mejorar sus productos propios de detección, por lo que en muchos de los casos las limitaciones actuales siguen estando vigentes.

A través de este trabajo se han creado múltiples modelos efectivos de *Machine Learning* que han sido capaces de detectar muestras de *malware* no conocidas previamente.

Asimismo, se han aportado los detalles, códigos y referencias necesarias para que los profesionales de la Ciberseguridad dispongan de la capacidad de crear sus propios modelos de detección.

Abstract (in English, 250 words or less):

The execution of malware in computer systems is a problem that affects society, and whose economic consequences are increasing every year. The evolving technical capabilities of malicious binaries, which allow them to remain undetected, limit the protection capabilities of traditional anti-virus software, posing a very high risk to the security of organizations and individuals.

In this situation, Artificial Intelligence offers effective techniques that improve current defensive capabilities in malware detection. However, current investigations do not provide detailed tools or references that allow cybersecurity professionals to improve their own detection products, so in many cases the current limitations are still in place.

Through this work, multiple effective Machine Learning models have been created that have been able to detect previously unknown malware samples.

Furthermore, all the relevant details, codes and references have been provided so that cybersecurity professionals are able to create their own detection models.

Índice

Índice de figuras

Índice de tablas

1. Introducción

1.1 Contexto y justificación del Trabajo

El impacto económico producido por ciberataques ha aumentado de forma considerable en los últimos años. Según *Cybersecurity Ventures* [1], el coste total anual superó los 325 millones de dólares en 2015, los 5.000 millones en 2017 y los 11.500 millones en 2019, siguiendo una tendencia ascendente.

Para que un ciberataque sea exitoso, es necesaria la ejecución, en los sistemas informáticos de la víctima, de un programa capaz de dañar y modificar su comportamiento habitual. A este tipo de programas se les conoce como *malware* [2], término proveniente de *"malicious software*".

La Ciberseguridad, como disciplina destinada a la protección de los entornos tecnológicos [3], se ha erigido como la encargada de salvaguardar los intereses de la sociedad ante los ciberataques.

En este sentido, los profesionales del sector de la Ciberseguridad, con el propósito de evitar la ejecución de *malware*, han desarrollado y perfeccionado, a lo largo de los años, los programas conocidos como antivirus [4]. Este tipo de *software* de protección, cuando detecta la existencia de un nuevo binario (programa) en un sistema informático, determina si puede ser o no ejecutado en base a una serie de reglas de detección.

Al margen de la metodología empleada, el proceso de análisis de *malware* puede realizarse, o bien de forma estática, o bien de forma dinámica. En lo que respecta al análisis estático, la muestra binaria se analiza sin ser ejecutada. Por su parte, en el análisis dinámico sí que se requiere la ejecución del *malware*.

En todo caso, la figura del profesional de la seguridad, que es quien debe crear las reglas de detección de forma manual, está siempre presente. Este hecho, sumado a una serie de limitaciones que presentan las técnicas empleadas en el proceso de análisis de *malware*, tiene como consecuencia un riesgo muy elevado para la seguridad de organizaciones y particulares.

En primer lugar, se carece de los recursos necesarios para tener en cuenta todas las capacidades técnicas, y cada vez más avanzadas, de las que disponen los binarios maliciosos para camuflarse y no ser detectados.

En segundo lugar, el número diario de nuevos desarrollos de *malware* es muy superior al número de muestras que pueden ser analizadas de forma manual.

En tercer lugar, el proceso de creación de reglas manuales de detección limita el número de binarios maliciosos que pueden detectarse, pues no se pueden tener en cuenta las capacidades de todos los nuevos desarrollos.

Por último, la demanda mundial de profesionales de la ciberseguridad sigue superando a la oferta [5]. Este hecho, sumado a todo lo anterior, acrecienta las limitaciones actuales.

La necesidad de detectar los ciberataques en sus primeras fases es vital; cuanto mayor sea el tiempo de detección, mayor será el daño en los sistemas informáticos infectados, y aumentarán considerablemente las probabilidades de propagación a otros dispositivos, aumentando pues el impacto final y sus consecuencias.

Ante esta situación de desequilibro, el uso de técnicas de Inteligencia Artificial aplicadas a la Seguridad Informática se ha mostrado efectivo, y su uso se ha ido normalizando en los últimos años, como en la clasificación de SPAM y la detección de documentos maliciosos [6].

Prueba de ello es la capacidad para crear modelos efectivos de detección de anomalías, sin depender en exclusiva de los casos de uso, reglas manuales, y pericia del analista. Es decir, se ha comprobado como el uso de técnicas de Inteligencia Artificial mejora las capacidades en ciberdefensa y reduce el tiempo de detección y de análisis [7].

En lo que respecta al análisis de binarios maliciosos, diversos estudios, como los realizados por *Hongfa et al.,* [8] *Gibert et al.* [9] y *Chen et al.,* [10], han demostrado que el uso de técnicas de Inteligencia Artificial es viable y efectivo en la detección de *malware*.

Sin embargo, en la comunidad no se encuentran proyectos o herramientas de código abierto para el gran público, por lo que, a efectos prácticos, los analistas deben seguir empleando las técnicas manuales habituales para realizar su trabajo.

Con el objetivo de mitigar las limitaciones de accesibilidad, en el presente proyecto se sientan las bases para la creación de un modelo de clasificación que pueda ser empleado por los analistas de Ciberseguridad para el análisis de *malware*, haciendo uso de técnicas de Inteligencia Artificial. Con este fin, el proceso se divide en dos fases claramente diferenciadas.

Una primera fase, consistente en la creación y estudio de un *dataset* (conjunto de datos) compuesto por binarios tanto legítimos como maliciosos, que se han analizado según las técnicas de análisis dinámico de *malware*.

Una segunda fase, consistente en la creación de un modelo predictivo basado en técnicas de Inteligencia Artificial, para poder clasificar binarios según si son legítimos o *malware*. Para el aprendizaje de los modelos, se ha empleado el *dataset* creado durante la primera fase.

1.2 Objetivos del Trabajo

1.2.1 Objetivos generales

Los objetivos generales de este Trabajo Fin de Máster son los siguientes:

- Realizar un proceso de análisis de *malware* de diversas muestras, con la finalidad de crear y justificar un *dataset* de binarios legítimos y maliciosos.

- Experimentar diferentes métodos y técnicas para obtener modelos basados en técnicas de Inteligencia Artificial capaces de clasificar binarios, dependiendo si son legítimos o no.

1.2.2 Objetivos específicos

Los objetivos específicos de este Trabajo Fin de Máster son los siguientes:

- Analizar la literatura actual sobre detección de *malware*.
- Analizar la literatura actual sobre Inteligencia Artificial.
- Seleccionar un conjunto de muestras binarias, tanto maliciosas como legítimas.
- Clasificar las muestras del conjunto de datos de forma manual, según si son legítimas o maliciosas.
- Determinar qué metodologías van a emplearse para realizar un análisis de *malware* de las muestras.
- Determinar qué características son de utilidad para representar el comportamiento de un binario.
- Implementar entorno para análisis de binarios, a través del uso de una *sandbox* dentro de una Máquina Virtual Ubuntu.
- Crear *dataset* de binarios legítimos y no legítimos con las características seleccionadas, para su posterior análisis mediante técnicas de Inteligencia Artificial.
- Normalizar y transformar las características del *dataset* para adaptarlas a las necesidades de entrada del modelo de Inteligencia Artificial.
- Realizar las modificaciones necesarias para mejorar la calidad del *dataset*.
- Determinar qué algoritmos de *Machine Learning Supervisado* son capaces de clasificar binarios según si son legítimos o no.
- Crear modelos predictivos partir del *dataset* creado durante la primera fase de desarrollo.
- Analizar y validar los resultados.

1.3 Enfoque y método seguido

En la primera fase de desarrollo del proyecto se ha creado un *dataset* de binarios legítimos y maliciosos. Las características del conjunto de datos se han determinado y extraído según los criterios de la literatura actual, especialmente en lo que se refiere a los análisis dinámicos.

Durante esta primera fase, con el propósito de acotar el conjunto de binarios a analizar, se ha creado el dataset teniendo en cuenta únicamente binarios de tipo PE (*Portable Executable*) *Windows.* Son dos las razones que han motivado esta estrategia. En primer lugar*, Windows* es el Sistema Operativo (S.O) más empleado por los usuarios comunes [11], por lo que la mayoría de los desarrollos de *malware* se idean para ser compatibles en este S.O. En segundo lugar, los binarios de tipo PE son aquellos diseñados específicamente para el Sistema Operativo *Windows* [12].

En una segunda fase de desarrollo del proyecto, se ha creado un entorno diseñado para estudiar, entrenar y validar el conjunto de datos creado durante la primera fase. Todo ello se ha realizado mediante la aplicación de múltiples modelos de entrenamiento, según las técnicas de *Machine Learning* supervisadas, dado que el *dataset* se encuentra etiquetado dependiendo de si el binario es malicioso o no.

1.4 Planificación del Trabajo

La planificación del trabajo se establece a partir de las siguientes tareas a realizar:

1. Desarrollo de conceptos
    - Estudio de *malware* y capacidades defensivas contra ciberataques
    - Estudio de técnicas de *Machine Learning*.
2. Estudio del contexto tecnológico sobre Seguridad Informática y *Machine Learning*.
3. Puesta a punto de máquinas virtuales con *VirtualBox* y *VKM* para análisis de binarios.

4. Creación del *dataset* a través de la extracción de características de binarios.

5. Estudio y validación del *dataset*.

6. Puesta a punto de laboratorio virtual para desarrollo del modelo de *Machine Learning*.

7. Desarrollo de un modelo de *Machine Learning* para aprendizaje y clasificación de binarios según si son legítimos y no legítimos.

    - Escoger y justificar el lenguaje de programación.

    - Escoger y justificar la técnica: supervisado contra no supervisado.

    - Escoger y justificar la elección de algoritmos de *Machine Learning*.

    - Desarrollar y validar modelos predictivos.

8. Desarrollo de memoria con los resultados obtenidos.

La estimación en tiempo esperada es la siguiente:

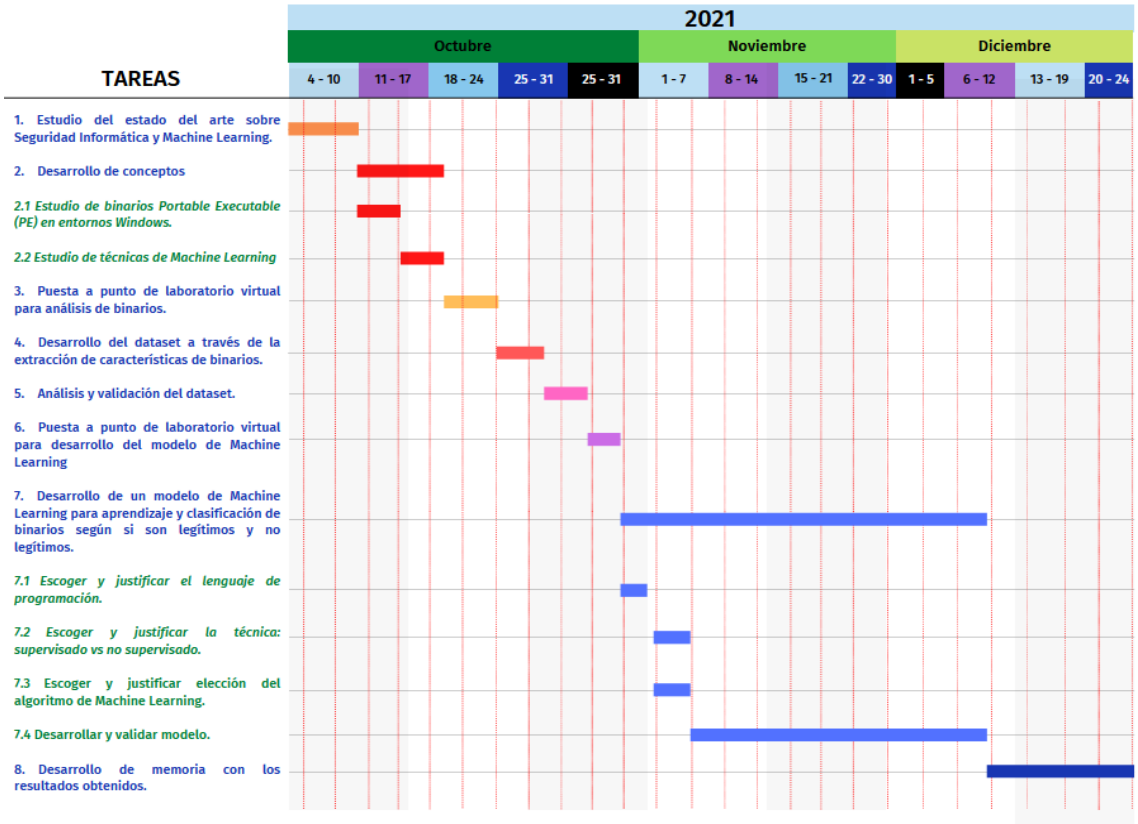

Figura 1. Estimación de la duración del proyecto.

1.5 Breve sumario de productos obtenidos

De este proyecto se espera obtener los siguientes productos:

- Entorno para creación y estudio de *dataset* de binarios.
- Entorno para estudio de *dataset* mediante técnicas de *Machine Learning* Supervisado.
- Entorno para predicción de nuevas muestras de binarios.

1.6 Breve descripción de los otros capítulos de la memoria

En el Capítulo 2 se han revisado los conceptos necesarios para entender cómo los profesionales de la Ciberseguridad protegen a la sociedad de los ciberataques, en concreto de la ejecución de *malware* en los sistemas informáticos de las víctimas.

En el Capítulo 3 se han revisado los conceptos relacionados con la inteligencia Artificial, haciendo hincapié en aquellos relacionados con el desarrollo propuesto de este Trabajo Fin de Máster.

En el Capítulo 4 se ha revisado el contexto tecnológico actual, en lo que se refiere a la aplicación de técnicas de *Machine Learning* para la detección de binarios.

En el Capítulo 5 se ha llevado a cabo el desarrollo de la solución propuesta en este trabajo. Para ello, en una primera fase, se ha creado y estudiado un *dataset* compuesto por binarios tanto legítimos como maliciosos, que se han analizado previamente según las técnicas de análisis dinámico de *malware*. En una segunda fase, se han creado y validado múltiples modelos de clasificación basados en técnicas de *Machine Learning* supervisado, con la finalidad de poder detectar nuevas muestras de binarios maliciosos. Para el aprendizaje de los modelos, se ha empleado el *dataset* creado durante la primera fase.

En el Capítulo 6 se han expuesto las conclusiones y el trabajo futuro.

Finalmente, para referencia del lector, en el Capítulo 7 se encuentra un glosario de los términos más comunes empleados en este documento.

2. Fundamentos de la Ciberseguridad

La finalidad de este punto es dar a conocer los conceptos relacionados con la Ciberseguridad, en materia de detección y prevención de la ejecución de *malware.*

2.1 *Malware*

El incesante aumento de los ciberataques en los últimos años responde al nivel de dependencia tecnológica en el que se encuentra la sociedad. Este hecho se ha visibilizado con mayor intensidad durante la pandemia como consecuencia de la COVID-19, donde los niveles de dependencia han aumentado de forma notable, aumentando así también el número de ciberataques [13].

En el transcurso de los procesos necesarios para que un ciberataque sea exitoso, existe la necesidad de ejecutar, dentro del entorno informático de la víctima, un programa malicioso conocido como *malware*. El término *malware*, derivado de *"malicious software"*, hace referencia a todo tipo de programa diseñado y desarrollado con el propósito de robar, destruir o modificar la información de los sistemas informáticos.

Existen diversos tipos de *malware*, clasificados según el propósito de la infección y/o los mecanismos que emplea para ello.

2.1.1 Tipos principales de *malware*

Se conoce como *spyware* [14] al tipo de *malware* que busca obtener información sensible de la víctima, como cuentas bancarias, contraseñas y datos de navegación, entre otros. Una variante del *spyware* es el *keylogger* [15]. Si bien es cierto que el propósito de ambos tipos de *malware* es el mismo, el *keylogger* actúa registrando el uso que hace la víctima de su teclado.

Otro tipo de *malware*, conocido como troyano [16], modifica el comportamiento habitual de un programa legítimo con la finalidad de realizar actividades maliciosas en segundo plano. Dado que la apariencia y las características principales del programa legítimo no se modifican, la actividad maliciosa se produce sin el conocimiento de la víctima.

Para que un ciberataque sea efectivo, algunas de las acciones maliciosas que se ejecutan en el dispositivo de la víctima necesitan permisos elevados, también conocidos como permisos de administrador. El tipo de *malware* conocido como *rootkit* [17], permite al cibercriminal realizar actividad maliciosa que requiere este tipo de privilegios.

Uno de los tipos de *malware* más conocidos por la sociedad es el *ransomware* [18]. Este tipo de programa malicioso cifra los datos de la víctima y no le permite hacer uso de sus sistemas informáticos. La finalidad de un ciberataque de estas características es obtener un beneficio económico, pues a la víctima se le solicita un rescate por recuperar su información.

Por último, existe un tipo de *malware* conocido como gusano [19], cuyo mecanismo de infección es la replicación de sí mismo, infectando a otros sistemas. El objetivo principal de los ciberataques de este tipo es el consumo del ancho de banda de los sistemas informáticos de la víctima, evitando que pueda hacer uso de los recursos de red, como Internet.

2.2 Capacidades defensivas contra la ejecución de *malware*

Los costes económicos por ciberataques han superado la barrera de los 11.500 millones de euros [1]. Por este motivo, detectar y evitar la ejecución de los programas maliciosos es un objetivo prioritario para los profesionales de la Ciberseguridad.

A lo largo de los años, desde el sector de la Ciberseguridad se han desarrollado y perfeccionado los antivirus [4] como programa de detección y protección contra el *malware*.

La función principal de este tipo de *software* es evitar que se ejecute *software* no deseado en los entornos informáticos. Con este fin, cada vez que detecta la existencia de un nuevo binario en el sistema, lo analiza a través de la extracción de sus características principales, que son comparadas con una serie de reglas de detección que posee el propio antivirus. Por lo tanto, si alguna de las características del binario coincide con alguna de las reglas de detección, el antivirus lo considera malicioso y evita su ejecución, evitando también la infección del sistema.

El proceso mediante el cual se crean nuevas reglas de detección tiene dos fases. En primer lugar, el binario ha de analizarse, o bien de forma estática, o bien de forma dinámica. Esta primera fase permite explorar y conocer el propósito y comportamiento del *malware*. En segundo lugar, según los resultados del análisis del binario, el analista crea una serie de reglas de detección, o bien a través de métodos basados en firmas, o bien a través de métodos heurísticos.

En los siguientes subapartados se presentan tanto los distintos tipos de *reversing* (análisis) de *malware*, como los distintos métodos de creación de reglas de detección. Por último, se presentan las limitaciones de las capacidades defensivas como consecuencia de las mejoras técnicas que incorporan los desarrollos de *malware* más avanzados.

2.2.1 Análisis estático de *malware*

Se conoce como análisis estático de *malware* al estudio que se realiza de un binario sin la necesidad de ejecutarlo [20]. En este caso, el estudio del binario se lleva a cabo sobre su código fuente, siendo el analista el encargado de comprender su funcionamiento y extraer aquellas características que considere maliciosas. Las herramientas que facilitan el proceso de análisis son de tipo *disassembler*, *debugger* y *decompiler*.

2.2.1.1 Herramientas de análisis estático de *malware*

Tal y como se ha mencionado en la introducción, los binarios suelen estar compilados. Es decir, su lenguaje se transforma para ser legible por la computadora, lo que imposibilita su comprensión para el ser humano. La función de un *disassembler* [21] es transformar el lenguaje máquina de un binario, a otro entendible por el ser humano.

Por otra parte, existen los *debugger* [22] como herramienta de análisis. Este tipo de programas permiten la inspección del comportamiento del binario, a través de su ejecución instrucción a instrucción. No se considera un método de análisis dinámico porque el analista tiene el control, en todo momento, de qué se ejecuta y qué no. Es decir, el analista puede ejecutar únicamente partes concretas del binario, o puede incluso modificar previamente las instrucciones maliciosas del código para evitar el riesgo de infección.

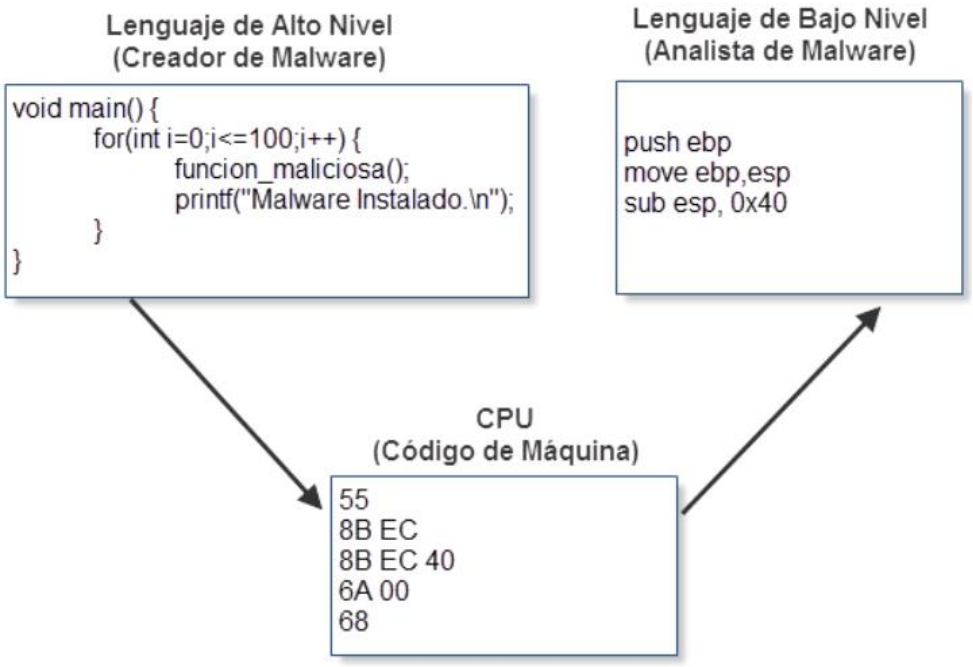

Figura 2. Niveles de abstracción del lenguaje de un binario.

Fuente: https://www.welivesecurity.com/la-es/2014/01/14/bases-analisis-estatico-malware-bases-desensamblado/

Por último, para facilitar el proceso de análisis estático se puede hacer uso de un *decompiler* [23]. Del mismo modo que ocurre con el *disassembler*, la finalidad de esta herramienta es transformar el lenguaje máquina para ser interpretable por el ser humano. No obstante, tomando como referencia la Figura 2, el *disassembler* genera un lenguaje de bajo nivel, mientras que el *decompiler* lo genera de alto nivel. La diferencia entre ambos reside en que el de alto nivel es mucho más sencillo de comprender y estudiar.

2.2.1.2 Características extraíbles a través del análisis estático de *malware*

Las características que se pueden extraer de un binario tras su análisis de forma estática son las siguientes:

- *Strings* [24]. A través del análisis del código fuente de un binario se pueden extraer cadenas de texto (como ficheros, URLs, direcciones IP, etc.), que pueden aportar indicios del tipo de *malware* al que puede pertenecer el binario.

- *Opcode N-Grams* [25]. Un *n-gram* es una secuencia contigua de n elementos de una determinada secuencia de texto del código fuente. Los *opcode* (códigos de operación) son aquellas instrucciones que el binario ejecuta en lenguaje de bajo nivel. Por lo tanto, los *opcode n-grams* son el conjunto de secuencias de operaciones de bajo nivel que se extraen del código fuente del binario. De nuevo, tomando como referencia la Figura 2, se proporciona un ejemplo de *opcode n-grams*:

Tipo	**Conjuntos *opcode***
1-gram	*push, ebp, move, esp, sub*
2-gram	*push ebp, move ebp, ebp esp, sub esp*
3-gram	*move ebp esp*

Tabla 1. Ejemplos de opcode n-gram según la Figura 2.

- Llamadas a funciones de las Interfaces de Programación de Aplicaciones [26]. Las Interfaces de Programación de Aplicaciones, en adelante API, son bibliotecas de funciones predeterminadas de *Windows*. Gracias a ellas, los desarrolladores de *software* disponen de la posibilidad de hacer uso de múltiples funcionalidades de este Sistema Operativo, sin tener que programarlas desde cero. Algunos desarrollos de *malware* tratan de sobrescribir estas funciones para insertar código malicioso, o hacer uso de ellas para actividades malintencionadas. Por lo tanto, su estudio puede aportar información clave para representar el comportamiento del *malware*.

- Entropía [27]. Los desarrollos de *malware* más sofisticados suelen emplear técnicas de ofuscación para ocultar las cadenas de texto que pueden identificarles como maliciosos. Encontrar cadenas cifradas o comprimidas, por tanto, podría ser un indicador de que el binario no es legítimo.

2.2.1.3 Limitaciones del análisis estático de *malware*

Los desarrollos más avanzados de *malware* presentan la capacidad de ofuscar su código fuente. Desde el punto de vista de los análisis estáticos, donde es necesaria la revisión del código fuente, este hecho dificulta enormemente la tarea de comprensión del analista [28].

Se conoce como ofuscación [29] a la técnica que modifica el código fuente de un binario para hacerlo incomprensible. Este proceso se puede realizar, o bien de forma simple (dotando a las variables y funciones del código de una nomenclatura aleatoria), o bien de forma compleja (cifrando el contenido del código fuente).

```
# Código original
def ejecutaMalware (binario, tiempo):
        if (tiempo > 50):
                ejecuta(binario)
#Código ofuscado
def anteSi (cosa, esperanza):
        otros = 0
        flor i in range (0, 100):
                otros = otros + 1
        if otros > 0 and esperanza > 50:
                estepe(cosa)
```

Figura 3. Ejemplo básico de ofuscación de código.

2.2.2 Análisis dinámico de *malware*

Se conoce como análisis dinámico de *malware* al estudio que se realiza de un binario a través de su ejecución [30]. Gracias a este tipo de análisis, se puede trazar su comportamiento a través de su actividad.

Las características que se pueden extraer son, entre otras, las llamadas a procesos, funciones y comunicaciones desde/hacia Internet. Por seguridad, el código malicioso se ejecuta en un entorno virtual controlado para evitar que el *malware* se propague hacia otros sistemas.

2.2.2.1 Características extraíbles a través del análisis dinámico de *malware*

Las características principales que se pueden extraer de un binario tras su análisis de forma dinámica son las siguientes:

- Eventos del registro [31]. El registro es una de las principales fuentes de información para que un binario pueda conocer el entorno del Sistema Operativo donde se está ejecutando. Asimismo, su uso es común para todos los tipos de *malware*. A modo de ejemplo, a través del registro un binario malicioso puede obtener persistencia para no ser eliminado cuando el usuario reinicia el sistema. Por lo tanto, tener la posibilidad de analizar qué acciones realiza un binario sobre el registro, puede revelar información que indique si su comportamiento es malicioso o no.

- Tráfico de red [32]. En la mayoría de las ocasiones, el *malware* trata de conectar con el exterior, o bien para la extracción de información, o bien para ser controlado de forma remota. Por lo tanto, disponer del tráfico de red que genera un binario aporta información de gran interés. A modo de ejemplo, se podrían analizar las direcciones IP según su geolocalización. De este modo, si se detecta alguna conexión desde/hacia un país del cuál no se espera mantener ningún tipo de conexión, se podría considerar como indicador de comportamiento malicioso.

- Trazas de llamada a la API. Al igual que ocurre con los análisis estáticos, a través de los análisis dinámicos pueden extraerse las trazas de llamada a la API de *Windows*.

- Trazas de instrucciones [33]. Durante la ejecución de un binario se puede extraer el conjunto de acciones que realiza en el sistema (creación de ficheros temporales, la creación de variables en el registro, borrado de información, etc.). Estas pueden analizarse para detectar actividad sospechosa del binario.

- Eventos en la memoria [34]. Cualquier acción realizada en un sistema informático se almacena temporalmente en su memoria principal. La información que se puede obtener es la suma de algunas de las características anteriormente expuestas, como por ejemplo las llamadas al sistema y la información de las conexiones de red. Asimismo, se puede obtener información adicional, como la lista de procesos en ejecución, el historial de comandos ejecutados en consola, las llamadas al núcleo del Sistema Operativo, etc.

2.2.2.2 Herramientas de análisis dinámico de *malware*

Para realizar un análisis dinámico de *malware* es necesaria su ejecución. Si el proceso no se realiza de forma controlada, los sistemas informáticos del analista podrían infectarse, con las consecuencias que ello implica.

Por seguridad, desde el sector de la Ciberseguridad se han desarrollado entornos virtuales seguros conocidos como *sandbox* [35], donde las muestras binarias se ejecutan y analizan de forma controlada y aislada en un entorno virtual específico, evitando la infección y la propagación del virus informático.

La diferencia entre ejecutar *malware* en una *sandbox,* respecto a su ejecución un entorno virtual genérico, es que una *sandbox* está configurada para simular que se trata de un entorno informático real. Asimismo, la *sandbox*, tras cada análisis, genera un informe que incluye información relacionada con las características mencionadas en el apartado anterior.

A pesar de que la finalidad de todas las *sandbox* es la misma, existen distintos desarrollos y no todos extraen las mismas características de un binario. Se muestra a continuación un esquema comparativo de las *sandbox* más relevantes junto al tipo de características principales que son capaces de extraer del análisis:

Sandbox	Características				
	Eventos del registro	**Tráfico de red**	**Llamadas a la API**	**Trazas de instrucciones**	**Eventos en memoria**
Cape [36]	Sí	Sí	Sí	Sí	Sí
Cuckoo [37]	Sí	Sí	Sí	Sí	Sí
Drakvuf [38]	No	Sí	Sí	Sí	No
Joe Sandbox [39]	Sí	Sí	No	Sí	Sí
F-Secure SEE [40]	No	No	No	No	Sí

Tabla 2. Comparativa de sandbox según características extraídas.

2.2.2.3 Limitaciones del análisis dinámico de *malware*

Los desarrollos más avanzados de *malware* disponen de la capacidad de detectar que están siendo ejecutados en una *sandbox* [41]. En tal caso, el binario no realiza ningún tipo de comportamiento que pueda ser considerado como malicioso, mostrando una actividad aparentemente legítima. Este hecho puede llegar a dificultar y tergiversar los resultados obtenidos a través de los análisis dinámicos de *malware*.

2.2.3 Reglas de detección basadas en firmas

En un sistema informático, cada uno de los binarios es poseedor de una firma que lo identifica de forma única e inequívoca. Las reglas basadas en firmas hacen uso de este atributo para detectar las muestras de *malware* ya conocidas [42].

Por lo tanto, cuando un analista realiza un análisis de *malware* y determina que la muestra es maliciosa, almacena su identificador único dentro de la base de datos de reglas.

De este modo, cada vez que un antivirus detecta un nuevo binario, busca si su identificador único coincide con alguno de los almacenados previamente. En caso positivo, el binario es reconocido como *malware*.

La ventaja de este método es su sencillez de cara a su implementación, así como la rapidez con la que un antivirus puede revisar si un identificador se encuentra dentro de su base de datos.

2.2.3.1 Limitaciones de las reglas de detección basadas en firmas

Cada día se desarrollan 230.000 nuevas muestras de *malware* [43], disponiendo todas ellas de su identificador único. Este hecho dificulta enormemente mantener actualizada la base de datos de firmas.

Asimismo, los desarrollos más avanzados de *malware,* conocidos como *malware* polimórficos y metamórficos, presentan la capacidad de evadir las reglas de detección basadas en firmas.

El polimorfismo [44] es la capacidad del *malware* para cambiar su estructura de código cada vez que se ejecuta una copia de este. Es decir, aunque la lógica y mecanismos de actuación sean siempre los mismos, su identificador único va a ser siempre distinto.

Por su parte, el *malware* metamórfico [45] tiene la capacidad de reprogramarse a sí mismo, modificando su código fuente en cada ejecución. Por lo tanto, su identificador único va a ser también siempre distinto en tiempo de ejecución.

Ambas capacidades avanzadas del *malware* dificultan su detección a través de métodos basados en firmas, pues no es posible contemplar todas y cada una de las variantes que genera.

2.2.4 Reglas de detección basadas en métodos heurísticos

Cuando un analista realiza el proceso de análisis de *malware,* extrae una serie de características que transforma en reglas de detección. Tal y como se ha estudiado en los apartados anteriores, las características extraídas dependerán de si el análisis se realiza de forma estática o dinámica.

La detección de *malware* basado en métodos heurísticos se centra en la comparativa de un binario con las reglas creadas por los analistas [46]. De este modo, cada vez que un antivirus detecta un nuevo binario, busca si alguna de sus características, estáticas y/o dinámicas, coinciden con alguna de las reglas que se han creado previamente. En caso positivo, el binario es reconocido como *malware*.

La ventaja de este método de reglas de detección, respecto a los basados en firmas, es su capacidad para detectar variantes de programas maliciosos existentes, y también de programas maliciosos no conocidos previamente. Este hecho se debe a que, cuando se analiza un binario y se extraen sus características, para considerarlo malicioso no es necesario que coincidan todas sus características con las existentes en el conjunto de reglas.

2.2.4.1 Limitaciones de las reglas de detección basadas en métodos heurísticos

Del mismo modo que ocurre con las reglas basadas en firmas, el número total de nuevas muestras de *malware* desarrolladas a diario afectan a la efectividad de este tipo de reglas. Si bien es cierto que este método es capaz de detectar variantes de muestras ya conocidas, no es posible tener en cuenta todas y cada una de las características de los nuevos desarrollos, por lo que van a surgir características que no coincidan con ninguna de las reglas creadas de forma manual.

Asimismo, este método de detección puede dar lugar a elevadas tasas de falsos positivos si las reglas no se crean adecuadamente.

2.2.5 Limitaciones de las capacidades defensivas actuales

La información expuesta hasta este momento evidencia una serie de deficiencias a considerar, en lo que respecta a las capacidades actuales de los sistemas de defensa para la detección de *malware*.

En primer lugar, los profesionales del sector de la Ciberseguridad carecen de los recursos necesarios para tener en cuenta todas las capacidades técnicas, y cada vez más avanzadas, de las que disponen los binarios maliciosos para camuflarse y no ser detectados.

En segundo lugar, el número diario de nuevos desarrollos de *malware* es muy superior al número de muestras que pueden ser analizadas de forma manual. Por lo tanto, se da la existencia binarios maliciosos que no disponen de reglas capaces de detectarlos, poniendo en riesgo la seguridad de organizaciones y particulares.

En tercer lugar, la demanda mundial de profesionales de la ciberseguridad sigue superando a la oferta [5]. Este hecho, sumado al incremento incesante del número de ciberataques anuales, donde cada día se desarrollan 230.000 nuevas muestras de *malware* [43], tienen como consecuencia que el sector de la Ciberseguridad no pueda proteger a la sociedad de todos y cada uno de los desarrollos de *malware* existentes.

Por último, la creación de reglas manuales de detección depende de la pericia e intuición del analista. Por lo tanto, si bien es cierto que las reglas heurísticas pueden detectar nuevas variantes de *malware*, no es posible tener en cuenta todas y cada una de las características de los nuevos desarrollos.

Para dar una muestra real de la importancia de esta tesitura, se propone como referencia el informe *Internet Security Report - Q1 2021* elaborado por *WatchGuard® Technologies* [47]. Entre los resultados más destacados, se encuentra el hecho de que el 74% de las amenazas detectadas durante el primer trimestre de 2021 fueron producidas por *malware* que no fue detectado por las soluciones de antivirus tradicionales. Es decir, las reglas manuales creadas por los analistas fueron útiles para detectar únicamente el 26% de los ciberataques. *Corey Nachreiner*, *Chief Security Officer* de *WatchGuard*, aseguró al respecto que "*Las soluciones antimalware tradicionales por sí solas son simplemente insuficientes para el entorno de amenazas actual. Todas las organizaciones necesitan una estrategia de seguridad proactiva en capas que incluya el aprendizaje automático y el análisis del comportamiento para detectar y bloquear las amenazas nuevas y avanzadas*" [48].

3. Fundamentos de la Inteligencia Artificial

El uso de la Inteligencia Artificial ha aumentado exponencialmente en los últimos años en prácticamente todos los sectores de la sociedad y en la industria. Prueba de ello son las importantes inversiones en investigación y desarrollo que están realizando hoy en día las grandes corporaciones multinacionales [49].

La finalidad de este punto es dar a conocer los conceptos relacionados con la Inteligencia Artificial, profundizando en aquellos que se han empleado en el desarrollo propuesto de este Trabajo Fin de Máster.

3.1 *Inteligencia Artificial*

La Inteligencia Artificial (IA) se define como «*la ciencia e ingenio de hacer máquinas inteligentes, especialmente programas de cómputo inteligentes*» [50]. En otras palabras, se le puede atribuir el componente de Inteligencia Artificial a cualquier máquina que tenga la capacidad de resolver una tarea a través de un comportamiento "inteligente".

El concepto original de Inteligencia Artificial es difuso y genérico. Por este motivo, con el paso de los años, han ido surgiendo diferentes ramas o campos de investigación más específicos que complementan la definición inicial.

En 2017 *Forrester Research*, a través del estudio *TechRadar™: Artificial Intelligence Technologies, Q1 2017* [51], definió los campos de investigación actuales según su propósito:

Campo de investigación	Propósito general
Análisis de imágenes y vídeos	Determinar el contenido de un vídeo o una imagen
Biometría	Facilitar la identificación, verificación y autenticación del ser humano en los sistemas tecnológicos
Reconocimiento de voz y análisis de texto	Transcribir y transformar el habla y la escritura humana en información útil para los sistemas informáticos.

Procesamiento de Lenguaje Natural (NLP)	Extraer los atributos principales que caracterizan un texto.
Plataformas de aprendizaje automático (Machine Learning)	Clasificar información y realizar predicciones de sucesos futuros.
Plataformas de aprendizaje profundo (Deep Learning)	Mismo propósito que *el Machine Learning*, pero a través de un proceso de aprendizaje que "simula" el funcionamiento de las redes neuronales de un ser humano.
Tecnología semántica	Comprender las relaciones, contexto y complejidad de un conjunto de datos.
Hardware optimizado por IA	Mejorar la eficiencia de los componentes *hardware* actuales.
Inteligencia de enjambre	Mejorar la capacidad de las máquinas para la toma de decisiones complejas.
Generación de lenguaje natural	Producir textos narrativos a partir de datos informáticos.
Gestión de decisiones	Asistir a la toma de decisiones de forma más ágil y precisa
Automatización de procesos robóticos	Automatizar tareas repetitivas para mejorar la eficiencia de los procesos de negocio.
Agentes virtuales	Asistir y ofrecer servicios a clientes mediante el uso de personajes ficticios con apariencia humana.

Tabla 3. Campos de investigación de la Inteligencia Artificial

3.2 *Machine Learning*

Uno de los campos de investigación de la Inteligencia Artificial más conocidos es el denominado Aprendizaje Automático (*Machine Learning* en inglés). Esta disciplina de estudio trata de identificar patrones en los datos y aprender de ellos para poder predecir situaciones futuras [52]. Su uso es ideal cuando se desea realizar, entre otras, tareas de regresión y clasificación. Mientras que el concepto de regresión se centra en la predicción de valores numéricos, el de clasificación se centra en la predicción de valores binarios.

El Aprendizaje Automático se categoriza según si el estudio se realiza de forma supervisada o no supervisada, aunque también se contemplan el aprendizaje por refuerzo y, en ocasiones, el aprendizaje semi-supervisado y el *Deep Learning*, entre otros.

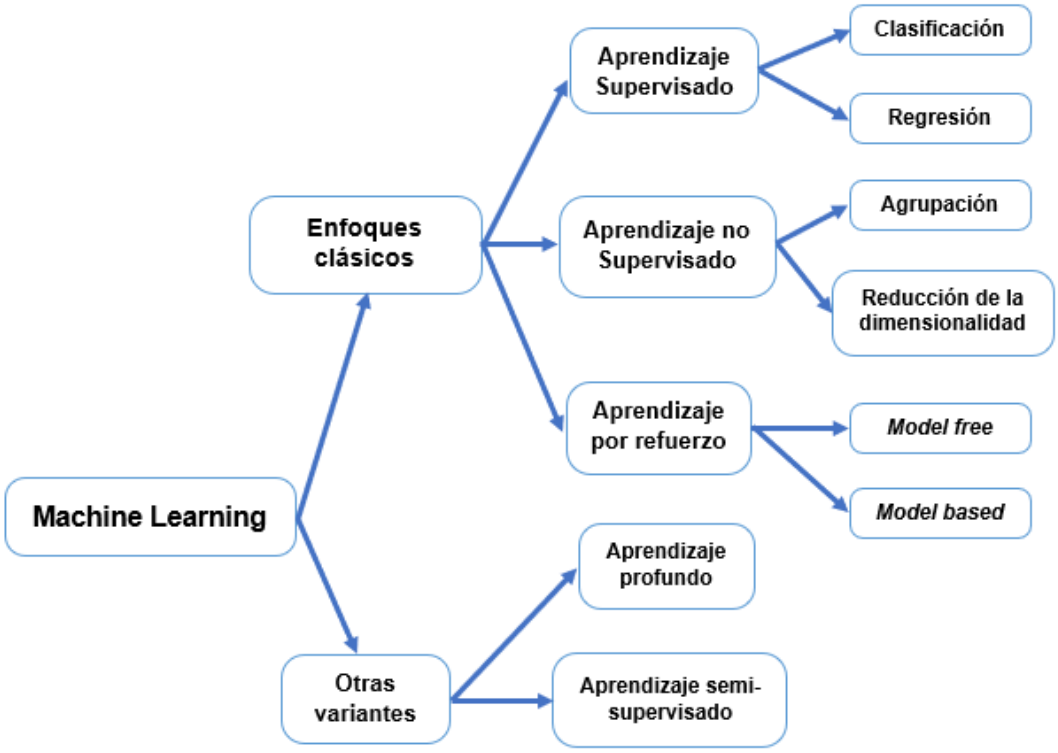

Figura 4. Taxonomía de Machine Learning.

Inspirado en el estudio de M. A. Hossain et al [53]

3.2.1 *Machine Learning* supervisado

El Aprendizaje Automático supervisado [54] implica guiar al algoritmo elegido a través de su proceso de desarrollo. Para ello, el conjunto de datos disponible se divide, por una parte, en datos de entrenamiento y, por otra parte, en datos de prueba. La finalidad del algoritmo es generar un modelo predictivo a partir de los datos de entrenamiento, teniendo en cuenta que los valores de salida son conocidos. Es decir, el algoritmo se basa en una serie de conocimientos previos. Una vez creado el modelo, los datos de prueba son utilizados para verificar la precisión del algoritmo en sus predicciones. La clasificación y la regresión son las técnicas de aprendizaje supervisado.

3.2.1.1 *Machine Learning* supervisado: clasificación

La clasificación [55] es el proceso de aprendizaje de sucesos ocurridos en el pasado, para poder categorizar sucesos futuros similares. A fin de construir un modelo, inicialmente los acontecimientos pasados se agrupan en lo que se conoce como *dataset*. El *dataset* ha de tener una serie de características que describan las muestras que lo componen, además de una etiqueta que indique a qué clase pertenecen.

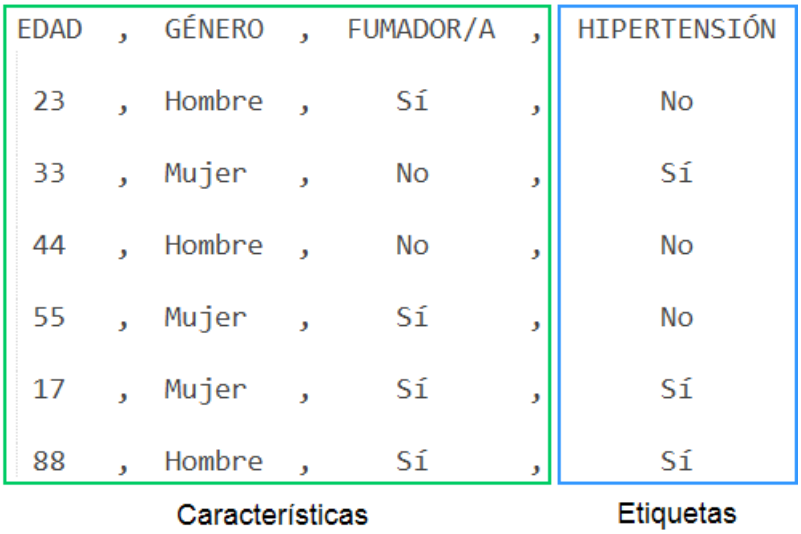

Figura 5. Ejemplo de características y etiquetas en un conjunto de datos.

Si se crease un modelo de aprendizaje sobre todo el conjunto de datos se incurriría en un error de metodología. Es decir, las predicciones del modelo sobre ese mismo conjunto de datos serían todas correctas, con el 100% de precisión, pero no aportaría predicciones certeras en conjuntos de datos nuevos, que es precisamente para lo que se crea un modelo de clasificación. A este problema se le conoce como *overfitting* [56], y como primer paso para evitarlo, se ha de dividir el conjunto de datos en dos partes, una de entrenamiento y otra de *test*.

El conjunto de datos de entrenamiento sirve para crear el modelo de predicción. Por su parte, el conjunto de *test* se emplea para determinar la precisión del modelo, pues son datos no conocidos previamente por el modelo y, por lo tanto, válidos para poder realizar una estimación.

El modelo de predicción se crea a partir de un algoritmo válido para la clasificación de datos. Se presentan a continuación los más relevantes:

- *Naive Bayes* [57]. Este algoritmo de clasificación está basado en teorema de Bayes para determinar la probabilidad condicional, que puede definirse como la posibilidad de que se produzca un suceso, en función de que se produzca un suceso anterior. Sus puntos fuertes son la facilidad de construir un modelo y el buen rendimiento que presenta en conjuntos de datos de gran tamaño. Por el contrario, una de sus grandes limitaciones reside en su característica principal: asumir que las características son independientes entre sí. En el mundo real, pocas veces se encuentran características que no tengan algún tipo de relación.

- Árboles de decisión (*Decision Trees*) [58]. Este algoritmo crea modelos con estructura de árbol. Su lógica de clasificación es aplicar la técnica "si-entonces" para cada característica del conjunto de datos, que se representa en forma de nodo. Por su parte, cada rama representa un valor que explica el sentido de cada nodo. Finalmente, las hojas representan la clasificación final tras evaluar las condiciones anteriores. Este algoritmo, como puntos favorables, tiende a aprender muy bien del conjunto de datos de entrenamiento y es sencillo de construir. Sin embargo, como punto negativo, este algoritmo tiende a realizar un sobreajuste del modelo durante el periodo de entrenamiento.

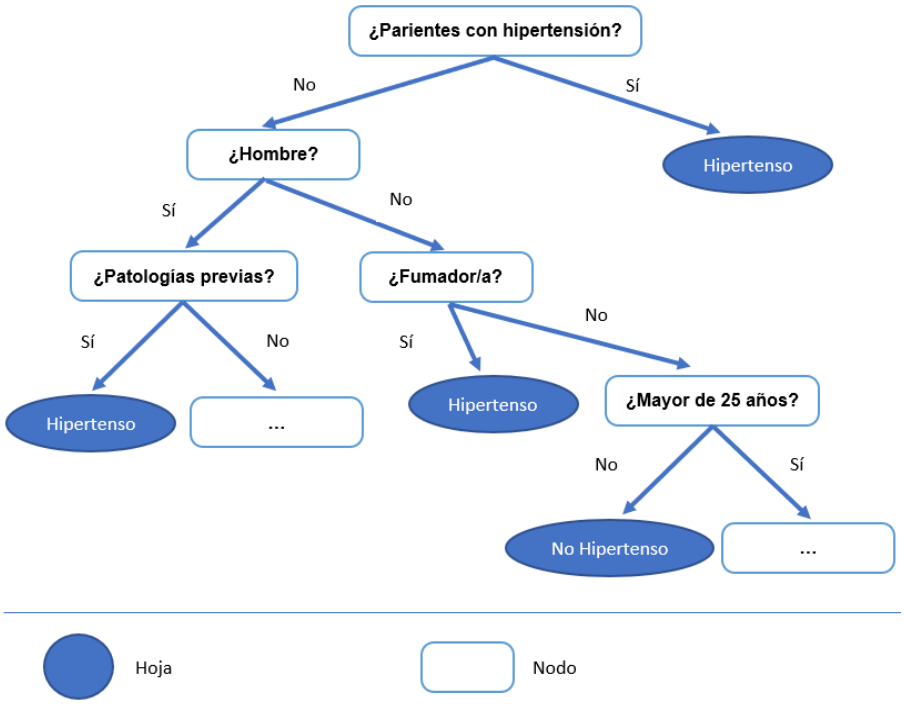

Figura 6. Ejemplo de árbol de decisión para determinar si un sujeto puede ser hipertenso.

- *K-Nearest Neighbors (KNN* por sus siglas en inglés*)* [59]. Este algoritmo almacena todas las instancias del conjunto de datos de entrenamiento en un espacio *n*-dimensional. Cuando recibe una nueva muestra para ser clasificada, posiciona el valor en el plano y analiza la etiqueta de los *k* vecinos más cercanos que tiene a su alrededor, devolviendo como resultado la clase más común. En la Figura 7 se muestra un ejemplo para *k* = 5 y *k* = 9. En el primer caso, la muestra sería catalogada como hipertensa. Sin embargo, en el segundo caso sería catalogada como no hipertensa. El punto fuerte de este algoritmo es que no necesita realizar un periodo de entrenamiento antes de realizar predicciones. Por lo tanto, se le puede añadir nueva información al modelo sin distorsionar su precisión. Como contra, crear modelos a partir de este algoritmo es costoso computacionalmente hablando. Por último, en lo que se refiere a aspectos negativos, este algoritmo es sensible a los *outliers*.

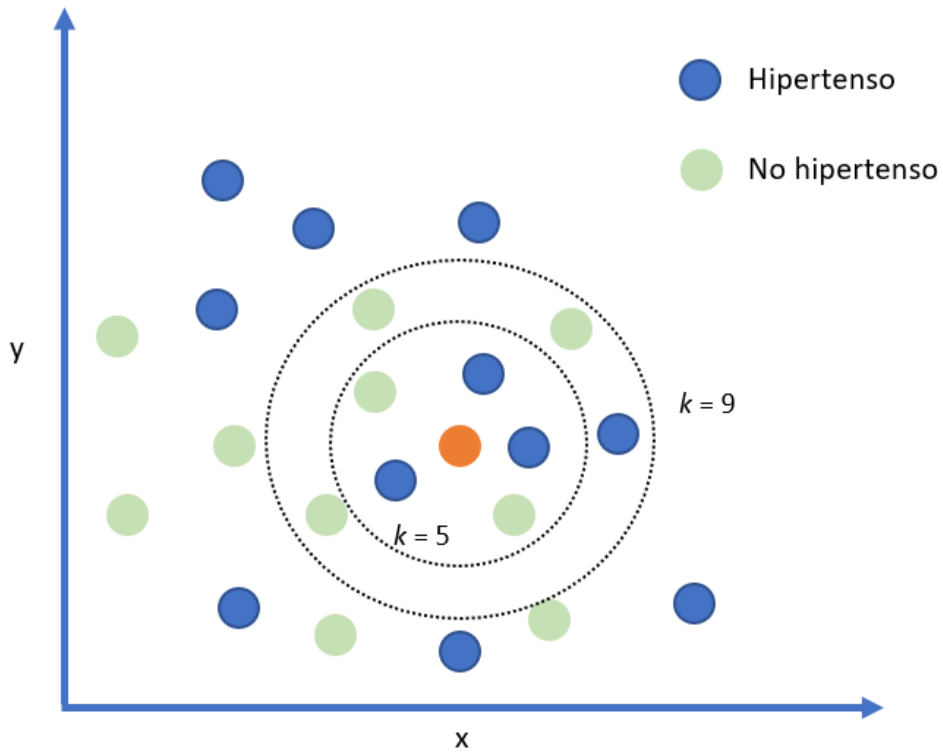

Figura 7. Ejemplo de predicción de una muestra a través de algoritmo k-nearest neighbor.

- *Random Forest* [60]. Este algoritmo crea múltiples árboles de decisión individuales que funcionan como parte del modelo de predicción final.

Cada árbol realiza una predicción de una parte distinta del conjunto de datos de entrenamiento, solucionando los problemas de *overfit* de los árboles de decisión. Como aspecto negativo, el tiempo que necesita para realizar el proceso de entrenamiento es mayor.

- Máquinas de Vectores de Soporte [61] (*SVM* por sus siglas en inglés). Este algoritmo almacena todas las instancias del conjunto de datos de entrenamiento en un hiperplano, con el objetivo de dividir de la mejor forma posible a las clases. El hiperplano se crea con los puntos de cada clase que mejor ayudan a separar el conjunto de datos, conocidos como Vectores Soporte. Cuando el algoritmo recibe una nueva muestra para ser analizada, posiciona el valor en el plano y clasifica según la frontera que le quede más cerca. Como punto fuerte de este algoritmo, se encuentra su efectividad entrenando modelos cuando el número de características es mayor al número de muestras. Sin embargo, como aspecto negativo, el tiempo que necesita para realizar el proceso de entrenamiento es elevado cuando el *dataset* tiene un tamaño considerable. Asimismo, este algoritmo también es propenso, si no se configura correctamente, a realizar un sobreajuste del modelo durante el periodo de entrenamiento. Por último, como aspecto negativo, este algoritmo también es sensible a los *outliers*.

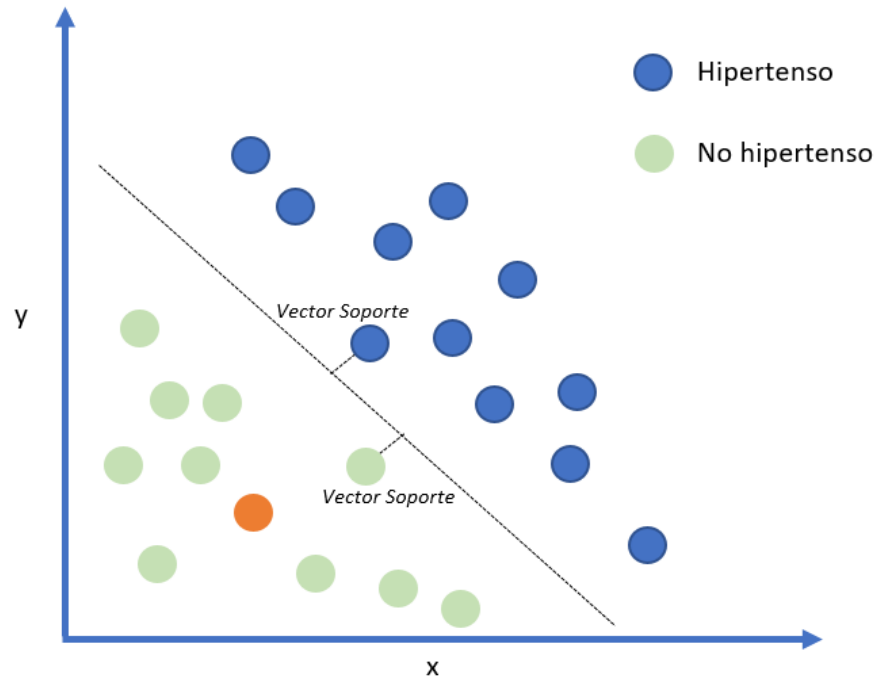

Figura 8. Ejemplo de predicción de una muestra a través de algoritmo SVM.

Finalmente, los resultados del modelo entrenado deben validarse, siendo esta fase una de las más importantes de la clasificación y de las técnicas de Aprendizaje Automático en general, pues revela de forma objetiva la viabilidad o no de aplicar un modelo en el mundo real. Los conceptos relacionados con la validación de un *dataset* son los siguientes:

- Verdadero positivo (*True Positive, TP*): muestra etiquetada como clase objetivo de forma correcta.

- Falso Positivo (*False Positive, TP*): muestra etiquetada como clase objetivo de forma incorrecta.

- Verdadero negativo (*True Negative, TN*): muestra no etiquetada como clase objetivo de forma correcta.

- Falso negativo (*False Negative, FN*): muestra no etiquetada como clase objetivo cuando sí pertenece a clase objetivo.

Las métricas empleadas de forma habitual para validar un modelo de clasificación son los siguientes, oscilando sus valores en el rango comprendido entre 0 y 1, y basando sus cálculos en los conceptos anteriores [62]:

- Exactitud. La exactitud (*accuracy* en inglés) mide el porcentaje de muestras que se ha clasificado correctamente.

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

- Precisión. La precisión compara la tasa de verdaderos positivos, respecto al total de muestras que ha intentado predecir (tanto las correctas como las incorrectas). En otras palabras, mide el porcentaje de casos en los que ha acertado respecto al total. Un valor de precisión elevado indica una tasa baja de falsos positivos.

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

- *Recall*. Esta métrica refleja el porcentaje total de las muestras que han sido correctamente clasificadas, respecto al total de muestras que ha intentado predecir de esa misma clase. Un valor de *recall* elevado indica una tasa baja de falsos negativos.

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

- *F1-score*. Esta métrica se emplea para combinar la precisión y el *recall* en un único valor. Un valor de *F1* elevado indica una buena precisión y robustez del modelo.

$$F1 = 2 * (\frac{Precision * Recall}{Precision + Recall})$$

- Curva *ROC* (*Receiver Operating Characteristic Curve*). Esta métrica de validación se emplea para comparar la tasa de verdaderos positivos respecto a la de falsos positivos.

$$FPR = \frac{FP}{FP + TN}$$

- Área bajo la curva ROC (*ROC AUC*). Esta variante de la Curva ROC permite comparar dos clasificadores de forma más sencilla. Un valor de la Curva ROC AUC elevado indica una mayor fiabilidad del modelo.

3.2.2 *Machine Learning* no supervisado

Se hace uso de las técnicas de Aprendizaje Automático no supervisado [63] cuando es necesario conocer las relaciones que existen entre un conjunto de datos, sin tener a priori conocimiento de estas. Por lo tanto, el objetivo del algoritmo escogido es clasificar el conjunto de datos en diferentes grupos, de acuerdo con las propiedades comunes de los valores de entrada. La agrupación en clústeres y la reducción de la dimensionalidad son las técnicas de aprendizaje no supervisado.

3.2.2.1 *Machine Learning* no supervisado: reducción de la dimensionalidad

En ocasiones, el conjunto de datos a estudiar contiene una cantidad muy elevada de características. Dependiendo del algoritmo escogido para entrenar el modelo, este hecho podría impactar negativamente en el tiempo de entrenamiento (como ocurre, por ejemplo, con las Máquinas de Vectores de Soporte). La reducción de la dimensionalidad se emplea para solventar esta problemática. Se exponen a continuación las técnicas de reducción más habituales:

- Métodos de selección de características [64]. Como su propio nombre indica, está técnica persigue la selección de aquellas características que mejor definen un conjunto de datos, descartando el resto.

- *Manifold Learning* [65]. Esta técnica se emplea para transformar la estructura multidimensional de un conjunto de datos en otra igualmente válida con una menor dimensionalidad.

- Factorización de matrices [66]. Esta técnica se emplea para reducir la dimensionalidad de un conjunto de datos en sus partes constituyentes. Es decir, haciendo uso del álgebra lineal, se selecciona el subconjunto de datos que mejor representa la totalidad de la información. El método más empleado para aplicar esta técnica es el Análisis de Componentes Principales [67] (*PCA* por sus siglas en inglés).

3.2.3 Otros conceptos de interés

La finalidad de este apartado es dar a conocer otros conceptos que están relacionados con el *Machine Learning*, la construcción de los modelos y la mejora de la calidad de estos.

3.2.3.1 *Outliers*

La calidad de un modelo de *Machine Learning* depende en gran medida de la calidad del *dataset*. Un conjunto de datos mal formado puede afectar negativamente el rendimiento del algoritmo empleado, generando en consecuencia unos modelos de Aprendizaje Automático pobres e inválidos.

Se ha introducido con anterioridad cómo los algoritmos *SVM* y *KNN* presentan sensibilidad a los *outliers*. Se conoce como *outlier* a todo punto que se encuentra distante del resto de observaciones de su mismo tipo [68].

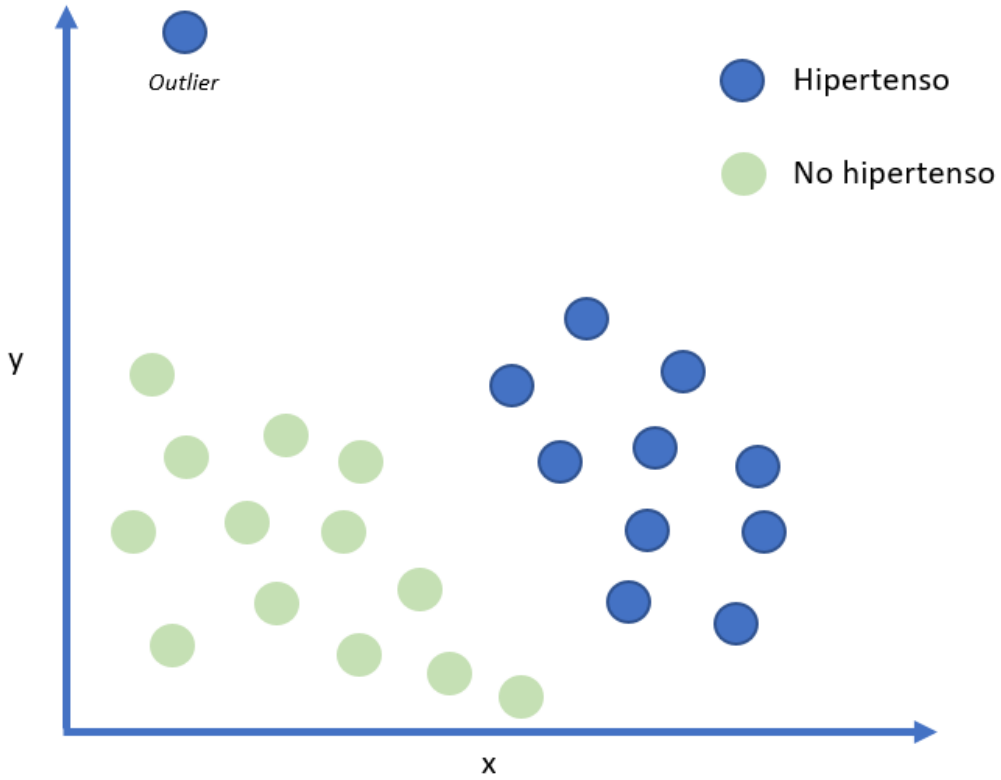

Figura 9. Ejemplo de outlier en un conjunto de datos.

La forma de lidiar con estos puntos anómalos depende en gran medida del conjunto de datos. Sin embargo, las acciones que se realizan de forma habitual son las siguientes:

- Eliminarlos del conjunto de datos. Los *outliers* que no aportan nada al conjunto de datos pueden ser eliminados para mejorar la calidad del modelo.

- Asignarles un nuevo valor. Esta acción se realiza cuando se detecta que a la muestra se le asignó un valor de etiqueta erróneo (error humano), mientras se elaboraba el *dataset*.

- Transformar el *dataset*. A través de la aplicación de conceptos estadísticos se puede transformar un conjunto de datos para mejorar la calidad del *dataset*. A modo de ejemplo, aplicando la función cuantil [69] se puede transformar un *dataset* para que siga, o bien una distribución de probabilidad estándar normal, o bien una distribución uniforme. Este proceso suaviza el impacto de los *outliers* en los modelos de *Machine Learning*.

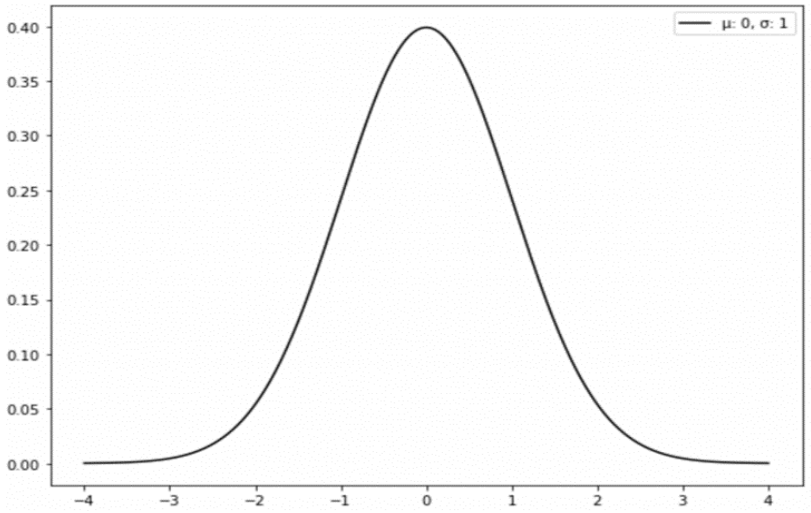

Figura 10. Conjunto de datos con una distribución estándar normal.

Fuente: https://www.statology.org/normal-distribution-vs-standard-normal-distribution/

3.2.3.2 Validación cruzada

Algunos de los algoritmos empleados para crear modelos de Aprendizaje Automático, como los árboles de decisión, son propensos a realizar un sobreajuste del modelo durante el periodo de entrenamiento. La validación cruzada (*cross-validation en inglés*) [70] ayuda a mitigar esta problemática.

Para ello, en primer lugar, el conjunto de datos se divide en *k* conjuntos más pequeños de tamaños similares. A continuación, el algoritmo se entrena haciendo uso de *k - 1* conjuntos de entrenamiento, validando sus resultados con la parte restante del conjunto de entrenamiento. Este proceso se repite *k* veces empleando un grupo distinto como validación en cada una de las iteraciones. Finalmente, del proceso se obtienen *k* resultados de validación, empleando su promedio como estimación final.

Gracias a la validación cruzada, por una parte, se crean modelos que realizan mejores predicciones sobre las muestras del mundo real. Por otra parte, permite hacer un uso más eficiente del *dataset*, ya que todo el conjunto se emplea tanto para el proceso de entrenamiento como para probar la calidad del modelo.

3.2.3.3 *Term frequency - Inverse document frequency*

En ocasiones, algunas características del conjunto de datos están compuestas de textos. En ese caso, requieren de un preprocesado previo para adaptarlas a lo que los modelos de Inteligencia Artificial esperan recibir como parámetros de entrada. Es decir, el texto debe codificarse a valores numéricos a través del proceso conocido como vectorización o extracción de características [71].

Uno de los métodos más conocidos para realizar este proceso es conocido como *Term frequency - Inverse document frequency* (*TF-IDF*) [72]. Por una parte, la frecuencia de término recopila cuántas veces aparece una palabra en un documento. Por otra parte, la frecuencia inversa de los documentos determina si un término es habitual o no en una colección de documentos. En definitiva, este método refleja cómo de importante es una palabra para un documento en un corpus.

3.2.3.4 *One-Hot Encoding*

Cuando se emplea un *dataset*, puede darse la existencia de características categóricas en lugar de numéricas. Si bien es cierto que algunos algoritmos permiten valores categóricos como parámetros de entrada (como los árboles de decisión), otros muchos requieren de una transformación de los valores a representaciones numéricas. Esta representación se puede realizar aplicando *One-Hot Encoding* [73].

A modo de ejemplo, y reutilizando el *dataset* ficticio creado en la Figura 5, los atributos de la característica “Género” son categóricos, admitiendo como válidos los valores “Hombre” y “Mujer”. Tras aplicar *One-Hot Encoding* sobre la característica “Género”, esta desaparece y los atributos “Hombre” y “Mujer” se convierten en nuevas características del *dataset*, indicando en sus respectivas columnas, para cada muestra, un 1 si es “Hombre” y un 0 si es “Mujer” y viceversa.

4. Contexto tecnológico

A lo largo de los últimos años, con el objetivo de mejorar las capacidades defensivas actuales y mitigar las limitaciones existentes, los profesionales de la Ciberseguridad han implementado técnicas de Inteligencia Artificial en los productos de ciberdefensa. De hecho, sus aplicaciones son comúnmente conocidas para la detección y clasificación de *SPAM*, la detección de documentos maliciosos y la detección de anomalías, entre otros [74].

Esta fusión de disciplinas ha resultado exitosa, en parte, porque elimina las limitaciones mencionadas en el Capítulo 2, que no son exclusivas para la detección de *malware*, sino que en esencia afectan también a otras muchas áreas de la Ciberseguridad. Es decir, la Inteligencia Artificial aplicada a la Ciberseguridad permite automatizar tareas que hasta ahora han sido casi siempre artesanales. Asimismo, evita depender de la creación de reglas específicas de forma manual, abarcando un mayor espectro de amenazas, independientemente de si estas son nuevas o si son variantes de las ya conocidas.

En este mismo sentido, con el objetivo de mitigar las limitaciones actuales para el caso específico de la detección de *malware,* desde el sector de la Ciberseguridad se ha comenzado a hacer uso de las técnicas de *Machine Learning* para la detección de *malware*. Múltiples investigadores, como *Firdausi et al.* [75] y *Kruczkowski et al.* [76], evidencian que los algoritmos de clasificación como *K-Nearest Neighbors*, *Naive Bayes*, Árboles de Decisión y *SVM*, entre otros, se muestran efectivos para la detección de binarios maliciosos.

Para crear un modelo de clasificación se necesita disponer de un *dataset de binarios* legítimos y maliciosos*,* compuesto por un conjunto de características que representen a cada muestra, así como una columna de etiquetado que identifique la clase a la que pertenecen. En este sentido, según la literatura actual, las características que representan a un binario son aquellas que pueden extraerse a partir de su análisis estático [77] y/o dinámico [78], según lo estudiado en los puntos 2.2.1 y 2.2.2 de la memoria.

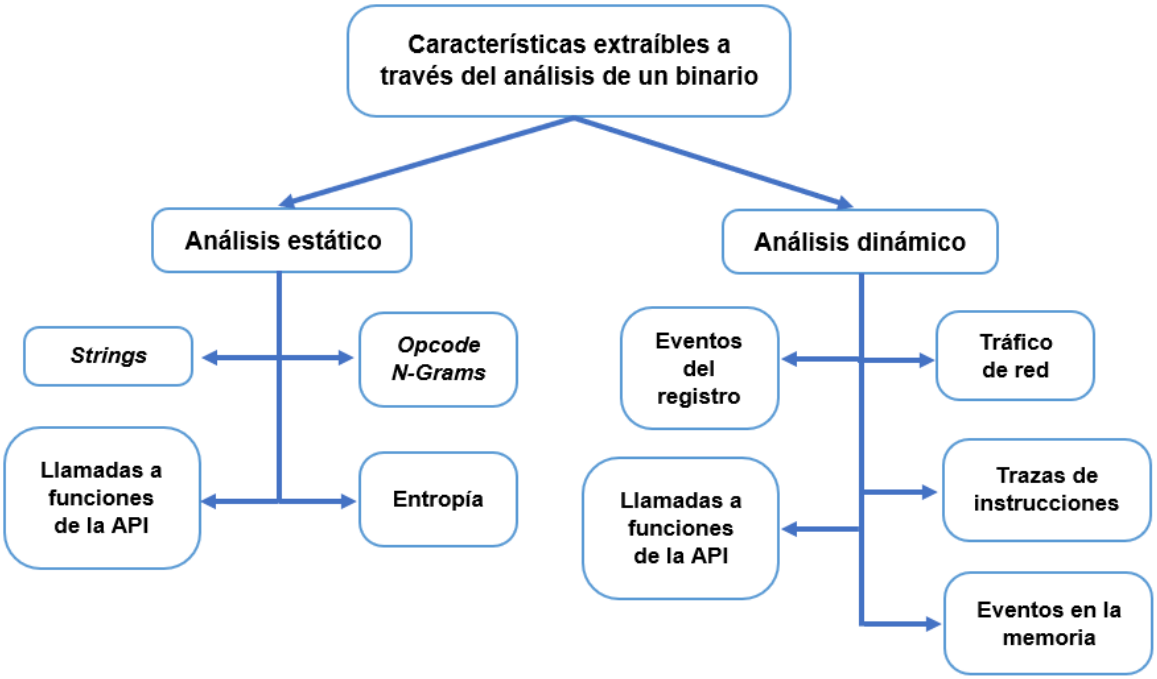

Figura 11. Taxonomía de características a extraer tras analizar un binario, según lo estudiado en los puntos 2.2.1.2 y 2.2.2.1 de la memoria.

Existen distintos autores que han hecho uso de las características obtenidas por los análisis estáticos para entrenar un modelo de clasificación. Por ejemplo, *Schultz et al.* [79] crearon un modelo de clasificación a partir de las características *strings y n-grams,* logrando una precisión del modelo superior al 97%, siendo *Naive Bayes* el algoritmo escogido para entrenar el modelo. *Sami et al.* [80] crearon un modelo de clasificación a partir de la característica de las llamadas a funciones de la API, logrando una precisión del modelo superior al 99%, y haciendo uso del algoritmo *Random Forest* para entrenar el modelo. Por último, *Wojnowicz et al.* [81] desarrollaron de forma eficiente un modelo de clasificación a partir de las características de la entropía y los *strings,* logrando una precisión del modelo superior al 98%.

Sin embargo, no se debe obviar que los análisis estáticos presentan unas limitaciones que podrían invalidar los modelos anteriores en el mundo real *(ver punto 2.2.1.3 de la memoria)*. Es decir, los modelos basados en características estáticas podrían no predecir correctamente los desarrollos de *malware* avanzados que presentasen técnicas de evasión tales como la ofuscación.

Por lo tanto, otros autores consideran necesario emplear las características obtenidas por los análisis dinámicos para entrenar un modelo de clasificación. Por ejemplo, *Bekerman et al.* [82] crearon un modelo de clasificación a partir de la característica del tráfico de red en las capas de Internet, Transporte y Aplicación. Haciendo uso de los algoritmos *Naive Bayes*, Árboles de Decisión y *Random Forest,* lograron un *ROC AUC* medio del modelo superior a 0.966. *Galal et al.* [83] crearon unos modelos de clasificación a partir de la característica de llamadas a la API, mediante diversos algoritmos de clasificación como Árboles de Decisión, *Random Forests* y *SVM*. La precisión lograda para cada algoritmo fue del 97.3%, 97.2% y 92.28%, respectivamente. Por último, *Okane et al.* [84] propusieron un modelo de clasificación basado en el algoritmo *SVM,* escogiendo la característica de trazas de instrucciones, obteniendo unos resultados algo más discretos, con una tasa de detección del 83.41%.

Relativo a la precisión y rendimiento de los modelos, existen dificultades de comparación entre estudios. Esto se debe a que, pese a la existencia de repositorios conocidos de binarios de *malware*, los binarios legítimos suelen estar protegidos por leyes de *copyright* que impiden su difusión. Por lo tanto, cada estudio emplea un *dataset* diferente, de creación propia, con diferentes tamaños de muestras. Asimismo, cada estudio valora de forma distinta qué características dinámicas se van a emplear para la creación del modelo, por lo que genera más variedad de resultados difícilmente comparables.

Estudio	**Algoritmos**	**Características**	**Resultados**
Bekerman et al. [82]	*Naive Bayes*, Árboles de Decisión y *Random Forest*	Tráfico de red	*ROC AUC:* 0.966
Galal et al. [83]	Árboles de Decisión, *Random Forests* y *SVM*	Trazas de llamadas a la API	Precisión: 97.3%, 97.2% y 92.28% respectivamente
Okane et al. [84]	*SVM*	Trazas de instrucciones	Tasa de detección: 83.41%
Radu et al. [85]	*SVM*	Trazas de llamadas al sistema	Precisión: 80%
*Strandlund et al***.** [86]	*Supervised Learning Random Forests*	Trazas de llamadas al sistema y API	Precisión: 94% *TPR*: 0,9 *FPR*: 0,05

Ghiasi et al. [87]	*Supervised Learning Random Forests*	Trazas de llamadas al sistema y API	Precisión: 90% *TPR*: 0,98 *FPR*: 0,01
Mahinthan et al. [88]	*Rotation Random Forest*	Trazas de llamadas al sistema y API	Precisión: 97,6% *FPR*: 0,049
Lindorfer et al. [89]	*SVM*	Trazas de llamadas al sistema, Registro, Procesos/Hilos	Precisión: 99,4% *FPR*: 0,01

Tabla 4. Machine Learning Supervisado para la clasificación de malware. Comparativa de resultados.

5. Solución propuesta

La comunidad ha demostrado que el uso de las técnicas de *Machine Learning* para la detección de *malware* es viable y efectivo. Su uso permite romper con las limitaciones de las reglas manuales, detectando un mayor espectro de amenazas no conocidas previamente. Asimismo, se ha demostrado que las características extraídas de los binarios, a través de los análisis dinámicos, permiten crear unos modelos de clasificación robustos que no pueden conseguirse a través de los análisis estáticos.

Sin embargo, en la literatura no se encuentran proyectos o herramientas de código abierto para el gran público, por lo que, a efectos prácticos, los profesionales de la Ciberseguridad deben seguir creando las reglas de detección de *malware* de forma manual.

Con el objetivo de mitigar las limitaciones de accesibilidad, en el presente Capítulo se propone el desarrollo de un modelo de clasificación que pueda ser accesible por los analistas de *malware*. Es decir, se propone un desarrollo detallado paso a paso, adjuntando el código necesario en el Anexo de este documento, con la idea de que pueda ser empleado como referencia por cualquier profesional interesado. Con este fin, el presente Capítulo se divide en dos fases claramente diferenciadas.

Una primera fase, consistente en la creación y estudio de *dataset* compuesto por binarios tanto legítimos como maliciosos, que se han analizado previamente según las técnicas de análisis dinámico de *malware*.

Una segunda fase, consistente en la creación de múltiples modelos de clasificación basados en técnicas de *Machine Learning* supervisado, con la finalidad de poder detectar nuevas muestras de binarios maliciosos. Para el aprendizaje de los modelos, se ha empleado el *dataset* creado durante la primera fase.

5.1 Conjunto de datos

> En este apartado, como primera fase de desarrollo, se describe el proceso de creación del entorno necesario para la recopilación y análisis de las muestras binarias, además de la posterior selección y extracción de sus características. El objetivo final es la creación de un *dataset* que pueda emplearse para construir un modelo de detección de *malware*.

El desarrollo realizado en esta fase ha sido necesario como consecuencia de la escasez de *dataset* públicos, ya no únicamente de binarios legítimos y maliciosos, sino de conjuntos de datos con las características ya extraídas. En el Capítulo dedicado al contexto tecnológico se hace referencia a esta situación.

5.1.1 Creación de entorno para análisis de *malware*

Con la finalidad de crear un ecosistema favorable para la ejecución y análisis de binarios, se ha propuesto la siguiente solución:

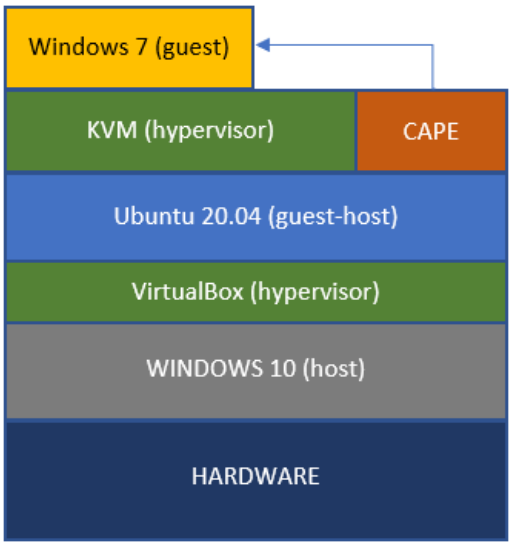

Figura 12. Entorno propuesto para el análisis de malware.

Por seguridad, uno de los requisitos indispensables del entorno precisa que el *malware* se ejecute de forma aislada, sin acceso a la red. Para conseguir ese objetivo, se ha implementado, sobre el host principal, una Máquina Virtual con *Ubuntu* 20.04 [90] en *VirtualBox* [91].

El motivo de emplear el Sistema Operativo *Ubuntu* es por su compatibilidad con los distintos tipos de *sandbox* y lenguajes de programación, además de por su modelo de licencia, que permite su uso de forma gratuita. Por su parte, el motivo de usar *VirtualBox* es tanto por su reconocimiento en el sector de la informática (se trata de un producto de *Oracle*) como por su tipo de licencia, que también permite su uso de forma gratuita.

Tal y como se ha estudiado en el punto 2.2.2.3 de la memoria, en lo que respecta a las limitaciones de los análisis dinámicos, la capacidad del *malware* para detectar entornos virtuales puede invalidar o tergiversar los resultados de estudios posteriores. En este sentido, la identificación del hipervisor de *VirtualBox* por parte del binario malicioso es probable y, por lo tanto, su uso para análisis de *malware* no está recomendado [92]. Como alternativa, se ha hecho uso del hipervisor *kvm* [93] dado que, al ser compatible con *Linux* y no ser habitual para el análisis de *malware* (como si lo es *VirtualBox* o *VMware*), es menos probable que el binario malicioso proceda a ocultar su comportamiento.

A continuación, ha sido necesaria la elección de una herramienta de análisis dinámicos de binarios, como componente principal de la *sandbox*. Existen varias opciones, cada una con sus virtudes y defectos, tal y como se ha visto en la comparativa de la Tabla 2. Finalmente, se ha hecho uso de la *sandbox Cape* por los siguientes motivos:

- Permite el análisis dinámico de los binarios.
- Permite la extracción de multitud de características que definen el comportamiento de un binario.
- Su documentación es extensa.
- Es una herramienta *Open Source*.
- Es una herramienta compatible con *Ubuntu*.
- Permite el análisis de binarios *Windows PE*.
- Está basado en la *sandbox Cuckoo*, por lo que de forma adicional introduce capacidades tales como la extracción de la trazabilidad del registro de *Windows*, ficheros, procesos creados, timeline de ejecución del *malware* y conexión con servidores externos, entre otras.

Una vez seleccionada y configurada la *sandbox*, la ejecución de *malware* se ha realizado en un Sistema Operativo *Windows 7*, a través de las instrucciones enviadas desde *Cape*.

El proceso de análisis de un binario consta de dos fases. En primer lugar, cada vez que se ejecuta un binario, la *sandbox* extrae las características que le definen. En segundo lugar, Cape restaura el estado del Sistema Operativo a un punto anterior a su ejecución.

Gracias a esta configuración se ha logrado un aislamiento doble, pues bajo la capa del *host* principal se encuentran dos Máquinas Virtuales (*Ubuntu* y *Windows 7*), anidadas y aisladas, ejecutándose el *malware* en la última de ellas.

Para conseguir este entorno, se han llevado a cabo los siguientes pasos:

1. Se ha instalado *VirtualBox*, creando una Máquina Virtual *Ubuntu*.

2. Desde *VirtualBox*, se ha configurado en entorno de red de la Máquina Virtual para evitar conexiones desde/hacia el *host* principal (*Windows* 10).

3. Desde VirtualBox, se ha habilitado la opción "*VT-x/AMD-V*". A través de esta funcionalidad se ha podido implementar la máquina Virtual Windows 7 dentro de la Máquina Virtual Ubuntu.

4. Se ha instalado *Python* [94] como lenguaje de programación. Su elección se ha determinado, entre otros, por los siguientes motivos:

    a. La *sandbox Cape* hace uso de este lenguaje de programación para realizar el análisis y la extracción de características del binario.

    b. Sencillez de uso y aprendizaje.

    c. Gran comunidad de soporte.

    d. Multitud de entornos y librerías integradas en este lenguaje de programación. Entre ellas, multitud de librerías y soporte en lo que respecta a técnicas de *Machine Learning*. Por lo tanto, su elección también está fundamentada en las necesidades de la segunda fase de desarrollo.

5. Se ha instalado y configurado *Cape* sobre la Máquina Virtual *Ubuntu* [95].

6. Se ha desplegado una Máquina Virtual *Windows* 7 sobre *Ubuntu*, a través del hipervisor *kvm* [96].

7. Se ha instalado un agente en *Windows* 7 para habilitar las comunicaciones con *Cape* [97].

8. Siguiendo la documentación oficial de *Cape* [98], se han modificado distintos archivos de configuración para adaptar la *sandbox* al entorno de análisis. En esencia, se han cambiado las direcciones IP locales, el nombre del hipervisor (a *kvm*) y el nombre de la máquina virtual donde se va a ejecutar el malware (a *Win7*).

5.1.2 Selección de binarios para el análisis de *malware*

Una vez creado y configurado el entorno, el proceso continúa a través del análisis dinámico de *malware* de un conjunto de binarios, con la finalidad de generar un *dataset* que pueda emplearse en la segunda fase de desarrollo de este proyecto.

Para ello, en primer lugar, se han seleccionado una serie de muestras binarias benignas de la Máquina Virtual *Windows 7*, a través de un script en *PowerShell* [99], que puede visualizarse en el Anexo A.1 de este documento.

En segundo lugar, se han seleccionado una serie de muestras de *malware* del repositorio *TheZoo* [100]. Dos son los motivos principales que han propiciado el uso de este repositorio. El primer motivo es que es un repositorio *Open Source*. El segundo, es que contiene una gran cantidad de muestras de *malware* para *Windows* en formato Portable Executable (PE).

En total se han recopilado 701 muestras, de las cuales 413 son benignas y 288 son maliciosas. Asimismo, los binarios se han etiquetado como 0 y 1, dependiendo si son legítimos o maliciosos, respectivamente.

5.1.3 Análisis de *malware* de los binarios seleccionados y creación de dataset

Una vez recopiladas todas las muestras, estas se han transferido a la *sandbox* con la finalidad de ser analizadas de forma dinámica [101].

Los reportes que ha generado *Cape* se han preprocesado [102] para extraer las características de comportamiento de las muestras analizadas, a partir de las cuales se ha creado el *dataset* de binarios.

Las características del *dataset* están compuestas por información relativa a los eventos del registro, las trazas de instrucciones, las trazas de llamada a la API, y el tráfico de red. Cabe añadir que cada una de las características “generales”, que son las que acaban de ser mencionadas, están compuestas a su vez por una o más características específicas. A modo de ejemplo, la característica general de llamada a la API tiene a su vez características más específicas tales como el número de llamadas totales, el número de llamadas al registro, el número de llamadas al sistema Windows, etc. Por lo tanto, cuando se hace referencia al término características, también se está haciendo referencia a estos “subconjuntos”. El total de características extraídas se encuentra en el Anexo A.2 de este documento.

5.1.4 Tratamiento del conjunto de datos

Una vez creado el *dataset*, se ha detectado que algunas de las características obtenidas no son numéricas. Por lo tanto, requieren de un preprocesado previo para adaptarlas al tipo de datos que muchos de los algoritmos de clasificación de *Machine Learning* esperan recibir como parámetros de entrada. Es decir, deben codificarse a valores numéricos.

Con el objetivo de tratar los datos se ha empleado la librería *Scikit-Learn* [103]. Su elección se ha determinado, entre otros, por los siguientes motivos:

- Sencillez de uso y aprendizaje
- Gran comunidad de soporte.
- Librería *Open Source* con multitud de funcionalidades.
- Su uso es compatible con *Python*.

5.1.4.1 Transformación de la característica de uso del registro

La característica concreta que refleja el uso que hace un binario del registro se encuentra en la columna *registry_usage* del *dataset*, perteneciente al grupo de características de eventos del registro. Su estructura de datos está representada por un diccionario.

```
uso_del_registro_binario_de_ejemplo_ =
    {
        "read_keys":['a','b'],

        "write_keys": ['a','b'],

        "delete_keys": ['a','b','c']
    }
```

Figura 13. Representación de la característica de uso del registro

Para este tipo de datos, se ha decidido aplicar un proceso de transformación mediante el método de *one-hot encoding.* De esta forma, la columna no numérica se ha convertido en un conjunto de columnas numéricas, con valor para cada fila de 1, si existe ese conjunto, o 0, si no existe.

```
#Representación de datos de registry_usage:

uso_del_registro_binario_de_ejemplo =
    {
        "read_keys":['a','b'],

        "write_keys": ['a','b'],

        "delete_keys": ['a','b','c']
    }

uso_del_registro_binario_de_ejemplo_2 =
    {
        "read_keys":['a','b'],

        "write_keys": ['a','b'],

        "delete_keys": ['a','b','d']
    }

#Representación de datos de registry_usage en one-hot:

    [[1. 1. 1. 1. 1. 1. 1. 0.]
     [1. 1. 1. 1. 1. 1. 0. 1.]]
```

Figura 14. Ejemplo de transformación de diccionario a representación one-hot.

5.1.4.2 Transformación de la característica de trazas de llamada a la API

La característica concreta que refleja el conjunto de llamadas a la API que ha realizado un binario, está representada por la columna *api_call_traces* en el *dataset*. Su estructura de datos está representada por un *array* de *strings*.

```
#Representación de datos de api_call_traces:

llamadas_a_la_API_binario_de_ejemplo =

    ['GetSystemTimeAsFileTime', 'NtDelayExecution', 'GetSystemTimeAsFileTime',
    'NtDelayExecution', 'GetSystemTimeAsFileTime', 'NtDelayExecution',
    'GetSystemTimeAsFileTime', 'NtDelayExecution', 'GetSystemTimeAsFileTime',
    'NtDelayExecution', 'GetSystemTimeAsFileTime']

llamadas_a_la_API_binario_de_ejemplo_2 =

    ['FindResourceExW', 'FindResourceExW', 'FindResourceExW', 'FindResourceExW',
    'GetKeyboardLayout', 'NtClose', 'NtClose', 'RegOpenKeyExW', 'LdrLoadDll',
    'LdrGetProcedureAddress', 'LdrGetProcedureAddress', 'LdrGetDllHandle',
    'NtQuerySystemInformation']
```

Figura 15. Representación de la característica de llamadas a la API.

Esta estructura es muy similar a la que puede tener un documento de texto, donde en cada *array* están representadas todas y cada una de las palabras que lo componen. En este caso, las “palabras” están formadas por las funciones de la API que han sido utilizadas por el binario durante su ejecución. En este sentido, se ha estudiado en el punto 3.2.3.3 de la memoria que los documentos en bruto pueden transformarse en una matriz numérica de características *TF-IDF*.

Por lo tanto, se ha decidido llevar a cabo la transformación mencionada, haciendo uso de la función *TfidfVectorizer* [104] de *Scikit-Learn*. El código en cuestión puede verse en el Anexo A.4 de este documento. La función se ha configurado de acuerdo con una serie de parámetros que establecen las directrices de cómo se deben transformar los documentos a la matriz:

- *Max_features*: Se ha limitado el número de características totales que pueden generarse, considerado las 1000 más importantes, ordenadas por la frecuencia de los términos en el *corpus*.

Este parámetro se le ha aplicado para no afectar en exceso al rendimiento de los algoritmos de *Machine Learning*, cuyo tiempo de entrenamiento depende en parte del número de características que tiene un *dataset*.

- *ngram_range*. Se ha considerado únicamente los *unigrams* (1,1) para la creación de la matriz. Es decir, los términos de la matriz constan de una única palabra cada uno.

- *max_df*. Al construir el vocabulario se han ignorado los términos que tienen una frecuencia de documentos superior a 0.9 sobre 1.

5.1.4.3 Transformación de la característica del tráfico de red

La característica concreta que refleja el tráfico de red generado por un binario está representada por la columna *network_traffic* en el *dataset*. Su estructura de datos está formada por un diccionario, que registra en *arrays* de diccionarios las conexiones, dependiendo si son de tipo *TCP* o *UDP*.

```
trafico_de_red_binario_de_ejemplo =
    {
        'tcp':
        [
            {'src': '168.180.85.34', 'sport': 65076,
            'dst': '10.158.113.180', 'dport': 28946,
            'offset': 98, 'time': 2.8953031267945862},

            {'src': '94.21.158.52', 'sport': 26688,
            'dst': '243.245.30.114', 'dport': 46125,
            'offset': 556, 'time': 10.2698546230126984}

        ],
        'udp':
        [
            {'src': '192.168.122.171', 'sport': 54460,
            'dst': '192.168.122.1', 'dport': 53,
            'offset': 24, 'time': 3.5345783698656936},

            {'src': '192.168.122.171', 'sport': 65396,
            'dst': '239.255.255.250', 'dport': 3702,
            'offset': 300, 'time': 5.5612637996673584},
        ]
    }
```

Figura 16. Representación de la característica de tráfico de red.

Para este caso en concreto, se propone un tratamiento para generar 6 nuevas características en el *dataset,* que pueden ser de utilidad para determinar si un binario es o no malicoso. Estas son el número de conexiones sospechosas, número de conexiones *UDP, n*úmero de conexiones *TCP, c*onexiones a puertos conocidos [105] (del 1 al 1023), conexiones a puertos registrados [106] (del 1024 al 49151) y conexiones a puertos efímeros [107] (del 49152 al 65535).

En lo que respecta al número de conexiones sospechosas, el *malware* suele tratar de establecer conexiones de red a través de una serie de puertos concretos, como por ejemplo el 22, 80 y 443, que corresponden a *SSH* (*Secure Shell*), *HTTP* (*Hypertext Transfer Protocol*) y *HTTPS* (*Hypertext Transfer Protocol Secure*) [108] [109]. Por lo tanto, la frecuencia de conexiones a esos puertos debería ser mayor para un binario malicioso que para un binario legítimo.

El código desarrollado para realizar la transformación está disponible en el Anexo A.5 de este documento.

5.1.4.4 Transformación de las características numéricas

El resto de las características del *dataset* ya se encuentran representadas de forma numérica. Sin embargo, cabe la posibilidad de que la distribución de la información se encuentre muy sesgada por su naturaleza, y/o que existan *outliers* que favorezcan esta dispersión.

Por lo tanto, según las posibilidades introducidas en el apartado 3.2.3.1 de la memoria, se ha decidido aplicar a este conjunto de datos un proceso de transformación a través de la función cuantil.

La función cuantil, disponible en *Python* a través de la clase *QuantileTransformer* de *Scikit-Learn* [110], suaviza la relación de los datos dentro de un conjunto, permitiendo que se agrupen siguiendo una distribución de probabilidad estándar normal. El código desarrollado para realizar este proceso se encuentra disponible en el Anexo A.6 de la memoria.

5.1.4.5 Reducción de las características

Como consecuencia de las transformaciones realizadas en los apartados anteriores, el conjunto de datos resultante contiene una cantidad muy elevada de características. En concreto, la transformación realizada sobre característica de uso del registro ha generado 4184 nuevas columnas. Por su parte, la transformación realizada sobre la característica de llamadas a la API ha generado 278 nuevas columnas.

Teniendo en cuenta que este hecho podría impactar negativamente en la eficiencia del modelo, se le ha aplicado al *dataset* una reducción de la dimensionalidad a través de la técnica de factorización de matrices, concretamente a través del algoritmo *PCA*.

El algoritmo *PCA* realiza una transformación lineal sobre la dimensionalidad de los datos. Como resultado, se obtiene un nuevo espacio ortogonal, cuya información maximiza la varianza del conjunto [111]. Uno de los parámetros que debe establecerse, por tanto, es el número de componentes que son necesarios para explicar los datos. Dado que no se puede saber de antemano, pues depende de la propia naturaleza del *dataset,* se ha realizado de forma previa el cálculo para una varianza del 95%, gracias al parámetro *explained_variance_ratio_* de la función *sklearn.decomposition.PCA*, disponible en *Scikit-Learn* [112].

El código desarrollado para realizar este proceso se encuentra disponible en el Anexo A.7 de la memoria.

5.2 Creación de los modelos

En este apartado, como segunda fase de desarrollo, se describe el proceso de creación de los modelos de clasificación para la detección de *malware*, basado en las técnicas de *Machine Learning* supervisado

5.2.1 Creación de entorno para entrenamiento del modelo

El entorno se ha creado sobre el Sistema Operativo *Ubuntu* utilizado durante la fase anterior de desarrollo. Asimismo, se ha escogido *Python* como lenguaje de programación. En ambos casos, los motivos son los mismos que los mentados en el punto anterior.

En lo que respecta al desarrollo del modelo, se ha decidido hacer uso de usado la librería *Scikit-Learn*, por los mismos motivos que los expuestos en el punto 5.1.4 de este documento, sumado a la gran cantidad de algoritmos de *Machine Learning* disponibles.

5.2.2 Selección de características para entrenamiento del modelo

En la literatura no siempre se hace uso de todas las características que definen el comportamiento de un binario. Se ha visto como en algunos estudios se han entrenado modelos haciendo uso únicamente de la característica de llamadas a la API, en otros únicamente se ha hecho uso de la característica del tráfico de red, en otros se ha empleado una combinación de varias características, etc.

En este proyecto, con la finalidad de poder crear un modelo donde se puedan escoger libremente las características, el *dataset* se ha desglosado según el propósito general que caracteriza cada una de las columnas.

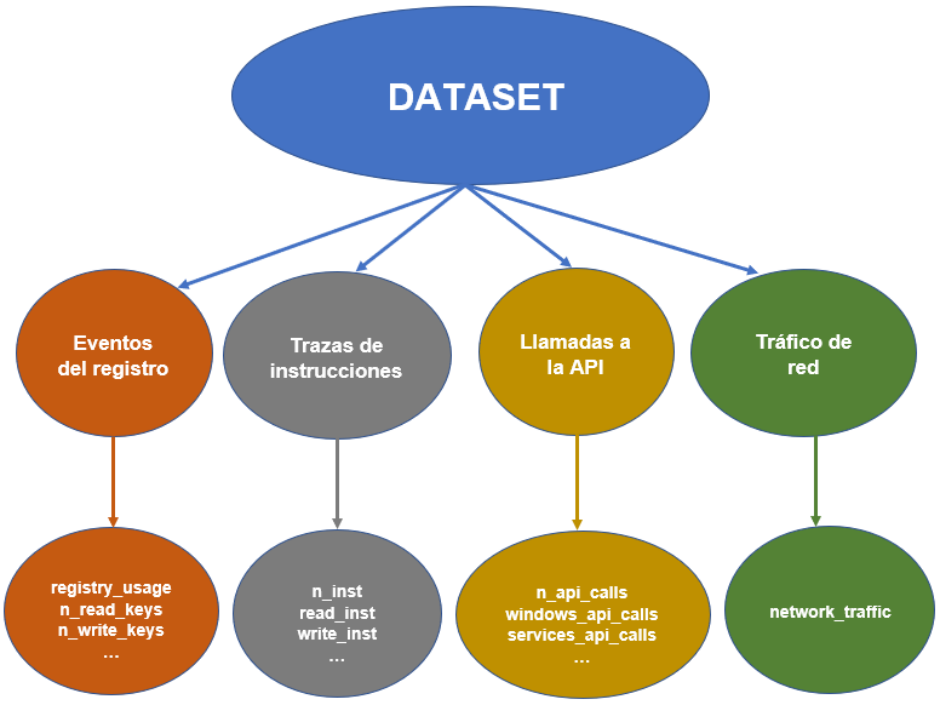

Figura 17. Desglose del dataset según propósito de las características.

De esta forma, se puede personalizar de forma sencilla las características que se van a emplear para construir el modelo de entrenamiento. El código desarrollado para realizar este proceso se encuentra disponible en el Anexo A.8 de la memoria.

En este proyecto se ha propuesto el uso de distintas combinaciones de características para comprobar la eficacia del modelo de detección en cada caso. En concreto, las combinaciones son las siguientes:

Identificador	Características seleccionadas
C1	*Eventos del registro*
C2	*Trazas de instrucciones*
C3	*Llamadas a la API*
C4	*Tráfico de red*
C5	*Eventos del registro, Trazas de instrucciones*
C6	*Eventos del registro, Llamadas a la API*
C7	*Eventos del registro, Tráfico de red*
C8	*Trazas de instrucciones, Llamadas a la API*
C9	*Llamadas a la API, Tráfico de red*
C10	*Eventos del registro, Trazas de instrucciones, Llamadas a la API, Tráfico de red*

Tabla 5. Combinaciones de características propuestas para la clasificación de malware.

5.2.3 Selección y configuración de algoritmos de clasificación

Del mismo modo que ocurre con la selección de características, en la literatura también existen distintas combinaciones en lo que se refiere a algoritmos empleados para entrenar los modelos de detección de *malware*. Se ha visto como en algunos estudios se ha empleado el algoritmo *SVM*, en otros el algoritmo Random Forest, en otros una combinación de *Naive Bayes*, *Árboles de Decisión* y *Random Forest*, etc.

En este proyecto, con la finalidad de analizar la efectividad de distintos modelos, se ha hecho uso de múltiples algoritmos de clasificación para entrenar los conjuntos de datos. Estos son: *KNN*, *SVM*, Árboles de Decisión, *Random Forest* y *Naive Bayes*.

Asimismo, algunos de estos algoritmos se han parametrizado de forma distinta para añadir más variabilidad al estudio. En concreto, se han propuesto las siguientes combinaciones:

Identificador	Configuración propuesta
A1	*Algoritmo: KNN* *Parámetros: leaf_size: 1, n_neighbors: 2, weights: 'distance'*
A2	*Algoritmo: KNN* *Parámetros: leaf_size: 1, n_neighbors: 3, weights: 'distance'*
A3	*Algoritmo: KNN* *Parámetros: leaf_size: 1, n_neighbors: raíz cuadrada del número de muestras totales, weights: 'distance'*
A4	*Algoritmo: SVM* *Parámetros: C: 0.25*
A5	*Algoritmo: SVM* *Parámetros: C: 1*
A6	*Algoritmo: SVM* *Parámetros: C: 2*
A7	*Algoritmo: Árboles de Decisión* *Parámetros: max_depth: 3, random_state: 42*
A8	*Algoritmo: Árboles de Decisión* *Parámetros: max_depth: 10, min_samples_leaf: 2*
A9	*Algoritmo: Árboles de Decisión* *Parámetros: max_depth: 10, random_state: 42*
A10	*Algoritmo: Random Forests* *Parámetros: max_depth: 10, random_state: 42*
A11	*Algoritmo: Random Forests* *Parámetros: max_depth: 50, random_state: 42*
A12	*Algoritmo: Random Forests* *Parámetros: max_depth: 100, random_state: 42*
A13	*Algoritmo: Naive Bayes* *Parámetros: Por defecto*

Tabla 6. Algoritmos propuestos para la clasificación de malware, junto a sus parámetros de configuración.

5.2.4 Resultados

El entrenamiento de los modelos se ha realizado a través del desarrollo del código ubicado en el Anexo A.9 de la memoria. El número de modelos entrenados, equivalente a 130, es el resultado de las combinaciones propuestas en las Tablas 5 y 6 de esta memoria. Asimismo, la selección arbitraria de características ha sido posible gracias al desarrollo realizado en el punto 5.2.2 de la memoria, ejemplificado a través de la Figura 17 del mismo apartado.

5.2.4.1 Validación de los modelos

Con el objetivo de evitar un sobreajuste de los modelos durante el periodo de entrenamiento, según lo estudiado en el apartado 3.2.3.2 de la memoria, se ha hecho uso de la validación cruzada con 10 iteraciones sobre el conjunto de datos para cada modelo. El código desarrollado se encuentra también en el Anexo A.9 de la memoria.

5.2.4.2 Validación de los resultados

La validación de los resultados para cada modelo se ha realizado conforme a las métricas *accuracy*, *ROC AUC*, F1 y *recall*, expuestas en el punto 3.2.1.1 de la memoria. Se muestran a continuación los resultados validación de algunos modelos relevantes. Los resultados de validación para todos y cada uno de los modelos que se han creado se encuentran en el Anexo A.10 de la memoria.

Modelo	***Accuracy***	***ROC AUC***	***F1***	***Recall***
C7 – A4	*0.975*	*0.977*	*0.976*	*0.964*
C10 – A4	*0.965*	*0.970*	*0.964*	*0.971*
C4 – A6	*0.956*	*0.960*	*0.956*	*0.924*
C4 – A10	*0.968*	*0.968*	*0.961*	*0.941*
C9 – A10	*0.935*	*0.937*	*0.930*	*0.943*
C1 – A13	*0.648*	*0.669*	*0.709*	*0.835*

Tabla 7. Validación de los resultados de algunos de los modelos más relevantes.

La Tabla anterior muestra unos valores de validación muy positivos para gran parte de los modelos de detección entrenados.

A través de la métrica *accuracy* se observa que, en el mejor de los casos, el porcentaje de muestras (tanto maliciosas como legítimas) que se han clasificado correctamente es del 97,5%.

Por su parte, a través de la métrica de *recall* se observa que, en el mejor de los casos, las muestras clasificadas como *malware* realmente eran *malware* en el 97,1% de las ocasiones.

En lo que respecta a la métrica F1, se observa un balance muy positivo entre la precisión y el *recall* de los modelos, lo que indica que son modelos robustos.

Por último, los resultados obtenidos en la métrica ROC AUC indican que el modelo es capaz de distinguir muy bien entre las clases legítimas y las maliciosas. Es decir, los modelos tienen buena capacidad para clasificar las muestras de *malware* como tal, y lo mismo con las legítimas.

5.2.4.3 Discusión

A la vista de los resultados obtenidos, se concluye que la clasificación de *malware* mediante modelos de *Machine Learning* es viable y efectiva. Asimismo, teniendo en cuenta los resultados de validación de todas las combinaciones propuestas, se puede concluir que los mejores algoritmos de clasificación para este *dataset* son *SVM* y *Random Forest*.

SVM es el algoritmo que ha obtenido los mejores resultados cuando su valor de *C* es 0.25. Los resultados con este algoritmo muestran un *ROC AUC* del 97,7%, siendo este valor el más alto de los 130 modelos que se han creado y validado. Por su parte, el algoritmo *Random Forest* es el algoritmo más constante en la obtención de buenos resultados. Del *top* 10 de mejores modelos según los resultados, en 8 de ellos se ha empleado este algoritmo.

Como contrapartida, el algoritmo *Naive Bayes* se ha posicionado como el peor clasificador de *malware* en este proyecto. El hecho de que este algoritmo asuma que las características son independientes entre sí, le ha penalizado en el análisis de un *dataset* donde las relaciones entre características sí importan. Cuando se da la ejecución de *malware* en un sistema informático, son diversas las acciones maliciosas que lleva a cabo, por lo que es muy posible encontrar trazas distintas en diversos lugares del sistema (conexiones de red, acceso al registro, trazas en la memoria, etc.). Es decir, todo está relacionado durante la ejecución de *malware*.

Prueba también de la importancia de las relaciones entre características, es que los mejores modelos han sido aquellos donde se han combinado distintas características. En concreto, la relación de características que mejores resultados ha obtenido tiene que ver con los eventos del registro y el tráfico de red, obteniendo un *ROC AUC* del 97,7%. Por su parte, la combinación de todas las características disponibles también ha ofrecido muy buenos resultados, con un *ROC AUC* del 97%.

Por último, la característica que mejores resultados ha obtenido en solitario ha sido la del tráfico de red, alcanzando un *ROC AUC* del 96,8% cuando ha empleado el algoritmo de clasificación *Random Forest*. Los buenos resultados podrían deberse a la transformación realizada de esta característica en el punto 5.1.4.3 de la memoria. El hecho de haber tenido en consideración los puertos que emplea habitualmente el *malware* en sus conexiones, ha podido tener un papel fundamental en estos resultados. Por lo tanto, uno de los aspectos a tener en cuenta tendría que ver con la transformación de características. Se debe invertir más tiempo en este proceso, ideando estrategias basadas en la Ciberseguridad que puedan explicar mejor el sentido de los datos.

6. Conclusiones

El objetivo principal del proyecto se ha logrado de forma satisfactoria, dado que a través del desarrollo de la solución propuesta se han generado modelos efectivos de *Machine Learning* capaces de detectar muestras de *malware* no conocidas previamente. Gracias al amplio abanico de combinaciones que se ha propuesto, sumado al trabajo previo de creación del *dataset* y transformación de algunas de sus características como idea novedosa, ha sido posible encontrar un modelo de clasificación óptimo para la detección de *malware,* obteniendo métricas objetivas muy prometedoras con un *accuracy* del 97,5%, un *ROC AUC* del 97,7%, un *F1* del 97,6% y un *recall* del 96,4%. Asimismo, se pone en valor la transparencia mostrada durante todo el proceso de desarrollo, donde no únicamente se han hecho visibles los resultados obtenidos, sino que se han aportado detalles y código de todo el proceso, que pueden servir como referencia a los profesionales de la Ciberseguridad para crear sus propios modelos de detección.

Cabe destacar que a través del desarrollo de este proyecto se han podido combinar de forma satisfactoria dos áreas distintas de la Ingeniería Informática, en concreto las áreas de Inteligencia Artificial y Ciberseguridad. En lo que respecta al área de la Inteligencia Artificial, se han adquirido las competencias necesarias en el campo de estudio del *Machine Learning* para la creación de modelos efectivos de clasificación de muestras binarias*.* En lo que concierne al área de Ciberseguridad, se han adquirido los conocimientos necesarios para poder analizar muestras de *malware* de forma dinámica, así como también los conocimientos necesarios para crear el *dataset* de binarios maliciosos y legítimos.

En referencia al párrafo anterior, el hecho de haber abordado 2 disciplinas diferentes de la Ingeniería Informática ha supuesto una dedicación compartida en tiempo que ha impedido profundizar en algunos aspectos teóricos, especialmente en lo que respecta al análisis de *malware* y a los algoritmos y parámetros escogidos durante la fase de desarrollo de los modelos de *Machine Learning*. Asimismo, la necesidad de haber tenido que crear un *dataset* desde cero ha condicionado la planificación temporal propuesta al comienzo del proyecto, ya que en un principio se esperaba que fuera posible disponer de un *dataset* público de muestras de binarios legítimos y maliciosos.

Como consecuencia de lo anterior, se ha tenido que realizar un esfuerzo adicional para adquirir las competencias en materia de análisis dinámico de *malware*, ya que ha sido un paso necesario para la creación de un *dataset* propio. Por lo tanto, la planificación temporal al respecto también ha tenido que adaptarse, paralelizando tareas que en principio estaban propuestas como secuenciales. Por lo tanto, se propone como trabajo futuro la ampliación del *dataset* con un mayor número de muestras tanto maliciosas como legítimas, así como aplicar sobre el conjunto de datos técnicas de *Deep Learning* en lugar de las técnicas de *Machine Learni*ng tradicionales, para valorar si los modelos que pudieran obtenerse son más eficaces o no.

7. Glosario

***Accuracy**:* métrica que mide el porcentaje de muestras que se ha clasificado correctamente.

Antivirus: programa de detección y protección contra el *malware*.

Área bajo la curva *ROC* (*ROC AUC*): variante de la métrica Curva ROC, cuyo uso permite comparar dos clasificadores de forma más sencilla.

Corpus: conjunto de documentos de un mismo tipo.

Curva *ROC* (*Receiver Operating Characteristic Curve*): métrica de validación que se emplea para comparar la tasa de verdaderos positivos respecto a la de falsos positivos.

***Debugger**:* programa que facilita la ejecución paso a paso de un código fuente.

***Decompiler**:* programa que transforma el lenguaje máquina en lenguaje de alto nivel.

***Disassembler**:* programa que transforma el lenguaje máquina en lenguaje de bajo nivel.

Falso negativo (*False Negative, FN*): muestra no etiquetada como clase objetivo cuando sí pertenece a clase objetivo.

Falso Positivo (*False Positive, TP*): muestra etiquetada como clase objetivo de forma incorrecta.

Inteligencia Artificial: característica que se le aplica a las máquinas que tienen la capacidad de resolver una tarea de forma “inteligente”.

Interfaces de Programación de Aplicaciones (API): bibliotecas de funciones predeterminadas de *Windows.*

Machine Learning: disciplina de estudio que trata de identificar patrones en los datos y aprender de ellos para poder predecir situaciones futuras.

Malware: *software* diseñado y desarrollado con el propósito de robar, destruir o modificar la información de los sistemas informáticos. También se emplea el término binario malicioso para hacer referencia a *malware*.

Máquina Virtual: *software* que permite instalar un Sistema Operativo dentro de otro sistema Operativo existente.

Metamorfismo: capacidad del *malware* de reprogramarse a sí mismo.

Ofuscación: técnica que modifica el código fuente de un binario para hacerlo incomprensible.

Opcode n-grams*:* conjunto de secuencias de operaciones de bajo nivel que se extraen del código fuente de un binario.

Outlier: observación anómala y extremada de una muestra.

Overfitting: situación que ocurre durante aprendizaje de un modelo predictivo, en el que se tiene en cuenta demasiada información del *dataset*.

Polimorfismo: capacidad del *malware* para cambiar su estructura de código cada vez que se ejecuta una copia de este.

Precisión: métrica que compara la tasa de verdaderos positivos, respecto al total de muestras que ha intentado predecir.

Recall: métrica que refleja el porcentaje total de las muestras que han sido correctamente clasificadas, respecto al total de muestras que ha intentado predecir de esa misma clase.

Sandbox: entorno virtual seguro preparado para ejecutar *malware* sin infectar otros sistemas informáticos.

TCP: protocolo de transmisión de información que proporciona un transporte fiable de los datos.

UDP: protocolo de transmisión de información que proporciona un transporte sin conexión de los datos.

Validación cruzada: técnica de entrenamiento de un modelo predictivo que reduce las posibilidades de sobreentrenar un modelo.

Verdadero negativo (*True Negative, TN*): muestra no etiquetada como clase objetivo de forma correcta.

Verdadero positivo (*True Positive, TP*): muestra etiquetada como clase objetivo de forma correcta.

8. Bibliografía

[1] S. Morgan, «Global Ransomware Damage Costs Predicted To Reach $20 Billion (USD) By 2021,» 2019. [En línea]. Available: https://cybersecurityventures.com/global-ransomware-damage-costs-predicted-to-reach-20-billion-usd-by-2021/. [Último acceso: 12 09 2021].

[2] R. Tahir, «A study on malware and malware detection techniques,» *International Journal of Education and Management Engineering, 8(2), 20,* p. 20, 2018.

[3] D. Craigen, N. Diakun-Thibault y R. Purse, «Defining Cybersecurity,» *Technology Innovation Management Review ,* 2014.

[4] A. Chavan, K. Kerakalamatti y S. S., «Implementation of Portable Antivirus System using Signature-based Detection and Heuristic Analysis,» *5th International Conference on Trends in Electronics and Informatics (ICOEI),* p. 1481, 2021.

[5] (ISC)2, «A Resilient Cybersecurity Profession Charts the Path Forward,» (ISC)2 CYBERSECURITY WORKFORCE STUDY, 2021.

[6] E. Costa, «Artificial Intelligence in Cybersecurity: The Benefits and Challenges.,» CENGN, 2021. [En línea]. Available: https://www.cengn.ca/information-centre/innovation/artificial-intelligence-in-cybersecurity-the-benefits-and-challenges/. [Último acceso: 1 10 2021].

[7] L. Gil, «AI brings speed to security.,» O'Reilly Media, 2019. [En línea]. Available: https://www.oreilly.com/content/ai-brings-speed-to-security/. [Último acceso: 1 10 2021].

[8] X. Hongfa, S. Shaowen, V. Guru y L. Tian, «Machine Learning-Based Analysis of Program Binaries: A Comprehensive Study.,» *IEEE,* p. 10.1109, 2019.

[9] D. Gibert, C. Mateu y J. Planes, «The rise of machine learning for detection and classification of malware: Research developments, trends

and challenges.,» *Journal of Network and Computer Applications 153,* nº 1084- 8045, p. 102526, 2020.

[10] Q. Chen y R. A. Bridges, «Automated Behavioral Analysis of Malware: A Case Study of WannaCry Ransomware.,» *IEEE,* pp. 454-460, 2017.

[11] «Market share held by the leading computer (desktop/tablet/console) operating systems worldwide from January 2012 to September 2021,» Statista, 2021. [En línea]. Available: https://www.statista.com/statistics/268237/global-market-share-held-by-operating-systems-since-2009/. [Último acceso: 10 10 2021].

[12] M. Pietrek, «An in-depth look into the Win32 portable executable file format, part 2,» *MSDN Magazine,* 2002.

[13] B. Pranggono y A. Arabo, «COVID-19 pandemic cybersecurity issues,» *Internet Technology Letters,* vol. 4(2), nº e247, 2021.

[14] E. Kirda, C. Kruegel, G. Banks, G. Vigna y R. Kemmerer, «Behavior-based Spyware Detection,» *Usenix Security Symposium,* p. 694, 2006.

[15] P. Tuli y P. Sahu, «System monitoring and security using keylogger,» *International Journal of Computer Science and Mobile Computing,* vol. 2, nº 3, pp. 106-111, 2013.

[16] Fortinet, «Trojan Horse Virus,» [En línea]. Available: https://www.fortinet.com/resources/cyberglossary/trojan-horse-virus. [Último acceso: 03 12 2021].

[17] E. Florio, «When malware meets rootkits,» *Virus Bulletin,* vol. 12, 2005.

[18] S. Aurangzeb, M. Aleem, M. A. Iqbal y M. A. Islam, «Ransomware: a survey and trends,» *Journal of Information Assurance & Security,* vol. 6, nº 2, pp. 48-58, 2017.

[19] R. Saranya, S. S. Kannan y N. Prathap, «A survey for restricting the DDOS traffic flooding and worm attacks in Internet,» *International Conference on Applied and Theoretical Computing and Communication Technology ,* nº 251-256, 2015 .

[20] M. Christodorescu y S. Jha, «Static analysis of executables to detect malicious patterns,» *Proceeding of USENIX Security Symposium,* nº 169-186, 2003.

[21] M. Nar, A. G. Kakisim, M. N. Yavuz y İ. Soğukpinar, «Analysis and Comparison of Disassemblers for OpCode Based Malware Analysis,» *International Conference on Computer Science and Engineering,* nº 17-22, 2019 .

[22] E. F. Nordmark, «The fundamentals of debuggers and the challenges of Reverse Debugging,» *NTNU,* 2019.

[23] A. Gussoni, A. Di Federico, P. Fezzardi y G. Agosta, «A comb for decompiled C code,» *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security,* pp. 637-651, 2020.

[24] D. Konopiský, «Malware detection in applications based on presence of computer generated strings». Patente U.S. Patent Application No. 15/942,129., 4 Oct 2018.

[25] F. Z. y T. Z., «Malware detection and classification based on n-grams attribute similarity.,» *IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC),* vol. 1, pp. 793-796, 2017.

[26] M. Alazab, S. Venkataraman y P. Watters, «Towards understanding malware behaviour by the extraction of API calls,» *Second cybercrime and trustworthy computing workshop, IEEE,* pp. 52-59, 2010.

[27] L. R. y H. J, «Using entropy analysis to find encrypted and packed malware.,» *IEEE Security Privacy,* vol. 5, nº 2, pp. 40-45, 2007.

[28] A. Moser, C. Kruegel y E. Kirda, «Limits of static analysis for malware detection.,» *In Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007),* pp. 421-430, 2007.

[29] I. You y K. Yim, «Malware obfuscation techniques: A brief survey,» *International conference on broadband, wireless computing, communication and applications ,* pp. 297-300, 2010.

[30] C. Willems, T. Holz y F. Freiling, «Toward automated dynamic malware analysis using cwsandbox,» *EEE Security & Privacy,* vol. 5, nº 2, pp. 32-39, 2007.

[31] M. Ghiasi, S. A. y S. Z., «Dynamic malware detection using registers values set analysis,» *9th International ISC Conference on Information Security and Cryptology,* pp. 54-59, 2012.

[32] G. Zhao, K. Xu, L. Xu y B. Wu, «Detecting APT malware infections based on malicious DNS and traffic analysis. IEEE access,» vol. 3, pp. 1132-1142, 2015.

[33] B. Anderson, D. Quist, J. Neil, C. Storlie y T. Lane, «Graph-based malware detection using dynamic analysis.,» *Journal in computer Virology,* vol. 7, nº 4, pp. 247-258, 2011.

[34] T. Teller y A. & Hayon, «Enhancing automated malware analysis machines with memory analysis,» *Black Hat USA,* 2014.

[35] P. Arntz, «Sandbox in security: what is it, and how it relates to malware,» Malwarebytes Labs, 24 September 2020. [En línea]. Available: https://blog.malwarebytes.com/awareness/2020/09/sandbox-in-security/. [Último acceso: 04 12 2021].

[36] K. O'Reilly, «Cape Sandbox,» 06 2018. [En línea]. Available: https://github.com/kevoreilly/CAPEv2.

[37] C. Guarnieri, «Cuckoo 2.0.7 Automated Malware Analysis,» 06 2019. [En línea]. Available: https://cuckoosandbox.org/.

[38] C. Polska, «DRAKVUF Sandbox,» 06 2020. [En línea]. Available: https://github.com/CERT-Polska/drakvuf-sandbox.

[39] «Joe Sandbox,» Joe Security, [En línea]. Available: https://www.joesandbox.com/.

[40] «SEE (Sandboxed Execution Environment),» F-Secure, 2015. [En línea]. Available: https://github.com/F-Secure/see.

[41] «Virtualization/Sandbox Evasion,» MITRE ATT&CK, 19 April 2019. [En línea]. Available: https://attack.mitre.org/techniques/T1497/. [Último acceso: 05 12 2021].

[42] M. Al-Asli y T. A. Ghaleb, «Review of signature-based techniques in antivirus products,» *International Conference on Computer and Information Sciences (ICCIS),* pp. 1-6, 2019 .

[43] Purplesec, «2021 Cyber Security Statistics - The Ultimate List Of Stats, Data & Trends,» Purplesec, 2021. [En línea]. Available: https://purplesec.us/resources/cyber-security-statistics/. [Último acceso: 03 12 2021].

[44] P. O'Kane, S. Sezer y K. McLaughlin, «Obfuscation: The hidden malware,» *IEEE Security & Privacy,* vol. 9, nº 5, pp. 41-47, 2011.

[45] D. Baysa, R. M. Low y M. Stamp, «Structural entropy and metamorphic malware,» *Journal of computer virology and hacking techniques,* vol. 9, nº 4, pp. 179-192, 2013.

[46] Z. Bazrafshan, H. Hashemi, S. M. H. Fard y A. Hamzeh, «A survey on heuristic malware detection techniques,» *The 5th Conference on Information and Knowledge Technology,* nº 113-120, 2013.

[47] WatchGuard's Threat Lab, «Internet Security Report - Q1 2021,» 2021. [En línea]. Available: https://www.watchguard.com/wgrd-resource-center/security-report-q1-2021. [Último acceso: 14 12 2021].

[48] WatchGuard , «New WatchGuard Research Reveals Traditional Anti-Malware Solutions Miss Nearly 75% of Threats,» 24 06 2021. [En línea]. Available: https://www.watchguard.com/wgrd-news/press-releases/new-watchguard-research-reveals-traditional-anti-malware-solutions-miss. [Último acceso: 15 12 2021].

[49] «Tech giants investing in artificial intelligence,» Techworld, [En línea]. Available: https://www.techworld.com/picture-gallery/data/tech-giants-investing-in-artificial-intelligence-3629737/. [Último acceso: 10 12 2021].

[50] J. McCarthy, M. L. Minsky, N. Rochester y C. E. Shannon, «A proposal for the Dartmouth summer research project on artificial intelligence,» 1955.

[51] R. Curran y B. Purcell, «TechRadar™: Artificial Intelligence Technologies, Q1 2017,» Forrester, 2017.

[52] C. Torrano Giménez, P. Recuero, F. Ramirez, S. Hernández y J. Torres, Machine Learning aplicado a la Ciberseguridad, Madrid: Zeroxword computing, 2019, pp. 19-19.

[53] M. A. Hossain, R. M. Noor, K. L. A. Yau, S. R. Azzuhri, M. R. Z'aba y I. Ahmedy, «Comprehensive survey of machine learning approaches in cognitive radio-based vehicular ad hoc networks,» *IEEE Access,* vol. 8, pp. 78054-78108, 2020.

[54] «Machine Learning. Aprendizaje supervisado,» Mathworks, 2019. [En línea]. Available: https://es.mathworks.com/discovery/supervised-learning.html. [Último acceso: 14 10 2021].

[55] S. B. Kotsiantis, I. Zaharakis y P. Pintelas, «Supervised machine learning: A review of classification techniques.,» *Emerging artificial intelligence applications in computer engineering,,* vol. 160, nº 1, pp. 3-24, 2007.

[56] T. Dietterich, «Overfitting and undercomputing in machine learning,» *ACM computing surveys (CSUR),* vol. 27, nº 3, pp. 326-327, 1995.

[57] I. Rish, «An empirical study of the naive Bayes classifier,» *IJCAI 2001 workshop on empirical methods in artificial intelligence,* vol. 3, nº 22, pp. 41-46, 2001.

[58] L. &. M. O. Rokach, Decision trees. Data mining and knowledge discovery handbook, Boston: Springer, 2005, pp. 165-192.

[59] J. Laaksonen y E. Oja, «Classification with learning k-nearest neighbors,» *Proceedings of International Conference on Neural Networks (ICNN'96),* vol. 3, pp. 1480-1483, 1996.

[60] T. Yiu, «Understanding Random Forest,» Towards Data Science, 12 06 2019. [En línea]. Available:

https://towardsdatascience.com/understanding-random-forest-58381e0602d2. [Último acceso: 06 12 2021].

[61] W. S. Noble, «What is a support vector machine?,» *Nature biotechnology,* vol. 24, nº 12, pp. 1565-1567, 2006.

[62] D. M. Powers, «Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation.,» *arXiv preprint arXiv,* 2020.

[63] C. Torrano Giménez, P. Recuero, F. Ramirez, S. Hernández y J. Torres, Machine Learning aplicado a la Ciberseguridad, Madrid: Zeroxword computing, 2019, pp. 23-23.

[64] M. Dash y H. Liu, «Feature selection for classification,» *Intelligent data analysis,* vol. 1, nº 1-4, pp. 131-156, 1997.

[65] L. Cayton, «Algorithms for manifold learning,» *Univ. of California at San Diego Tech,* vol. 12, nº 1-17, p. 1, 2005.

[66] S. Tsuge, M. Shishibori, S. Kuroiwa y K. Kita, «Dimensionality reduction using non-negative matrix factorization for information retrieval,» *IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace,* vol. 2, pp. 960-965, 2001.

[67] H. Hotelling, «Analysis of a complex of statistical variables with principal components.,» *J. Educ. Psy.,* vol. 24, pp. 498-520, 1933.

[68] H. Liu, X. Li, J. Li y S. Zhang, «Efficient outlier detection for high-dimensional data,» *IEEE Transactions on Systems, Man, and Cybernetics: Systems,* vol. 48, nº 12, pp. 2451-2461, 2017.

[69] W. T. Shaw, «Refinement of the Normal quantile,» 2007. [En línea]. Available: https://web.archive.org/web/20070609171237/http://www.mth.kcl.ac.uk/~shaww/web_page/papers/NormalQuantile1.pdf. [Último acceso: 10 12 2021].

[70] D. Anguita, L. Ghelardoni, A. Ghio, L. Oneto y S. Ridella, «The 'K'in K-fold cross validation,» *20th European Symposium on Artificial Neural*

Networks, Computational Intelligence and Machine Learning (ESANN), pp. 441-446, 2012.

[71] S. Khalid, T. Khalil y S. Nasreen, «A survey of feature selection and feature extraction techniques in machine learning,» *Science and information conference,* pp. 372-378, 2014.

[72] J. Ramos, «Using tf-idf to determine word relevance in document queries,» *Proceedings of the first instructional conference on machine learning ,* vol. 242, nº 1, pp. 29-48, 2003.

[73] J. Brownlee, «Why One-Hot Encode Data in Machine Learning?,» Machine Learning Mastery, 30 06 2020. [En línea]. Available: https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/. [Último acceso: 14 12 2021].

[74] C. Torrano Giménez, P. Recuero, F. Ramirez, S. Hernández y J. Torres, Machine Learning aplicado a la Ciberseguridad, Madrid: Zeroxword computing, 2019, pp. 89-98.

[75] I. Firdausi, A. Erwin y A. S. Nugroho, «Analysis of machine learning techniques used in behavior-based malware detection,» *Second international conference on advances in computing, control, and telecommunication technologies,* pp. 201-203, 2010.

[76] M. Kruczkowski y E. N. Szynkiewicz, «Support vector machine for malware analysis and classification.,» *IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT),* vol. 2, pp. 415-420, 2014.

[77] H. V. Nath y B. M. Mehtre, «Static malware analysis using machine learning methods,» *International Conference on Security in Computer Networks and Distributed Systems,* pp. 440-450, 2014.

[78] R. S. Pirscoveanu, S. S. Hansen, T. M. Larsen, M. Stevanovic, J. M. Pedersen y A. Czech, «Analysis of malware behavior: Type classification using machine learning,» *International conference on cyber situational awareness, data analytics and assessment (CyberSA),* pp. 1-7, 2015.

[79] M. G. Schultz, E. Eskin, F. Zadok y S. J. Stolfo, «Data mining methods for detection of new malicious executables,» *In Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001,* pp. 38-49, 2001.

[80] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi y A. Hamze, «Malware detection based on mining API calls,» *In Proceedings of the 2010 ACM symposium on applied computing ,* pp. 1020-1025, 2010.

[81] M. C. G. W. M. &. Z. X. Wojnowicz, «Wavelet decomposition of software entropy reveals symptoms of malicious code.,» *Journal of Innovation in Digital Ecosystems,,* vol. 3, nº 2, pp. 130-140, 2016.

[82] D. Bekerman, B. Shapira, L. Rokach y A. Bar, «Unknown malware detection using network traffic classification,» *Conference on Communications and Network Security (CNS) ,* pp. 134-142, 2015.

[83] H. S. Galal, Y. B. Mahdy y M. A. Atiea, «Behavior-based features model for malware detection,» *Journal of Computer Virology and Hacking Techniques,* vol. 12, nº 2, pp. 59-67, 2016.

[84] P. O'kane, S. Sezer y K. McLaughlin, «Detecting obfuscated malware using reduced opcode set and optimised runtime trace,» *Security Informatics,* vol. 5, nº 1, pp. 1-12, 2016.

[85] R. S. Pirscoveanu, S. S. Hansen, T. M. Larsen, M. Stevanovic, J. M. Pedersen y A. Czech, «Analysis of malware behavior: Type classification using machine learning.,» *International conference on cyber situational awareness, data analytics and assessment (CyberSA),* pp. 1-7, 2015.

[86] S. S. Hansen, T. M. T. Larsen, M. Stevanovic y J. M. Pedersen, «An approach for detection and family classification of malware based on behavioral analysis,» *International conference on computing, networking and communications (ICNC),* pp. 1-5, 2016.

[87] Z. Salehi, A. Sami y M. Ghiasi, «Using feature generation from API calls for malware detection,» *Computer Fraud & Security,* vol. 9, pp. 9-18, 2014.

[88] M. T. H. B. K. B. L. C. S. L. K. &. P. B. M. Chandramohan, «A scalable approach for malware detection through bounded feature space

behavior modeling,» *28th IEEE/ACM International Conference on Automated Software Engineering (ASE),* pp. 312-322, 2013 .

[89] M. Lindorfer, C. Kolbitsch y P. M. Comparetti, «Detecting environment-sensitive malware. In International Workshop on Recent Advances in Intrusion Detection,» *In International Workshop on Recent Advances in Intrusion Detection ,* pp. 338-357, 2011.

[90] M. G. Sobell, A practical guide to Ubuntu Linux., Pearson Education, 2015.

[91] «Virtualbox, Oracle VM,» [En línea]. Available: https://www.virtualbox.org/.

[92] Y. Lusky y A. Mendelson, «Sandbox Detection Using Hardware Side Channels,» *22nd International Symposium on Quality Electronic Design (ISQED),* pp. 192-197, 2021.

[93] RedHat, «¿Qué son las KVM?,» 2018. [En línea]. Available: https://www.redhat.com/es/topics/virtualization/what-is-KVM. [Último acceso: 15 12 2021].

[94] G. Van Rossum, «Python 3,» [En línea]. Available: https://www.python.org/download/releases/3.0/.

[95] K. O'Reilly, «Cape Sandbox. Instalación.,» 2018. [En línea]. Available: https://github.com/kevoreilly/CAPEv2/blob/master/systemd/README.md. [Último acceso: 14 10 2021].

[96] M. Aleksic, «Despliegue del hipervisor kvm sobre Ubuntu 20.04.,» PhoenixNAP, 2020. [En línea]. Available: https://phoenixnap.com/kb/ubuntu-install-kvm#ftoc-heading-6. [Último acceso: 15 10 2021].

[97] K. O'Reilly, «Cape Sandbox. Agente de comunicación con el entorno de análisis.,» 2018. [En línea]. Available: https://raw.githubusercontent.com/kevoreilly/CAPEv2/master/agent/agent.py. [Último acceso: 15 10 2021].

[98] K. O'Reilly, «Cape Sandbox. Documentación.,» 2018. [En línea]. Available:

https://capev2.readthedocs.io/en/latest/installation/host/configuration.html. [Último acceso: 16 10 2021].

[99] L. Fueris, «PIFMANIA: Pipeline for Malware Analysis. Script en Powershell para la extracción de todos los binarios de una máquina Windows,» *Universitat Oberta de Catalunya. Trabajo Fin de Máster.,* p. 60, 2021.

[100] TheZoo, «A Live Malware Repository,» [En línea]. Available: https://github.com/ytisf/theZoo. [Último acceso: 16 10 2021].

[101] L. Fueris, «PIFMANIA: Pipeline for Malware Analysis. Malware analysis pipeline.,» *Universitat Oberta de Catalunya. Trabajo Fin de Máster.,* pp. 15-16, 2021.

[102] L. Fueris, «PIFMANIA: Pipeline for Malware Analysis. Feature engineering,» *Universitat Oberta de Catalunya. Trabajo Fin de Máster,* pp. 55-56, 2021.

[103] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss y V. D. e. al., «Scikit-learn: Machine learning in python.,» *Journal of Machine Learning Research,* vol. 12, p. 2825–2830, 2011.

[104] «sklearn.feature_extraction.TfidfVectorizer,» Scikit-Learn, [En línea]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html. [Último acceso: 8 10 2021].

[105] J. Reynolds y J. Postel, «Assigned numbers,» *STD 2, RFC 1700,* 1994.

[106] ICANN, «Contract Between ICANN and the United States Government for Performance of the IANA Function,» 2001.

[107] M. Larsen y F. & Gont, «Recommendations for transport-protocol port randomization,» *BCP 156, RFC 6056,* 2011.

[108] TrendMicro, «Trojan ports usage,» 2012. [En línea]. Available: https://docs.trendmicro.com/all/ent/officescan/v10.6/en-us/osce_10.6_sp1_olh/gls_trojan_port.html. [Último acceso: 18 10 2021].

[109] I. Ilascu, «Most Cyber Attacks Focus on Just Three TCP Ports,» Bleeping Computer, 2019. [En línea]. Available: https://www.bleepingcomputer.com/news/security/most-cyber-attacks-focus-on-just-three-tcp-ports/. [Último acceso: 18 10 2021].

[110] Scikit-Learn, «QuantileTransformer function,» [En línea]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.QuantileTransformer.html. [Último acceso: 19 10 2021].

[111] J. Amat Rodrigo, «Análisis de Componentes Principales (Principal Component Analysis, PCA) y t-SNE,» Ciencia de datos, 06 2017. [En línea]. Available: https://www.cienciadedatos.net/documentos/35_principal_component_analysis. [Último acceso: 20 10 2021].

[112] «sklearn.decomposition.PCA,» Scikit-Learn, [En línea]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html. [Último acceso: 18 10 2021].

[113] A. Chavan, K. Kerakalamatti y S. S., «Implementation of Portable Antivirus System using Signature-based Detection and Heuristic Analysis,» *5th International Conference on Trends in Electronics and Informatics (ICOEI),* p. 1483, 2021.

[114] A. Chavan, K. Kerakalamatti y S. S., «Implementation of Portable Antivirus System using Signature-based Detection and Heuristic Analysis,» *5th International Conference on Trends in Electronics and Informatics (ICOEI),* p. 1482, 2021.

[115] D. Gibert, C. Mateu y J. Planes, «The rise of machine learning for detection and classification of malware: Research developments, trends and challenges.,» *Journal of Network and Computer Applications,* vol. 153, p. 102526, 2020.

[116] «sklearn.feature_extraction.DictVectorizer,» Scikit-Learn, [En línea]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.DictVectorizer.html. [Último acceso: 7 10 2021].

9. Anexos

A.1 *Script* en *Powershell* para la extracción de binarios *Windows*

```
$ Get−ChildItem−Path C:\ −Include *.exe −File −Recurse −ErrorAction \
−SilentlyContinue | ForEach−Object {Write−Host $_.FullName; \
New−Item−Path "C:\Users\cape\Desktop\good\" −Name \
$_.BaseName −ItemType "directory"; $dst_dir= −join \
("C:\Users\cape\Desktop\good\", "", $_.BaseName); Copy−Item \
$_.FullName −Destination $dst_dir}
```

A.2 Características extraídas para el *dataset*

A.2.1 Eventos del registro

registry_usage. Operaciones realizadas sobre el registro (claves leídas, escritas, eliminadas, etc.). Ejemplo:

- {'read_keys': ['DisableUserModeCallbackFilter', 'HKEY_LOCAL_MACHINE\\SOFTWARE\\Classes\\AppID\\upnpcont.exe\\AppId', 'HKEY_LOCAL_MACHINE\\SOFTWARE\\MICROSOFT\\COM3\\Com+Enabled', 'HKEY_LOCAL_MACHINE\\SOFTWARE\\Classes\\Wow6432Node\\CLSID\\{6D8FF8E0-730D-11D4-BF42-00B0D0118B56}\\(Default)'}

n_read_keys. Número de operaciones de lectura sobre el registro.

n_write_keys. Número de operaciones de escritura sobre el registro.

n_delete_keys. Número de operaciones de borrado sobre el registro.

A.2.2 Trazas de instrucciones

instruction_traces. Funciones y métodos ejecutados en modo usuario. Ejemplo:

- ['load', 'load', 'create', 'read']

instruction_traces_data. Datos de las funciones o métodos realizados en modo usuario en el sistema. Está característica está enlazada con la característica anterior. Ejemplo:

- ['KERNEL32.DLL', 'KERNELBASE.DLL', 'kernelbase.dll', 'C:\\Users\\cape\\AppData\\Local\\Temp\\FixKlez.com', 'ADVAPI32.dll']

A.2.3 Trazas de llamada a la API

api_call_traces. Llamadas a la API de Windows. Ejemplo:

- ['GetSystemTimeAsFileTime', 'memcpy', 'NtAllocateVirtualMemory', 'NtAllocateVirtualMemory', 'LdrGetDllHandle', 'LdrGetDllHandle', 'LdrGetProcedureAddress', 'NtOpenProcess', 'NtClose', 'SystemParametersInfoW', 'NtClose', 'NtOpenKey', 'GetSystemMetrics', 'GetSystemMetrics', 'SystemParametersInfoW', 'SystemParametersInfoW', 'RegCloseKey']

n_api_calls. Número de llamadas a la API de Windows totales.

windows_api_calls. Número de llamadas a la categoría *Windows* de la API de Windows.

services_api_calls. Número de llamadas a la categoría *Services* de la API de Windows.

system_api_calls. Número de llamadas a la categoría *System* de la API de Windows.

synchronization_api_calls. Número de llamadas a la categoría *Synchronization* de la API de Windows.

registry_api_calls. Número de llamadas a la categoría *Registry* de la API de Windows.

threading_api_calls. Número de llamadas a la categoría *Threading* de la API de Windows.

process_api_calls. Número de llamadas a la categoría *Process* de la API de Windows.

network_api_calls. Número de llamadas a la categoría *Network* de la API de Windows.

misc_api_calls. Número de llamadas a la categoría *Misc* de la API de Windows.

hooking_api_calls. Número de llamadas a la categoría Hooking de la API de Windows.

filesystem_api_calls. Número de llamadas a la categoría *Filesystem* de la API de Windows.

device_api_calls. Número de llamadas a la categoría *Device* de la API de Windows.

crypto_api_calls. Número de llamadas a la categoría *Crypto* de la API de Windows.

com_api_calls. Número de llamadas a la categoría *Com* de la API de Windows.

browser_api_calls. Número de llamadas a la categoría *Browser* de la API de Windows.

A.2.4 Tráfico de red

network_traffic. Tráfico de red (TCP y UDP) generado por la ejecución del binario. Ejemplo:

- {'tcp': [], 'udp': [{'src': '192.168.122.171', 'sport': 54460, 'dst': '192.168.122.1', 'dport': 53, 'offset': 24, 'time': 0.0}, {'src': '192.168.122.171', 'sport': 65396, 'dst': '239.255.255.250', 'dport': 3702, 'offset': 392, 'time': 4.251806974411011}, {'src': '192.168.122.171', 'sport': 54461, 'dst': '239.255.255.250', 'dport': 3702, 'offset': 1439, 'time': 4.324991941452026}]}

A.3 Transformación de la característica de uso del registro

```
from sklearn.feature_extraction import DictVectorizer
import re

def categorize_registry_usage(data):

    test=data
    i = 0
    for row in test:
        for key in row:
            j = 0
            for value in test[i][key]:
                try:
                    value_selected = re.sub(r"\..*","",re.search("\\\\(?:.(?!\\\\))+$",value).group(0))
                    test[i][key][j] = value_selected
                except:
                    continue
                j = j + 1
        i = i + 1

    data_categorized = test
    vec = DictVectorizer()
    vec = vec.fit_transform(data_categorized).toarray()
    data_categorized = np.array(vec).reshape((len(vec),-1))
    data_categorized = normalizeDataset (data_categorized)

    print("Registry features shape: " + str(data_categorized.shape))

    return data_categorized
```

A.4 Transformación de la característica de llamadas a la API

```
from sklearn.feature_extraction.text import TfidfVectorizer
def categorize_api_call_traces(data):

    data_categorized = data

    tfidf_model = TfidfVectorizer(preprocessor=lambda x: x, tokenizer=lambda x: x,
                                  ngram_range=(1, 1), max_features=1000, max_df=0.9,
                                  strip_accents='unicode', lowercase=True,
                                  analyzer='word', use_idf=True, smooth_idf=False,
                                  sublinear_tf=True)

    tfidf_vectorized = tfidf_model.fit_transform(data_categorized).toarray()

    data_categorized = np.array(tfidf_vectorized).reshape((len(tfidf_vectorized),-1))
    data_categorized = normalizeDataset (data_categorized)

    print("Api calls features shape: " + str(data_categorized.shape))

    return data_categorized
```

A.5 Transformación de la característica de tráfico de red

```
def categorize_network_traffic(data):

    test=data
    suspiciousConnectionPorts = ["7","19","20","21","22","23","25","37","53","69","79","80",
"102","110","111","135","137","138","139","443","445","161","443","502","512","513","514","1
433","1434","1723","3389","8080","20000",”44818","1001","10048","10100","1026","11000"
,"1120","113","1234","12345","12349","1243","139","18006","2140","21544","22222","2343
2",
25685","27374","3131","31337","31338","31339","3150","4000","44444","6267","6400","64
666","666","6667","6711","6776","6969","7300","7597","7626","7777","8012"]

    suspiciousConnections = []
    i = 0
    for row in test:
        j = 0
        numberOfTCPconnections = 0
        numberOfUDPconnections = 0
        suspiciousConnection = 0
        wellKnownPortsConnections = 0
        registeredPortsConnections = 0
        ephemeralPortsConnections = 0

        for key in row.keys():
            if (key == "udp"):
                numberOfUDPconnections = len(test[i][key])
            elif (key == "tcp"):
                numberOfTCPconnections = len(test[i][key])

            if (len(test[i][key])>0):
                k = 0
                for connection in test[i][key]:
                    if (str(test[i][key][k]['sport']) in suspiciousConnectionPorts or
str(test[i][key][k]['dport']) in suspiciousConnectionPorts):
                        suspiciousConnection = suspiciousConnection + 1

                    if (int(test[i][key][k]['sport']) < 1024 or int(test[i][key][k]['dport']) < 1024):
                        wellKnownPortsConnections = wellKnownPortsConnections + 1
                    elif (int(test[i][key][k]['sport']) > 49152 or int(test[i][key][k]['dport']) > 49152 ):
                        ephemeralPortsConnections = ephemeralPortsConnections + 1
                    else:
                        registeredPortsConnections = registeredPortsConnections + 1
                    k = k + 1
            j = j + 1
        i = i + 1

suspiciousConnections.append([suspiciousConnection, numberOfUDPconnections,
numberOfTCPconnections, wellKnownPortsConnections, registeredPortsConnections,
ephemeralPortsConnections])
    data_categorized = np.array(suspiciousConnections)
    data_categorized = categorize_int_column (data_categorized)

    print("Network features shape: " + str(data_categorized.shape))
    return data_categorized
```

A.6 Transformación de las características numéricas a través de la función cuantil

```
from sklearn.preprocessing import QuantileTransformer

def categorize_int_column (data):

    data_categorized = np.array(data).reshape((len(data),-1))
    data_categorized = data_categorized.astype('int64')
    quantile = QuantileTransformer(n_quantiles=5,  output_distribution='normal', random_state=42)
    data_categorized = quantile.fit_transform(data_categorized)

    return data_categorized
```

A.7 Reducción de características mediante el algoritmo PCA

```
from sklearn.decomposition import PCA
def PCACalculator(data, classes):

    pca = PCA()
    pca.fit(data)
    acumvar = pca.explained_variance_ratio_.cumsum()
    variance = 0.95
    components = sum(acumvar<variance) + 1

    pca = PCA(n_components=components)
    pca.fit(data)
    values_pca = pca.transform(data)

    transform_Back = pca.inverse_transform(values_pca)

    return transform_Back
```

A.8 Desglose del dataset según características

```
import sys
import csv
import json

def read_and_process_data(filename):
    print('Data path:', filename)
    try:

        header = []
        data=[]
        classes = []
        binaryMetadata = []
        benignTimes = 0
        malwareTimes = 0
        csv.field_size_limit(sys.maxsize)
        with open(filename) as file:
            rowCounter = 0
            csvReader = csv.reader(file, delimiter=',')
            for row in csvReader:
                if rowCounter == 0:
                    header = row[3:-1]
                else:
                    if (row[2] == "0"):
                        data.append([x for x in row[3:-1]])
                        classes.append(row[2])
                        binaryMetadata.append(row[1])
                        benignTimes = benignTimes + 1
                    elif (row[2] == "1"):
                        data.append([x for x in row[3:-1]])
                        classes.append(row[2])
                        binaryMetadata.append(row[1])
                        malwareTimes = malwareTimes + 1
                rowCounter = rowCounter + 1

        return header, data, classes, binaryMetadata
    except IOError:
        raise IOError('Could not read: ' + filename)
```

```
def split_dataset (header, data, columns):
    newData=[]
    newHeader = columns
    for line in data:
        newline = []
        for column in columns:
            indexColumn = header.index(column)
            newline.append(line[indexColumn])
        newData.append(newline)

    return newHeader, newData

def split_dataset_per_column (header, data,classes):

    datasetPerColumn = {}

    for column in header:
        datasetPerColumn[column] = {}
        datasetPerColumn[column]["data"] = []
        datasetPerColumn[column]["name"] = column

    lineIndex = 0
    for line in data:
        columnIndex = 0
        for column in line:

            column = column.replace("\"", "'")
            column = column.replace("\', \'", "\", \"")
            column = column.replace("\", \'", "\", \"")
            column = column.replace("\', \"", "\", \"")
            column = column.replace("[\'", "[\"")
            column = column.replace("\']", "\"]")
            column = column.replace("': ", "\": ")
            column = column.replace("], \'", "], \"")
            column = column.replace("{\'", "{\"")
            column = column.replace("}\'", "}\"")
            column = column.replace("\"\"", "'\"")
            column = column.replace(", '\"", ", \"\"")
            column = column.replace("\\'", "'")
            column = column.replace(": '", ": \"")
            column = column.replace(", \'", ", \"")

            columnParsed = json.loads(column)

            if lineIndex == 0:
                datasetPerColumn[header[columnIndex]]["data_type"] = type(columnParsed)

            datasetPerColumn[header[columnIndex]]["data"].append(columnParsed)
            columnIndex = columnIndex + 1
        lineIndex = lineIndex + 1

    return datasetPerColumn
```

```
if __name__ == '__main__':
    print('Jorge Díaz Navarro: TFM')
    if len(sys.argv) > 1:
        filename = sys.argv[1]
    else:
        filename = '../malware-dataset-theZoo.csv'

    header, data, classes , binaryMetadata = read_and_process_data(filename)
    header_index = 0
    data_index = 1
    classes_index = 2
    binaryMetadata_index = 3

    registry_columns = ['registry_usage', 'n_read_keys', 'n_write_keys', 'n_delete_keys']

    instruction_traces_columns = ['instruction_traces', 'instruction_traces_data', 'n_inst',
                                  'read_inst', 'write_inst', 'delete_inst', 'load_inst',
                                  'create_inst', 'execute_inst', 'move_inst', 'copy_inst',
                                  'findwindow_inst', 'start_inst', 'modify_inst', 'browser_inst']

    api_calls_columns = ['api_call_traces', 'n_api_calls', 'windows_api_calls',
                                    'services_api_calls', 'system_api_calls', 'synchronization_api_calls',
                                  'registry_api_calls', 'threading_api_calls', 'process_api_calls',
                                  'network_api_calls', 'misc_api_calls', 'hooking_api_calls',
                                  'filesystem_api_calls', 'device_api_calls',  'crypto_api_calls',
                                  'com_api_calls', 'browser_api_calls']

    network_columns = ['network_traffic']

    registry_dataset = split_dataset(header, data ,registry_columns)
    instruction_traces_dataset = split_dataset(header, data ,instruction_traces_columns)
    api_calls_dataset = split_dataset(header, data ,api_calls_columns)
    network_dataset = split_dataset(header, data ,network_columns)

    registry_per_columns = split_dataset_per_column(registry_dataset[header_index],
                                                    registry_dataset[data_index], classes)

    instruction_per_columns =
split_dataset_per_column(instruction_traces_dataset[header_index],
                                                     instruction_traces_dataset[data_index],
                                            classes)

   api_calls_per_columns = split_dataset_per_column(api_calls_dataset[header_index],
                                                    api_calls_dataset[data_index], classes)

   network_dataset_per_columns =
split_dataset_per_column(network_dataset[header_index],
                                                        network_dataset[data_index],classes)
```

A.9 Entrenamiento del conjunto de datos

```
def machineLearning (attributes, classes):

    names = ["A1","A2","A3","A4","A5","A6","A7","A8","A9","A10","A11","A12","A13"]

    square_root=int(math.sqrt(len(classes)))

    classifiers = [
        KNeighborsClassifier(leaf_size=1, n_neighbors=2, p=2,
                weights='distance'),
        KNeighborsClassifier(leaf_size=1, n_neighbors=3, p=2,
                weights='distance'),
        KNeighborsClassifier(leaf_size=1, n_neighbors=square_root, p=2,
                weights='distance'),
        SVC(kernel="linear", C=0.25),
        SVC(kernel="linear", C=1),
        SVC(kernel="linear", C=2),
        DecisionTreeClassifier(max_depth=3, random_state=42),
        DecisionTreeClassifier(max_depth=10, min_samples_leaf=2),
        DecisionTreeClassifier(max_depth=10, random_state=42),
        RandomForestClassifier(max_depth=10, random_state=42),
        RandomForestClassifier(max_depth=50, random_state=42),
        RandomForestClassifier(max_depth=100, random_state=42),
        GaussianNB()]

kf = KFold(n_splits=10, shuffle=True)

    scores_dict = {
          'accuracy_score':make_scorer(accuracy_score),
          "roc_auc_score":make_scorer(roc_auc_score),
          "f1_score":make_scorer(f1_score),
          "recall_score":make_scorer(recall_score)
          }
    for name, clf in zip(names, classifiers):

        print("*************" + name + "***************")

        for score in scores_dict.keys():

            scores = cross_val_score(clf, attributes, classes, scoring=scores_dict[score], cv=kf,
n_jobs=1)

            print('%.3f,' % (mean(scores)))
```

A.10 Validación de los resultados

La tabla muestra la validación de los resultados a través de las métricas *accuracy_score, roc_auc_score, f1_score, recall_score,* en ese orden.

		Algoritmos												
		A1	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**A10**	**A11**	**A12**	**A13**
Características	**C1**	0.832, 0.830, 0.811, 0.742,	0.828, 0.825, 0.826, 0.774,	0.861, 0.861, 0.842, 0.842,	0.807, 0.807, 0.794, 0.743,	0.834, 0.837, 0.833, 0.813,	0.838, 0.835, 0.833, 0.834,	0.805, 0.794, 0.801, 0.809,	0.817, 0.827, 0.833, 0.836,	0.837, 0.842, 0.827, 0.834,	0.875, 0.876, 0.881, 0.879,	0.878, 0.868, 0.868, 0.878,	0.873, 0.887, 0.876, 0.872,	0.648, 0.669, 0.709, 0.835,
	C2	0.809, 0.802, 0.784, 0.741,	0.807, 0.814, 0.818, 0.808,	0.847, 0.856, 0.856, 0.891,	0.792, 0.787, 0.775, 0.732,	0.787, 0.787, 0.776, 0.733,	0.788, 0.791, 0.777, 0.739,	0.763, 0.769, 0.751, 0.829,	0.827, 0.831, 0.840, 0.856,	0.822, 0.841, 0.836, 0.862,	0.866, 0.868, 0.874, 0.893,	0.869, 0.875, 0.871, 0.891,	0.862, 0.862, 0.855, 0.892,	0.769, 0.767, 0.728, 0.659,
	C3	0.844, 0.839, 0.840, 0.805,	0.858, 0.850, 0.836, 0.837,	0.858, 0.859, 0.869, 0.898,	0.780, 0.790, 0.778, 0.758,	0.784, 0.782, 0.775, 0.755,	0.792, 0.787, 0.778, 0.758,	0.830, 0.816, 0.827, 0.877,	0.861, 0.857, 0.871, 0.885,	0.868, 0.858, 0.862, 0.871,	0.884, 0.880, 0.881, 0.909,	0.891, 0.884, 0.891, 0.921,	0.891, 0.884, 0.896, 0.915,	0.717, 0.719, 0.691, 0.637,
	C4	0.941, 0.943, 0.947, 0.955,	0.938, 0.944, 0.943, 0.957,	0.935, 0.931, 0.934, 0.949,	0.955, 0.957, 0.955, 0.920,	0.959, 0.955, 0.955, 0.923,	0.956, 0.960, 0.956, 0.924,	0.964, 0.965, 0.964, 0.948,	0.946, 0.951, 0.940, 0.942,	0.953, 0.944, 0.944, 0.947,	0.968, 0.968, 0.961, 0.941,	0.967, 0.966, 0.965, 0.949,	0.969, 0.966, 0.967, 0.950,	0.889, 0.891, 0.891, 0.917,

		Algoritmos												
		A1	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**A10**	**A11**	**A12**	**A13**
	C5	0.858, 0.852, 0.839, 0.795,	0.840, 0.847, 0.838, 0.827,	0.873, 0.870, 0.875, 0.900,	0.805, 0.797, 0.787, 0.743,	0.802, 0.803, 0.794, 0.752,	0.805, 0.806, 0.785, 0.746,	0.803, 0.796, 0.806, 0.844,	0.857, 0.864, 0.858, 0.880,	0.851, 0.858, 0.854, 0.873,	0.896, 0.888, 0.893, 0.932,	0.889, 0.891, 0.905, 0.935,	0.895, 0.892, 0.890, 0.922,	0.730, 0.716, 0.686, 0.604,
	C6	0.846, 0.843, 0.840, 0.809,	0.861, 0.863, 0.848, 0.826,	0.873, 0.881, 0.876, 0.869,	0.879, 0.874, 0.872, 0.867,	0.881, 0.883, 0.868, 0.881,	0.887, 0.883, 0.887, 0.880,	0.828, 0.810, 0.831, 0.868,	0.855, 0.875, 0.870, 0.883,	0.851, 0.853, 0.858, 0.903,	0.904, 0.908, 0.895, 0.916,	0.914, 0.905, 0.904, 0.928,	0.904, 0.908, 0.903, 0.915,	0.710, 0.715, 0.663, 0.576,
	C7	0.943, 0.940, 0.949, 0.948,	0.942, 0.939, 0.936, 0.950,	0.928, 0.917, 0.912, 0.907,	0.975, 0.977, 0.976, 0.964,	0.972, 0.973, 0.970, 0.963,	0.969, 0.962, 0.966, 0.958,	0.969, 0.959, 0.966, 0.959,	0.965, 0.966, 0.959, 0.955,	0.960, 0.952, 0.953, 0.957,	0.960, 0.961, 0.955, 0.937,	0.955, 0.959, 0.962, 0.947,	0.959, 0.953, 0.960, 0.957,	0.659, 0.669, 0.692, 0.849,
	C8	0.862, 0.854, 0.846, 0.809,	0.863, 0.866, 0.858, 0.852,	0.883, 0.879, 0.884, 0.905,	0.839, 0.827, 0.837, 0.842,	0.834, 0.834, 0.831, 0.829,	0.832, 0.840, 0.822, 0.839,	0.823, 0.813, 0.832, 0.844,	0.857, 0.874, 0.859, 0.891,	0.866, 0.866, 0.865, 0.879,	0.893, 0.892, 0.896, 0.916,	0.898, 0.889, 0.894, 0.920,	0.889, 0.907, 0.895, 0.927,	0.731, 0.734, 0.693, 0.609,
	C9	0.871, 0.871, 0.864, 0.906,	0.875, 0.866, 0.883, 0.894,	0.844, 0.845, 0.841, 0.845,	0.814, 0.826, 0.803, 0.808,	0.814, 0.820, 0.822, 0.826,	0.832, 0.826, 0.819, 0.826,	0.901, 0.891, 0.890, 0.918,	0.926, 0.928, 0.918, 0.915,	0.918, 0.912, 0.929, 0.939,	0.935, 0.937, 0.930, 0.943,	0.927, 0.926, 0.930, 0.944,	0.933, 0.928, 0.919, 0.946,	0.729, 0.721, 0.696, 0.648,

		Algoritmos												
		A1	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**A10**	**A11**	**A12**	**A13**
	C10	0.930, 0.929, 0.929, 0.960,	0.923, 0.939, 0.939, 0.953,	0.878, 0.876, 0.880, 0.871,	0.965, 0.970, 0.964, 0.971,	0.966, 0.960, 0.962, 0.967,	0.964, 0.961, 0.968, 0.965,	0.954, 0.957, 0.956, 0.948,	0.953, 0.933, 0.952, 0.944,	0.949, 0.942, 0.942, 0.957,	0.972, 0.969, 0.966, 0.970,	0.965, 0.957, 0.958, 0.963,	0.965, 0.964, 0.960, 0.964,	0.725, 0.717, 0.646, 0.541,

Tabla 8. Validación de los resultados de cada modelo entrenado.