Detección y Clasificación de Zero-Day Malware a través de Data Mining y Machine Learning

Augusto Recordon y Silvia Ruiz Diaz
Director de tesis: Dra Claudia Pons, Facultad de Informática, UNLP y Comisión de investigaciones Científicas CIC

Universidad Nacional de La Plata, Buenos Aires, Argentina
difusion@info.unlp.edu.ar
https://unlp.edu.ar/

Abstract. Muchos estudios sugieren que, durante los últimos años, ha habido un incremento exponencial de los ataques informáticos, causando a las organizaciones pérdidas financieras en el orden de los millones. Mientras muchas compañías dedican tiempo y recursos al desarrollo de antivirus; la complejidad, la velocidad de propagación y la capacidad polimórfica que poseen los virus modernos representan enormes desafíos para estas empresas. Motivados por encontrar nuevas alternativas, la comunidad de científicos de datos ha descubierto que la utilización de técnicas de *machine learning* y *deep learning* para la detección y clasificación de *malware* puede ofrecer una opción más que competitiva. Para esta investigación se comenzará realizando las extracción de información de un conjunto de datos compuesto por once mil archivos `ASM` y `bytes` correspondientes a nueve familias distintas de *malwares*. Luego, mediante la implementación de algoritmos de *machine learning* se intentará clasificar estos *malwares* en sus correspondientes familias. De forma complementaria, se realizará una clasificación binaria para detección *malware/no malware*, con un conjunto reducido de programas benignos, finalizando así con la elaboración de comparaciones y conclusiones.

Keywords: Machine learning · malware · seguridad informática · virus · zero-day · data mining · inteligencia artificial · redes neuronales

1 Introducción

El número de aplicaciones y utilidades de Internet es, sin lugar a dudas muy extenso, pero la base es una misma: el *software*. *Software* en nuestros dispositivos, en servidores y en equipos intermedios que nos conectan. *Software* en el cual, como se ha mencionado, confiamos diariamente. Sin embargo éste, como toda construcción humana, es susceptible a errores, vulnerabilidades y escenarios no previstos. Más aún, los errores y vulnerabilidades, una vez detectados, llevan tiempo corregir y, en muchos casos, depende de las compañías y usuarios de estos sistemas aplicar la actualizaciones que solucionan los problemas.

En este contexto, los *hackers*, personas malintencionadas que se aprovechan de las vulnerabilidades del *software* y/o de la confianza de las personas, despliegan sus conocimientos y herramientas para llevar a cabo sus objetivos. Quizás la más común de estas herramientas es el **malware**. El término *malware*, de **mal**icious soft**ware**, es un programa cuyo fin es comprometer cualquier computadora o dispositivo inteligente, diseñado por un *hacker* con fines maliciosos, ya sea para robar información confidencial, penetrar redes, dañar infraestructuras críticas, etc. Estos programas pueden incluir virus, *worms*, troyanos, *spyware*, *bots*, *rootkits*, *ransomware*, entre otros.

Empresas dedicadas al desarrollo de anti-virus, tales como *Norton*, *AVG*, *McAfee*, *Kaspersky*, entre otros; hacen su mejor esfuerzo para hacer frente a esta problemática. Tradicionalmente, estos programas utilizaban el método *signature-based* para la detección de *malware*. La *signature* es una secuencia corta de bytes que sirve para identificar *malwares* conocidos, sin embargo este método no es capaz de proveer una forma de identificar ataques *zero-day*, o *malwares* polimórficos que utilizando técnicas de ofuscación son capaces de crear cientos de variedades de si mismo, dado que no cuentan con registros de los mismos.

El propósito de esta investigación tiene como objetivo central evaluar la efectividad de utilizar distintas técnicas de análisis estático relacionadas con *machine learning* y *data mining* para la detección y clasificación de *malware*. Aplicar estas técnicas para la detección temprana de *malware* puede resultar de particular utilidad para identificar ataques *zero-day*. Un *zero-day* es un *malware* desconocido, es decir, un nuevo tipo de código malicioso, para el cual aún no se han creado parches o revisiones que resuelvan la falla de seguridad.

2 Data Mining

2.1 Conjunto inicial de datos

El conjunto inicial de datos con el que se realizó el experimento fue obtenido del sitio *Kaggle*[1]. El mismo se compone de un conjunto de archivos de tipo `ASM` (programas escritos en lenguaje ensamblador, que guardan una relación cercana con las instrucciones en código máquina) y su correspondiente archivo de tipo `BYTES` (archivos que contienen el código de máquina ejecutable en representación hexadecimal), equivalentes a aproximadamente once mil *malwares*. Cada uno de estos archivos pertenece a una de nueve clases llamadas Ramnit, Lollipop, Kelihos_ver3, Vundo, Simda, Tracur, Kelihos_ver1, Ofuscator.ACY y Gatak.

2.2 Análisis y selección de características más relevantes

Con los archivos ya descargados, el siguiente paso consiste en teorizar qué características puede llegar a ser útiles para identificar programa similares. Para esto

[1] `https://www.kaggle.com/c/malware-classification/`

es necesario estudiar tanto la estructura de cada archivo (tamaño y fisonomía), como su funcionamiento: instrucciones que ejecuta y uso de librerías. Entre algunas de las características que fueron consideradas más relevantes se encuentran las librerías DLL, códigos de operación, códigos de sección, códigos de operación - N Gramas, tamaños de archivos, tamaños de archivos comprimidos y *snapshots* de los primeros 1024 *bytes*.

2.3 Extracción de los Datos

Una vez que se han identificado aquellos aspectos de los archivos que desean ser estudiados, la siguiente tarea consiste en la elaboración de un conjunto de procesos que permita la extracción de estos datos.

A continuación se describen los distintos procesos que conforman un *pipeline* de tareas de minería de datos que fueron ejecutadas para la construcción de un *dataset* único a partir de miles de muestras de *malware*.

DLLs, Secciones y Códigos de Operación Partiendo del supuesto que los archivos de una misma familia son similares en cuanto a estructura y comportamiento, se estudian los códigos de operación, sección y uso de `DLL`s, con el fin de determinar la veracidad de este supuesto.

De este modo, para la extracción de estos atributos se diseñó e implementó un *pipeline* de procesos, el cual tiene como objetivo final determinar cuáles son los *features* más relevantes para cada familia, así como también contabilizar las ocurrencias de estos atributos relevantes para cada una de las muestras disponibles.

Procesamiento de archivos `ASM` El primer paso en el proceso de minería de estos *features* consiste en tomar los archivos `ASM` y recorrerlos línea por línea en búsqueda de la posible ocurrencia de atributos de interés, los cuales son totalizados. De esta manera, se construye una estructura que contiene, por cada archivo `ASM` disponible, la cantidad de veces que cada *feature* fue encontrado.

Identificación de DLLs La lógica para la detección de *DLL*s debe ser capaz de detectar la cadena `.dll` sin ser sensible a mayúsculas ni minúsculas. A su vez, debe evitar procesar líneas correspondientes a comentarios.

Identificación de Códigos de Sección Cada línea comienza con un prefijo que identifica el tipo de sección a la que la línea en cuestión pertenece. Este código siempre es seguido por un dos puntos. De este modo, extraer el código de

sección es tan simple como obtener la cadena a la izquierda del primer dos puntos.

Identificación de Códigos de Operación Se asume como código de operación a la primer palabra de una línea, en tanto y en cuanto no se trate de un comentario.

Totalización de Ocurrencias y Cálculo de Proporciones Una vez procesados los archivos `ASM` y obtenido un total de occurrencias de cada *feature* por archivo, se debe determinar cuáles de estos *features* son los que caracterizan realmente a cada una de las familias. Una solución simple a este problema podría ser totalizar las ocurrencias de cada atributo (por familia), para luego quedarse con aquellos que produjeron cuentas más altas. Sin embargo, este enfoque puede llevar a problemas e inconsistencias en presencia de valores anómalos. Por lo que la estrategia utilizada consistió en realizar un cálculo de proporciones.

De este modo, el proceso genera dos archivos de salida. Uno análogo al resultado del proceso anterior, pero con las cantidades de ocurrencias expresadas en proporciones. Este archivo será de gran utilidad para el último paso del proceso de extracción de atributos. El segundo archivo contiene una totalización de las proporciones anteriormente mencionadas por cada una de las nueve familias de *malware* que se disponen. Este archivo será utilizado por el siguiente paso del proceso.

Determinación de los *features* más relevantes Una vez que se ha ejecutado el paso anterior, se dispone de las proporciones de ocurrencias de cada *feature* por archivo. De este modo, es posible totalizar estos valores por familia, lo que permite determinar cuáles son los atributos más relevantes para cada una de las clases de *malware* disponible.

Dada la gran cantidad de atributos disponible, es necesario poder reducir este número sin sacrificar demasiada *performance* para los modelos de *machine learning*. Para ello, se debe escoger un *punto de corte* adecuado. Esta tarea puede realizarse con facilidad si se vuelcan las cantidades de cada atributo en un gráfico de barras. Adicionalmente, cabe destacar que se elegirá el mismo número para todos los *features*, con la intención de producir un conjunto de datos balanceado.

Consolidación de Resultados Habiendo determinado los atributos más importantes para cada una de las familias, el único paso restante consiste en construir una lista única de atributos, sin repeticiones. Esta lista es utilizada en

conjunto con el archivo generado en el segundo paso, correspondiente a las proporciones de cada *feature* en cada archivo, para la elaboración de un *dataset* que contiene, para cada archivo, la proporción de ocurrencias de cada uno de los atributos que son relevantes para cualquiera de las familia.

2.4 *Snapshots* de Archivos `ASM`

De forma similar a la foto de identificación en un documento o un pasaporte, se propone *sacar una foto* en escala de grises de los primeros 1024 *bytes* de cada archivo `ASM` y utilizar estas imágenes de 32 x 32 *bytes* para entrenar una red neuronal que sea capaz de, por cada muestra analizada, asignar una probabilidad de que la misma pertenezca a cada una de las familias.

Captura de los *snapshots* El primer paso en la de minería de datos en este punto involucró la generación de *snapshots* a partir de los primeros 1024 *bytes* de cada archivo `ASM`. De este modo, se utilizó la librería de *Python* `imageio`[2] para la creación de imágenes de 32 x 32 *bytes* en escalas de grises. La figura 1 presenta una muestra en una escala de 125% de su tamaño original para su mejor visualización.

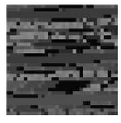

Fig. 1. *snapshot* de 32 x 32 *bytes*

Entrenamiento de la Red Neuronal Una vez generada la totalidad de las imágenes, se procedió a la construcción de una red neuronal utilizando *TensorFlow*. El objetivo de este proceso no es el de obtener la estimación más precisa posible, sino el de poder observar, a grandes rasgos, la similitud entre archivos pertenecientes a una misma clase. Teniendo que ser capaz de poder realizar estimaciones para cada una de las muestras, la red fue construida utilizando *k-fold*. La técnica de *k-fold* permite dividir el conjunto de datos en k partes iguales (conocidas como *folds* o *splits*) y ejecutar sobre ellas k corridas. En cada corrida se toma un *split* distinto como conjunto de *test* y a los restantes como conjunto de datos para entrenar el modelo. Para la construcción de la red se utilizó un k de 5. Una vez ejecutada la red, se obtuvo una precisión de alrededor de 0.62

[2] `https://pypi.org/project/imageio/`

2.5 Tamaños de Archivos y *Compression Rate*

Continuando con el estudio de las muestras desde el punto de vista de su fisonomía, se procedió a determinar el tamaño del archivo `ASM` y también del archivo `bytes` para cada *malware* que conforma el conjunto de datos. Adicionalmente se calcularon los tamaños de ambos archivos una vez comprimidos mediante `gzip`[3].

Partiendo del supuesto que archivos de una misma familia deben tener un tamaño similar, la minería de estos atributos se realiza con el objetivo de determinar si, en efecto, los distintos tamaños y *compression rate* pueden ser útiles para la identificación de cada *malware* dentro de su correspondiente familia.

Así, se construyó un *script* capaz de comprimir los distintos archivos y de determinar el tamaño de los mismos, almacenando los resultados en una tabla.

2.6 N-gramas

El último conjunto de atributos extraído de los archivos *malware* se trata de los 2, 3 y 4-gramas mas importantes, formados por las ocurrencias de los códigos de operación más relevantes. Esta fue, quizás, la tarea más extensa de todas.

Obtener *n-gramas* a partir de una secuencia de valores significa recorrer la misma con una ventana de tamaño n, desplazándose una posición a la vez, registrando el valor observado.

El objetivo al construir los *n-gramas* es analizar las operaciones ejecutadas pero, en lugar de enfocarnos en ellas individualmente, nos centramos en estudiar las secuencias de instrucciones que se ejecutan juntas. Se pretende determinar si miembros de una misma familia de *malware* tienden a ejecutar la misma serie de operaciones. Es importante tener en cuenta que se trabaja únicamente con los códigos de operaciones más relevantes para minimizar el número de combinaciones posibles.

De este modo, el trabajo a ser realizado se puede dividir en dos tareas. Por un lado tenemos la construcción de los *n-gramas* basados en los códigos de operación más importantes. Por otro lado se debe realizar la identificación y obtención de los *n-gramas* más relevantes para cada familia.

La primer tarea, la de la obtención de los *n-gramas*, se hace en base a los códigos de operación más relevantes, como fue mencionado en la subsección 2.3. Así, los archivos `ASM` deben ser recorridos nuevamente, para armar los *n-gramas*.

[3] *gzip*, o *GNU Zip*, es una técnica de compresión estandar.

Una vez extraídos los *n-gramas*, se procede al filtrado de aquellos más relevantes, de forma similar al trabajo realizado con otros *features* como `DLLs`, códigos de sección y operación.

Cabe destacar que la decisión de utilizar solamente los código de operación más relevantes se tomó con la intención de reducir los números de combinaciones posibles. Aún así, se obtuvieron un gran número de *n-gramas* distintos. Por ejemplo, la familia 1 posee 19258 *4-gramas* distintos.

3 Análisis Exploratorio y Preprocesamiento de los datos

El proceso de *Data Mining* finalizó con la construcción de un *dataset* producto de la extracción de los distintos atributos que fueron identificados como de interés para el análisis y clasificación de *malware*. Sin embargo, el archivo `csv` resultante no está listo aún para ser utilizado para entrenar los modelos de *machine learning*, sino que es necesario llevar a cabo un número de tareas adicionales para mejorar la calidad del mismo.

A través del Análisis Exploratorio de los Datos (**EDA** por sus siglas en inglés), se obtendrá un mejor entendimiento de los datos recolectados mediante métricas y gráficas. A su vez, este análisis permitirá el tratamiento de datos nulos o datos faltantes, determinar la importancia de las variables para proceder a su selección y extracción, y el estandarizado o escalado de los atributos.

3.1 Estructura y Contenido del *dataset*

Antes de comenzar cualquier tipo de análisis de los datos que conforman el *dataset*, es necesario estudiar la propia estructura del archivo. Esto permite adquirir cierta dimensión del volumen de datos con el que se trabajará. Utilizando *Pandas* se cargó el archivo `csv` en un *dataframe* y, así, se observó que el *dataset* resultante posee **10868** filas, correspondientes a cada una de las muestras disponibles, y **242** columnas.

3.2 Valores nulos o datos faltantes

La presencia de datos faltantes o nulos en un *dataset* suele tener un impacto negativo en la *performance* de lo modelos que se entrenen en base a él. Por este motivo la primera tarea que suele llevarse a cabo en todo preprocesamiento de datos es la de búsqueda e identificación de valores nulos, o faltantes, y su correspondiente corrección.

Existen diversas técnicas para el tratamiento de datos nulos. Para seleccionar la más adecuada es imperativo comprender la naturaleza de los valores faltantes. En nuestro caso existen dos orígenes de datos nulos:

1. Archivos que, por problemas de *encoding* e integridad no pudieron ser analizados, por ende estas filas estaban prácticamente vacías, salvo por las columnas correspondientes a lo tamaños de archivo. En total se identificaron un poco más de 800 muestras.
2. Archivos que no registraban ocurrencias para atributos que resultaron relevantes para otras familias.

Así, se identificó que los datos faltantes tienen dos orígenes distintos. Para la resolución del primer problema (archivos corruptos), no hubo otra alternativa más que eliminar las filas correspondientes. Mientras que para el segundo problema (columnas sin mediciones), se completaron las celdas con cero, dado que, en este contexto, un valor nulo representa cero ocurrencias.

3.3 Importancia de las variables

Con el *dataset* y sus valores nulos ya resueltos, se procedió a determinar la importancia de las variables. Hacer un estudio de la importancia de las variables es de gran utilidad, ya que permite comprender qué atributos son más relevantes para el modelo, lo que, a su vez, aporta beneficios adicionales tales como:

- Verificar la correctitud del modelo y evaluar posibilidades de mejoras al centrar el enfoque en aquellos atributos que son más importante para el modelo.
- Acortar los tiempos de entrenamiento sin sacrificar demasiada *performance* al seleccionar los atributos más relevantes y descartar aquellos cuya incidencia es despreciable.
- Mayor interpretabilidad del modelo sin tener que sacrificar, necesariamente, demasiada *performance*, si se hace una selección de atributos adecuada.

Para llevar adelante este proceso, se utilizó un algoritmo de clasificación *Random Forest*. Se debe recordar que un *Random Forest* es un ensamble de *Decision Trees* que utiliza una variación del método *bagging*, en donde muchos árboles independientes son entrenados utilizando el mismo conjunto de datos. Normalmente un *forest* puede contener varios cientos de árboles.

3.4 Selección de variables

Determinar la importancia de las variables no sólo nos puede ayudar a lograr una mejor interpretación de los datos, sino que también nos permite establecer un *ranking* y seleccionar aquellos *features* que son realmente importantes para el modelo de predicción.

El resultado de aplicar este proceso nos permitió reducir notablemente la complejidad del *dataset*, el cuál pasó de tener 242 columnas a tan sólo 89.

3.5 Estandarización de atributos

Muchas veces se trabaja con datos con magnitudes que resultan muy diferentes entre sí. Esto puede ser un problema para muchos estimadores empleados por los algoritmos de *machine learning* que son sensibles al estandarizado de atributos, ellos podrían tener un comportamiento errado si sus valores no fueran más o menos similares. Dependiendo del problema que se quiere abordar, se pueden utilizar diferentes técnicas, tanto de estandarización, como de normalización de los datos, para que estos resulten útiles a la hora de poner en marcha los algoritmos de *machine learning* que se desean. Para la presente investigación se realizó el escalado estándar, ya que para realizar la extración de atributos utilizando *Kernel PCA*, el escalado estándar funciona muy bien.

3.6 Correlación

Del gráfico de la correlación de *Pearson*, la cual calcula un coeficiente que establece la medida en la que dos variables se correlacionan, se desprenden algunas correlaciones obvias tales como las de los *n-gramas* con otros *n-gramas* similares, como `dd` y `dd_dd_dd_dd` con valor de 0.98. También se percibieron otras correlaciones que hablan de la estructura de los archivos. Por ejemplo, la sección `HEADER` presenta un **0.7** de correlación con la sección `.idata`. Por otro lado, algunas correlaciones son esperables dado el funcionamiento del lenguaje *assembler*, por ejemplo, la operación `mov`, para el llamado a subrutinas tiene una correlación de **.85** y **.82** con las operaciones relacionadas al pasaje de parámetros, `push` y `pop`.

3.7 Extracción de atributos con *Kernel PCA*

Principal Component Analysis (PCA), es una herramienta que se utiliza para reducir la dimensionalidad de los datos sin perder información. PCA reduce la dimensión hallando algunas combinaciones lineales ortogonales (componentes principales) de las variables originales con la varianza más alta. Estos componentes principales no tienen correlación y se encuentran ordenados de manera que los primeros componentes principales expliquen la mayoría de la varianza en los datos originales. Así se pudo determinar cómo, con un número de componentes superior a 40 se logra explicar un gran porcentaje de la varianza del modelo. En la práctica el número de componentes a seleccionar depende de qué tanta varianza se desea que el KPCA explique. Para esta investigación se fijó el valor en 99.5%, por lo que tuvieron que tomarse los primeros 53 componentes principales.

4 Machine Learning

Desde una perspectiva del *machine learning*, tanto el problema de la detección de *malware*, como el de identificación de familias para cada una de las muestras,

pueden ser considerados problemas de clasificación. En el caso de la detección de *malware*, donde lo que se intenta identificar es si una muestra es, en efecto, un programa malicioso, la clasificación es binaria: la muestra es o no es un *malware*. En el caso de clasificación de familias, el problema es multi-clase, dado que debe determinarse a cuál de las nueve familias pertenece la muestra.

5 Clasificación de Malware

De todos los modelos de clasificación disponibles, se seleccionaron los siguientes:

- *K-Nearest Neighbors*
- *Random Forest*
- *XGboost*
- *Red Neuronal*

Todos ellos serán ejecutados tomando como datos de entrada el *dataset* ya preprocesado. Los resultados serán luego evaluados mediante el valor correspondiente a la precisión gracias a la libreria `scikit-learn`, la matriz de confusión y la gráfica ROC (*Receiver Operating Characteristic*).

K-Nearest Neighbors En una primera instancia se realizó una implementación del algoritmo *K-Nearest Neighbors* (conocido también como KNN), clasifica cada punto mediante el análisis de sus vecinos más cercanos dentro del conjunto de entrenamiento. El punto es asignado a la clase más común que es encontrada entre dichos vecinos. Es un algoritmo no paramétrico, por lo que no realiza asunciones acerca de cómo los datos están distribuidos.

En dicha implementación, gracias a la utilización del análisis de componentes del vecindario, hemos logrado una precisión en la predicción del 98.98%, contra 95.34% de no haberla utilizado. Se debe recordar que la **precisión** es una de las métricas más comúnes utilizadas para medir la performance del modelo de clasificación. Ésta nos permitirá identificar rápidamente la proporción de aciertos que obtuvo nuestro modelo.

5.1 *Random Forest*

A continuación se realizó la implementación de un algoritmo *Random Forest*. *Random Forest* se compone de un ensamble de varios árboles de decisión que utiliza *bootstrapping* (selección aleatoria de un conjunto de observaciones con reemplazo), varios subconjuntos aleatorios del *dataset* cuando considera la separación de cada nodo en un árbol de decisión, y el voto promedio para mejorar la precisión de la predicción y controlar el sobreajuste (*overfitting*).

En este caso la precisión alcanzada fue del 98.80%.

5.2 *XGBoost*

XGBoost[4] o *Extreme Gradient Boosting* es una de las implementaciones de algoritmos predictivos supervisados más utilizados en la actualidad.

XGBoost utiliza el principio de *boosting*. La idea del *booting* es generar varios modelos de predicción "débiles" secuencialmente, con el fin de generar un modelo más "fuerte", con mayor poder predictivo y mayor estabilidad en sus resultados. Para lograr esto, el modelo emplea un algoritmo de optimización denominado *Gradient Descent* (descenso del gradiente).

Cada uno de estos modelos tomará los resultados del modelo anterior y los comparaá. Si el nuevo modelo tiene mejores resultados, entonces se utilizará como base para realizar modificaciones. Si, en cambio, tiene peores resultados, se regresa al mejor modelo anterior y el mismo será modificado de una manera diferente. Este proceso es iterativo y se repetirá hasta un punto en el que la diferencia entre los modelos consecutivos sea insignificante, lo que indicaría que se llegó al mejor modelo posible, o cuando se llega al número de iteraciones máximas definidas por el usuario.

Con la utilización de *XGBoost* se logró una precisión de 98.65% para la clasificación.

5.3 *Artificial Neural Networks*

Tanto las Redes Neuronales Artificiales (*Artificial Neural Network*), como las Redes Neuronales Profundas (*Deep Neural Network*) pueden ser utilizadas para realizar tareas de clasificación y obtener muy buenos resultados. Estas redes han tomado su inspiración en el proceso de aprendizaje que ocurre en el cerebro humano. Se componen de una red de funciones llamadas parámetros, que le permitirán a la red aprender, los cuales, a su vez, se pueden ajustar (*tuning*) a si mismos mediante el análisis de los datos. Cada uno de estos parámetros, también conocidos como neuronas, es una funcioón que produce una salida luego de haber recibido una o más entradas. Luego, estas salidas se pasarán a la siguiente capa de neuronas, la cual las utilizará como entrada en su función y generará su propia salida. Estas nuevas salidas se enviarán a la siguiente capa y así el proceso continuará sucesivamente hasta haber considerado todas las neuronas que conforman la red y las neuronas terminales hayan recibido su entrada. Las salidas de estas neuronas terminales será el resultado final del modelo.

La precisión alcanzada por el modelo propuesto de la *Artificial Neural Network* fue de un 99.06%.

5.4 Comparaciones y Conclusiones

Habiendo obtenidos en casi todos los casos precisiones cercanas al 99%, puede decirse que todas las implementaciones han resultados ser opciones más que

[4] `https://xgboost.ai/`

válidas y que pueden ser tenidas en cuenta para la clasificación de las diferentes familias de *malware*, haciendo que no sea necesario llevar a cabo una costosa implementación de una red neuronal para clasificar las muestras, ya que es posible obtener prácticamente los mismos resultados con cualquiera de los otros algoritmos de clasificación mencionados en esta investigación.

6 Detección de Malware

El problema de la detección de *malware* puede ser visto como uno de clasificación binaria, en donde lo que se debe determinar es si dado un archivo no conocido por el algoritmo, éste es capaz de determinar si el mismo es detectado como *malware*. Por lo tanto, será necesario contar con muestras de aplicaciones consideradas benignas.

De los sitios *CNET*[5] y *SourceForge*[6], se descargaron un total de 215 aplicaciones livianas. Una vez obtenidas todas estas aplicaciones, se procedió a realizarles el desensamblado. Para ello se utilizó la aplicación *Interactive Disassembler*[7], más conocida por su acrónimo IDA. Este desensamblador es generalmente utilizado para realizar ingeniería inversa sobre los ejecutables y, de este modo, poder convertir una aplicación en un archivo `ASM`, y de ser necesario uno `bytes`. Estos formatos son los que se requerirán para comenzar el proceso de extracción de la información.

Junto a las muestras correspondientes a archivos benignos, se seleccionaron un número igual de *malwares* provenientes del *dataset* con las nueve familias.

6.1 Generación del nuevo *dataset*

De forma similar al trabajo realizado para la clasificación de familias, todos lo pasos correspondientes a la extracción de atributos debieron ser realizados nuevamente, para generar un nuevo *dataset*.

6.2 Análisis Exploratorio y Preprocesamiento

Se estudió nuevamente la correlación *Pearson* y, análogo a lo sucedido en la clasificación de familias, se pueden observar las correlaciones más altas para los *n-gramas* similares, como por ejemplo `mov_sub_mov` y `sub_mov`.

[5] `https://download.cnet.com/`
[6] `https://sourceforge.net/`
[7] `https://www.hex-rays.com/products/ida/support/download_freeware/`

Por otro lado, el preprocesamiento consistió en el tratamiento de los datos nulos o faltantes, determinar la importancia de las variables, estandarizar los datos y por último la selección y extracción de variables aplicando los mismos criterios y pasos aplicados para la clasificación de familias.

6.3 Implementación de una solución

El objetivo de esta implementación consiste en poder determinar si un archivo puede o no ser clasificado como *malware* y con qué grado de precisión. Para ello se optó por llevar a cabo la implementación de una *Artifical Neural Network*, ya que la ésta fue la que mejor desempeño obtuvo para la clasificación de las familias de *malware*.

Modelo base

Para ello, se comenzó implementando la red neuronal respetando la misma configuración que se utilizó para la clasificación de familias, pero esta vez la clasificación sería binaria (*malware/no malware*). Esta red no logró un buen resultado, ya que su precisión fue apenas superior al 50%. Sin embargo, pudo observarse cómo el modelo está teniendo una precisión mayor al 90% durante el entrenamiento, mientras que con los datos de prueba obtiene valores inferiores al 60%, por lo que podría ser un claro indicio de que el modelo está sobreajustando (*overfitting*) los valores a los datos de entrenamiento.

Tratamiento del sobreajuste (*overfitting*)

Como se mencionó previamente, el modelo está sobreajustando los valores a los datos de entrenamiento. Existen varias alternativas que se pueden llevar a cabo para minimizar, e incluso eliminar, este problema. Por lo que fue necesario la aplicación de algunas de estas técnicas para intentar mitigar dicho problema.

Resultados obtenidos

Una vez obtenido un modelo con los parámetros optimizados, se realizaron nuevamente las pruebas para la clasificación binaria. Esta vez, el modelo obtuvo una precisión apenas superior al **68.6%**. Si bien se logró una mejora sustancial respecto al modelo base (alrededor del 40%), los resultados siguen sin ser buenos.

6.4 *XGBoost* como solución alternativa

En la sección anterior se vio como la implementación de la red neuronal no tuvo un buen desempeño realizando la clasificación binaria, dado que no fue posible reducir completamente el *overfitting*. A continuación, se utilizará un algoritmo de ensamble *XGBoost*, ya que estos, al igual que los *Random Forest* suelen realizar muy buenas generalizaciones y de este modo evitar el problema del *overfitting*.

Modelado

Al igual que en caso de la red neuronal, el modelo base del *XGBoost* tuvo que ser ajustado en sus hiperparámetros, ya que tampoco logró alcanzar una buena precisión.

La implementación de este modelo ha logrado cierta mejora respecto a la red neuronal artificial, alcanzando una precisión del **72.86%**.

7 Comparaciones y Conclusiones

Se comenzó implementando una red neuronal muy similar a la utilizada para la clasificación de familias de *malware*. Esta red, en su modelo base, no arrojó buenos resultados, lo que nos motivó a intentar ajustar sus hiperparámetros y determinar si de esta manera es posible lograr una mejora del modelo en sus predicciones. Para ello, se implementaron varias estrategias que ayudarían a dicho modelo a lograr un mejor desempeño. Este proceso se realizó combinando de diversas maneras los distintos parámetros del modelo, para luego determinar, mediante el uso de alguna métrica, la opción que produce mejores resultados.

Habiendo hecho esto, se modificó el modelo inicial para incorporar estos parámetros y se procedió a correr nuevamente las pruebas de clasificación. Si bien se logró cierta mejora en la precisión y reducir el *overfitting*, su resultado no alcanzó valores aceptables.

Por otro lado, como una alternativa a la red neuronal, se llevó a cabo la implementación de un *XGBoost*, y de este modo evaluar si es posible mejorar la precisión de la predicción alcanzada por la red neuronal. Este modelo logró cierta mejora respecto a la red, con una precisión superior al 72%, sin embargo su valor aún sigue siendo bajo.

Se identificaron al menos tres problemas que pueden impactar negativamente en la performance de ambos modelos: por un lado, no es posible garantizar que los *samples* benignos sean realmente benignos. Esto se debe a que los archivos considerados benignos fueron descargados de sitios que no pueden dar fe de su legitimidad. Por otro lado se encuentra la heterogeneidad de los archivos, siendo

estos muy dispares entre sí. Los programas pueden variar desde un *plugin*, hasta un editor de texto. Por último, cabe mencionar el tamaño del *dataset* utilizado, ya que sólo se dispuso de un total de 430 *samples*, mientras que la red de clasificación de familias contaba con un total superior a 10000 *samples*.

References

1. J. Gareth, D. Witten, T. Hastie and R. Tibshirani, An Introduction to Statistical Learning, Springer, New York, 2014
2. M. Sikorski and A. Honig, Practical Malware Analysis, No Starch Press, San Francisco, 2012
3. Mark Stamp, Introduction to Machine Learning with Applications in Information Security, CRC Press, Boca Raton, Florida, 2018
4. T. Mitchell, Machine Learning, McGraw Hill, 1997
5. H. Liu and M. Cocea, Granular Computing based Machine Learning, Springer, Suiza, 2018
6. Dipanjan Sarkar, Raghav Bali, Tushar Sharma, Practical Machine Learning with Python, Apress, Bangalore, Karnataka, India, 2018
7. Tom Hope, Yehezkel S. Resheff e Itay Lieder, Learning Tensorflow, O'Reilly, Sebastopol, California, Estados Unidos, 2017
8. Jake VanderPlas, Python Data Science Handbook, O'Reilly, Sebastopol, California, Estados Unidos, 2016
9. Matthew Kirk, Thoughtful Machine Learning with Python, O'Reilly, Sebastopol, California, Estados Unidos, 2016
10. Summet Dua y Xian Du, Data Mining and Machine Learning in Cybersecurity, CRC Press, Boca Raton, Florida, Estados Unidos, 2011
11. José Unpingco, Python for Probability, Statistics, and Machine Learning, Springer, San Diego, California, Estados Unidos, 2016
12. Christopher C. Elisan, Advanced Malware Analysis, Mc Graw Hill Education, Estados Unidos, 2015
13. Stuart j. Russell y Peter Norvig, Artificial Intelligent: A Modern Approach, Thrid Edition, Pearson, Inglaterra, 2010
14. Ian H.Witten, Eibe Frank, Mark A. Hall y Christopher j. Pal, Data Mining. Practical Machine Learning Tools and Techniques, Morgan Kaufmann, Cambridge, Miami, Estados Unidos, 2017
15. University of Virginia Computer Science, `http://www.cs.virginia.edu/~evans/cs216/guides/x86.html`. Último acceso Oct 2020
16. Scikit-learn, `https://scikit-learn.org/stable/tutorial/index.html`. Último acceso Oct 2020
17. Tensorflow, `https://www.tensorflow.org/tutorials`. Último acceso Oct 2020