UNIVERSIDAD DE BUENOS AIRES

Facultades de Ciencias Económicas,
Ciencias Exactas y Naturales e Ingeniería

Maestría en Seguridad Informática

Tesis de Maestría

Técnicas de Inteligencia Artificial aplicadas a la SI:
Un enfoque actual

Autor:

Lic. Khalil Alejandro Moriello

Tutor:

Dr. Hugo Scolnik

Noviembre de 2023 – Cohorte 2020

Declaración Jurada de origen de los contenidos

Por medio de la presente, el autor manifiesta conocer y aceptar el Reglamento de Trabajos Finales vigente y se hace responsable que la totalidad de los contenidos del presente documento son originales y de su creación exclusiva, o bien pertenecen a terceros u otras fuentes, que han sido adecuadamente referenciados y cuya inclusión no infringe la legislación Nacional e Internacional de Propiedad Intelectual.

FIRMADO

Khalil Alejandro Moriello

DNI 35375402

Resumen

Hoy en día es sabido que los diseñadores de tecnologías para prevenir ataques informáticos están en constante batalla contra los atacantes. Esto se debe a que las técnicas utilizadas por estos últimos se complejizan y mejoran muchas veces más rápido que las soluciones. Es un aporte interesante estudiar cómo aplicar inteligencia artificial para reducir los tiempos entre el ataque y la detección.

El objetivo de este trabajo final de Maestría es proponer un tipo de red neuronal basada en grafos que permita la detección de malware. Así también, se realizará un estudio de tipo descriptivo: se analizarán los tipos y las características de las distintas soluciones existentes. Tanto basadas en el análisis dinámico como en enfoques de *Deep learning.*

Palabras clave:

- Control Flow Graph (CFG)
- Ciberseguridad
- Detección de malware
- Redes neuronales
- Deep learning

Índice

Índice de Ilustraciones

Índice de Tablas

Cuerpo introductorio

En el presente trabajo se presentarán dos enfoques existentes utilizando técnicas de *machine learning* enfocadas al análisis dinámico y otro enfoque utilizando *deep learning*. Por último, se propondrá una mejora sobre un método previamente estudiado.

Para el análisis dinámico de malware se muestran las distintas líneas de investigación actuales sin mostrar resultados con pruebas reales. Se presentan y se describen varias técnicas de *machine learning* enfocadas al análisis dinámico. El objetivo de dicha mención es brindarle al lector una breve introducción sobre las distintas técnicas y formulaciones de los algoritmos.

Lo mismo aplica para el enfoque utilizando *deep learning*.

Por último, se presenta un cambio sobre una idea propuesta en la bibliografía y se estudia las diferencias.

Capítulo 1: Conceptos preliminares

1.1 Cadenas de Markov (Markov Chains)

Sección tomada de [1].

Las cadenas de Markov son un tipo de modelo matemático que describe una secuencia de eventos en la que la probabilidad de pasar de un estado a otro depende únicamente del estado actual y no de cómo se llegó a ese estado. Se utilizan en una variedad de campos, como la teoría de juegos, la modelización del comportamiento de sistemas dinámicos, la economía, la biología, y más. Son especialmente útiles para modelar situaciones en las que las transiciones entre estados son aleatorias o están sujetas a cierto grado de incertidumbre.

Formalmente, una cadena de Markov se define mediante un conjunto finito de estados y una matriz de transición que especifica las probabilidades de pasar de un estado a otro.

Denomínese a la variable de estado q_t como el estado del sistema en un tiempo discreto t. La Cadena de Markov está definida por la matriz de estados de transición $A = [a_{ij}]$, donde

$$a_{ij} = Pr(q_t = j | q_{t-1} = i), \qquad 1 \leq i,j \leq N \tag{1}$$

con las siguientes restricciones:

$$a_{ij} \geq 0 \tag{2}$$

y

$$\sum_{j=1}^{N} a_{ij} = 1; \ \forall \, i \tag{3}$$

En otras palabras, lo que representa la ecuación (1) es que las probabilidades de las transiciones no dependen del tiempo. Además, la probabilidad de estar en un estado j depende únicamente del estado previo y no sobre lo que ocurrió con anterioridad.

Asúmase que en $t = 0$ el estado del sistema q_0 está indicado por la probabilidad de estado inicial $\pi_i = Pr(q_0 = i)$. Entonces, para cualquier

secuencia de estados $q = (q_0, q_1, q_2, \ldots, q_T)$, la probabilidad de q siendo generado por la Cadena de Markov es

$$Pr(\boldsymbol{q}, A, \pi) = \pi_{q_0} a_{q_0 q_1} a_{q_1 q_2} \ldots a_{q_{T-1} q_r} \quad (4)$$

1.2 Modelos de Markov Ocultas (Hidden Markov Models - HMM)

Sección tomada de [1].

Supóngase que la secuencia de estados q no puede ser directamente observada. En su lugar, se visualiza cada observación O_t.

Se asume que la producción de O_t en cada estado posible i ($i = 1,2,\ldots,N$) es estocástico y es caracterizado por un conjunto de medidas de probabilidades de observación $B = \{b_i(O_t)\}_{i=1}^{N}$ donde

$$b_i(O_t) = Pr(O_t | q_t = i) \quad (5)$$

Si el estado de la secuencia q que condujo a la secuencia de observación $O = (O_1, O_2, \ldots, O_T)$ conocida. La probabilidad de O siendo generada por el sistema se asume que es

$$Pr(O|q, B) = b_{q_1}(O_1) b_{q_2}(O_2) \ldots b_{q_T}(O_T) \quad (6)$$

La probabilidad conjunta de O y q siendo producida por el sistema es simplemente el producto de (4) y (6), escrita como

$$Pr(O, q|\pi, A, B) = \pi_{q_0} \prod_{t=1}^{T} a_{q_{t-1} q_t} b_{q_t}(O_t) \quad (7)$$

Entonces, se sigue con que el proceso estocástico viene dado por la ecuación

$$Pr(O|\pi, A, B) = \sum_{q} Pr(O, q|\pi, A, B) = \sum_{q} \pi_{q_0} \prod_{t=1}^{T} a_{q_{t-1}q_t} b_{q_t}(O_t) \quad (8)$$

La ecuación anterior describe la probabilidad de O siendo producida por el sistema sin asumir el conocimiento de la secuencia de estados de la cual fue generada. La tupla $\lambda = (\pi, A, B)$ define el HMM.

El HMM tradicionalmente se asume invariante con respecto al tiempo. Esto implica que las matrices de transición y observación no son dependientes del tiempo t.

La principal diferencia entre HMM para la clasificación y los algoritmos de ML es que los primeros apuntan a clasificar instancias basadas en relaciones temporales y pueden ofrecer una perspectiva diferente sobre los dominios del problema.

Para fines de clasificación o reconocimiento, el objetivo es clasificar nuevas secuencias no vistas de símbolos observados.

Con este fin, se aprende un HMM único por categoría usando un conjunto de entrenamiento compilado de secuencias asociadas a una clase dada. Para un dominio del problema que consta de N clases, la nueva secuencia de observación se asigna al HMM que mejor describe la secuencia de símbolos de observación de los N HMM disponibles.

Si λ_j describe los parámetros que definen a cada uno de los N modelos, donde $j = 1, \dots, N$, y $\lambda_j = \{A_j, B_j, \pi_j\}$

Cuando una nueva observación de secuencia de símbolos se obtiene, es decir, $Y = \{Y_1, \dots, Y_T\}$, entonces, $Pr(\lambda_j|Y)$ se calcula para cada HMM y el HMM descripto por λ_{N^*} se selecciona de manera tal que

$$N^* = \underset{j}{\operatorname{argmax}}(Pr(\lambda_j|Y)) \quad (9)$$

Básicamente, se obtiene aquel HMM más probable que haya generado la secuencia.

La clasificación consiste en dos problemas diferentes. Primero, el usuario debe aprender el conjunto de parámetros λ_j para cada HMM. Luego, para cada nueva secuencia, evaluar la probabilidad para cada posible HMM.

Durante la fase de entrenamiento un HMM único debe ser aprendido para cada clase, donde el HMM es más probable que haya generado la secuencia observada.

El aprendizaje consiste en optimizar el conjunto de parámetros del modelo $\lambda = \{A, B, \pi\}$ para maximizar la probabilidad de la secuencia observada. Se utiliza un caso especial del algoritmo de *Expectation-Maximization* (EM) conocido como algoritmo de Baum-Welch. [2]

1.3 Minería OOA

Sección tomada de [3] y [4]

En el método tradicional el objetivo es descubrir cómo un conjunto de elementos está estadísticamente correlacionado mediante la extracción de reglas de asociación de la forma $\{i_1, \ldots, i_m\} \rightarrow i_{m+1}(\%s, \%c)$, donde $s\%$ (el soporte de la regla) es la probabilidad de que todos los elementos $i_1, \ldots, i_m$ ocurran juntos, y $c\%$ (la confianza de la regla) es la probabilidad condicional de i_{m+1} dado el conjunto de elementos $\{i_1, \ldots, i_m\}$. Tanto el valor de $s\%$ como el valor de $c\%$ se calculan mediante la frecuencia de los conjuntos de elementos correspondientes en un *dataet* dado DB. Estos valores son mayores o iguales que el soporte mínimo especificado por el usuario (ms) y la confianza mínima (mc), respectivamente.

Por otro lado, la minería OOA es un algoritmo típico de minería de datos que integra técnicas de clasificación y minería de reglas de asociación. Además, modela patrones de asociación que están explícitamente relacionados con un objetivo dado. Cada regla de asociación se define de la siguiente forma: $\{i_1, \ldots, i_m\} \rightarrow obj(\%s, \%c, u)$ donde obj es un objetivo; $s\%$ es la probabilidad de que todos los elementos $\{i_1, \ldots, i_m\}$ junto con obj se cumplan; $c\%$ es la probabilidad condicional de obj dado el conjunto de elementos $\{i_1, \ldots, i_m\}$; y u es la utilidad de la regla, que muestra en qué grado el patrón $\{i_1, \ldots, i_m\}$ respalda semánticamente a obj. Se puede describir de la siguiente manera:

Se asocia una base de datos DB con un conjunto finito de atributos, DB_{att}. Cada atributo A_i tiene un dominio finito V_i. Para cada v en V_i, se llama un elemento $A_i = v$. Un k-conjunto de elementos es un conjunto de elementos con k elementos. DB consiste en un conjunto finito de registros/transacciones

construidas a partir de DB_{att}, donde cada registro es un conjunto $\{A_1 = v_1, \ldots, A_m = v_m\}$ de elementos donde $A_i \neq A_j$, para cualquier $i \neq j$.

Un objetivo describe cualquier cosa que se quiera lograr. En la minería OOA, se dice que un objetivo obj se cumple en un registro r en DB (o se dice que r respalda obj) si obj es verdadero dado r. Además, para cualquier conjunto de elementos $I = \{i_1, \ldots, i_m\}$, se dice que $I \cup \{obj\} = \{i_1, \ldots, i_m, obj\}$ se cumple en r si tanto obj como todos los elementos en I son verdaderos en r. DB_{att} se puede dividir en dos subconjuntos disjuntos: DB_{att} = $DB_{obj-att} \cup DB_{nobj-att}$. Los atributos en $DB_{obj-att}$ se utilizan para definir obj, mientras que los atributos en $DB_{nobj-att}$ se utilizan para definir los elementos en I. Por conveniencia, los atributos en $DB_{obj-att}$ se refieren como atributos objetivos.

Definición 1: Sea $\{i_1, \ldots, i_m\} \rightarrow obj(\%s, \%c, u)$ una regla de asociación en la minería OOA. El soporte y la confianza se calculan:

$$s\% = \frac{count(\{i_1, \ldots, i_m, obj\}, DB)}{|DB|} \times 100\% \quad (10)$$

$$c\% = \frac{count(\{i_1, \ldots, i_m, obj\}, DB)}{count(\{i_1, \ldots, i_m\}, DB)} \times 100\% \quad (11)$$

donde $|DB|$ es el total de registros en DB. La función $count(*, DB)$ devuelve el número de registros en DB que son superconjuntos de $*$.

Definición 2: Sea A un atributo objetivo y V un dominio para A. Para cada $v \in V$, se denomina elemento objetivo o clase de A a $A = v$. Se utiliza $class(A)$ para denotar todas las clases de A. Basándose en obj, las clases de A pueden ser clasificadas subjetivamente en dos grupos disjuntos: $class(A) = class^+(A) \cup class^-(A)$; donde $class^+(A)$ consiste en todas las clases de A que muestran un apoyo positivo para obj, y $class^-(A)$ consiste en todas las clases de A que muestran un apoyo negativo para obj. Cada clase A = v en $class^+(A)$ o $class^-(A)$ se asocia con una utilidad $u_{A=v}$. Normalmente, la utilidad $u_{A=v}$es subjetiva y puede ser adquirida de un experto en el dominio.

Los grupos de clases que apoyan positivamente y negativamente un conjunto de datos DB para obj se definen respectivamente de la siguiente manera: $class^{+}(A) = \{A = v(u_{A=v}) | A \in DB_{ob} \quad \wedge A = v \in class^{+}(A)\}$ y $class^{-}(A) = \{A = v(u_{A=v}) | A \in DB_{obj-att} \wedge A = v \in class^{-}(A)\}$. Se define C_r como el conjunto de clases en r.

La utilidad positiva $u_{+r}(I), I \subseteq r$ (respectivamente la utilidad negativa $u_{-r}(I), c$) es la suma de las utilidades de todas las clases de apoyo positivo (o negativas) en C_r, que viene dado por:

$$u_{+r}(I) = \sum_{A=v \in C_r \wedge A=v \in class^{+}(A)} u_{A=v} \tag{12}$$

$$u_{-r}(I) = \sum_{A=v \in C_r \wedge A=v \in class^{-}(A)} u_{A=v} \tag{13}$$

La utilidad positiva y negativa de DB para I son:

$$u_{+DB}(I) = \sum_{r \in DB \wedge I \subseteq r} u_{+r}(I) \tag{14}$$

$$u_{-DB}(I) = \sum_{r \in DB \wedge I \subseteq r} u_{-r}(I) \tag{15}$$

Defínase $u_{DB}(I) = u_{+DB}(I) - u_{-DB}(I)$; entonces, la utilidad de la regla viene dada por

$$u = \frac{u_{DB}(I)}{count(\{i_1, \ldots, i_m\}, DB)} \times 100\% \tag{16}$$

OOA extrae las reglas cuyo soporte ($s\%$), confianza ($c\%$) y utilidad (u) son mayores que un mínimo especificado por el usuario: soporte mínimo (ms), confianza mínima (mc) y utilidad mínima (mu), respectivamente.

Los métodos comunes de clasificación basados en la extracción de reglas de asociación incluyen: clasificación basada en asociación (CBA), clasificación basada en múltiples reglas de asociación (CMAR) y clasificación basada en

reglas de asociación predictivas (CPAR). Los tres clasificadores utilizan diferentes métodos para extraer conjuntos de ítems frecuentes y emplean diferentes estrategias para utilizar las reglas extraídas en la clasificación.

CBA utiliza un algoritmo *apriori* para extraer conjuntos de ítems frecuentes. Primero genera todas las reglas de asociación utilizando dicho algoritmo. Luego, selecciona un conjunto reducido de reglas para formar un clasificador. Al clasificar una muestra, se elige la mejor regla (es decir, la regla con la mayor confianza) cuyo cuerpo se cumple en la muestra para la predicción. Extrae reglas utilizando el algoritmo de Crecimiento de Patrones Frecuentes.

Al predecir la etiqueta de clase de una muestra, CMAR selecciona un conjunto reducido de reglas con alta confianza y altamente relacionadas, y analiza la correlación entre esas reglas.

CPAR adopta la estrategia FOIL (*First Order Inductive Learner*) para generar las reglas. FOIL es un algoritmo *greedy* que aprende reglas para distinguir ejemplos positivos de ejemplos negativos. El conjunto de reglas predictivas se genera directamente a partir del conjunto de datos mediante predicción de reglas y análisis de cobertura. OOApriori es un tipo de algoritmo a*priori*; por lo tanto, en la minería tradicional de OOA se adopta CBA para clasificar objetos.

1.4 Técnicas de ofuscación

Sección tomada de [5].

1.4.1 Empaquetadores (*packers*)

Originalmente fueron diseñados para minimizar la memoria y el uso del ancho de banda durante el almacenamiento y transferencia.

La desventaja es que una pequeña modificación en el archivo genera un gran cambio en el archivo comprimido. Esta encripción inherente les provee a los desarrolladores de malware una herramienta poderosa que utilizan para evadir la detección de sus códigos maliciosos.

Algunas herramientas de antimalware calculan la entropía para determinar si está empaquetado o no. Este método no identifica al empaquetador en sí.

1.4.2 Polimorfismo

El polimorfismo es una técnica de encripción que muta el código binario estático, en lugar del código binario dinámico, para evadir la detección mediante las firmas. Aquel malware que cambie su contenido puede eludir la detección pues no existirá una firma unívoca que lo identifique.

Cada vez que el código se ejecute se mutará a sí mismo utilizando diferentes claves de encripción cada vez que copie. Esto hará que cada una de las copias tenga una firma complemente diferente.

Cuando el *engine* polimórfico descifre y cargue el malware en memoria, los *opcodes* van a ser semánticamente iguales para todas las instancias. Esto implica que el *engine* polimórfico no cambia significativamente los *opcodes* que se ejecutan en memoria. Por ende, es posible utilizar la detección de firmas en la memoria cuando el malware objetivo se esté ejecutando.

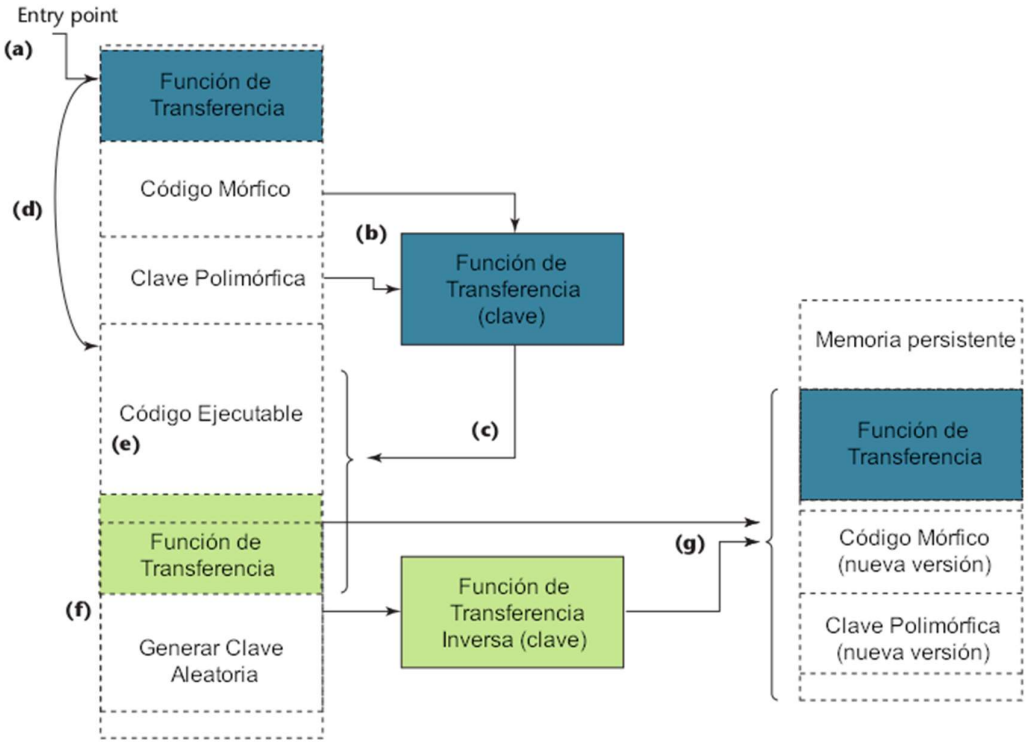

Ilustración 1: *Engine polimórfico simplicado*

Explicación de la ilustración 1:

(a) Punto de entrada del malware (*entry point*)

(b) La función de transferencia del motor descifra el código de malware mutado en el host infectado en *opcode* nativo
(c) El motor escribe ese *opcode* en la memoria.
(d) El malware se ejecuta.
(e) El malware ejecuta una actividad como robar datos personales.
(f) El motor genera una nueva clave.
(g) La función de transferencia inversa del motor transforma el *opcode* nuevamente en código mutado.

1.4.3 Metamorfismo

Cada vez que se ejecuta el malware metamórfico los *opcodes* que se cargan en la memoria se modifican. Luego, la nueva versión se escribe en el disco del objetivo. El malware mantiene su comportamiento malicioso sin tener la misma secuencia de *opcodes* en la memoria.

Existen dos clasificaciones para esta categoría:

- *Open-world malware*: se pueden comunicar con otros sitios de Internet y descargar actualizaciones.
- *Closed-world malware*: no dependen de actualizaciones online para mutar. Durante su evolución, el malware muta su propio código binario o usa una representación en pseudocódigo (metalenguaje) para generar el nuevo código mutado.

Para ocultar este tipo de malware, las técnicas de ofuscación pueden ser de las siguientes clasificaciones:

- ***Garbage insertion:*** inserta instrucciones benignas tales como `nop` y `clc`.
- ***Code substitution:*** cambia los *opcodes* a *opcodes* equivalentes que logran lo mismo.
- ***Code insertion:*** entrelaza el malware con el código binario de su host.
- ***Register swapping:*** sustituye un registro por otro en el código del malware.
- ***Changing flow control:*** salta y reordena la secuencia de llamadas agregando subrutinas.

Este tipo de malware es más complejo que los otros tipos e involucra hasta cinco pasos:

1. El desensamblador decodifica los *opcodes*.
2. Un compresor comprime el código desensamblado que previene el continuo crecimiento en cada generación.
3. Un permutador reordena las instrucciones poniéndolas en un orden aleatorio. Las conecta por medio de saltos (`jmp`) e implementa sustitución de *opcodes* y registros.
4. Un expansor recodifica un único *opcode* en varias instrucciones e inserta *opcodes* benignos. Esto sucede cada vez que el malware se ejecuta para generar nuevas instancias.
5. Un ensamblador recodifica lo que el expansor produjo y calcula los *offsets* de los saltos.

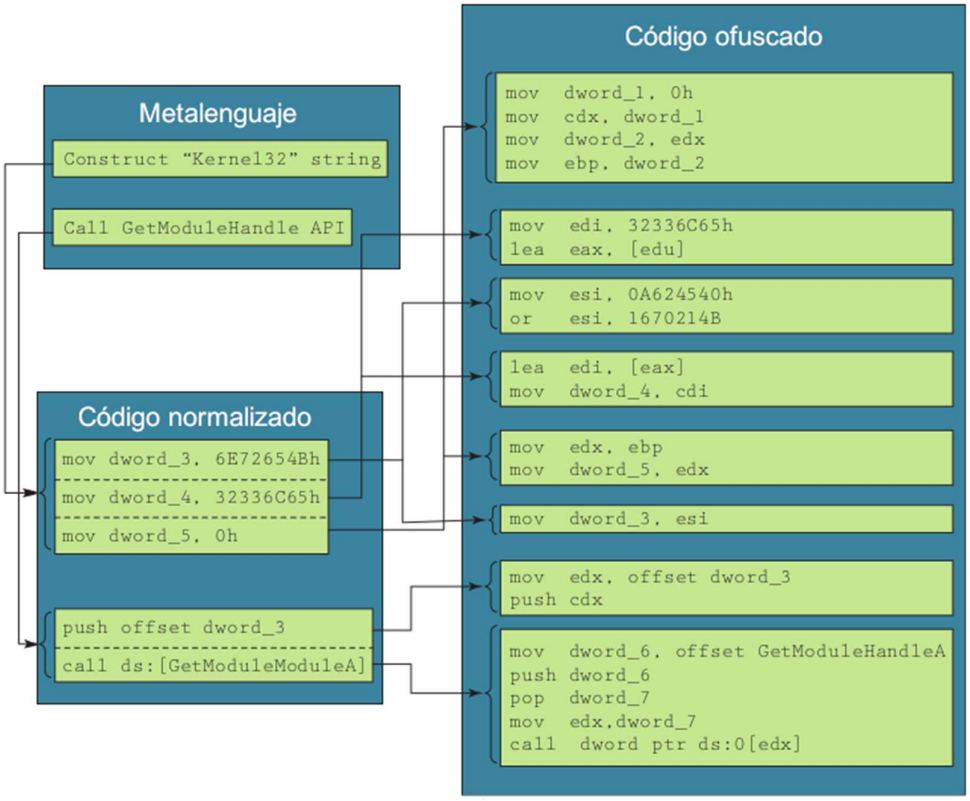

Ilustración 2: *Ofuscación de código metamórfico*

1.5 Redes neuronales

Sección tomada de [6].

Los perceptrones multicapa (MLP) son posiblemente una de las redes neuronales más visualmente ilustradas. Sin embargo, la mayoría de ellas carecen de algunas explicaciones fundamentales. Dado que los MLP son la base del *deep learning*, esta sección intenta proporcionar una perspectiva más clara.

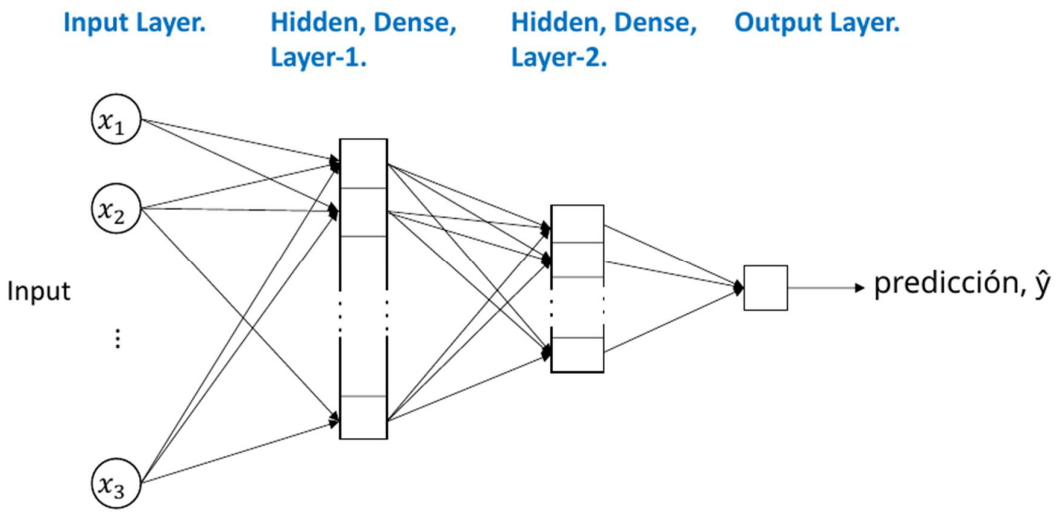

Ilustración 3: *Representación típica de MLP*

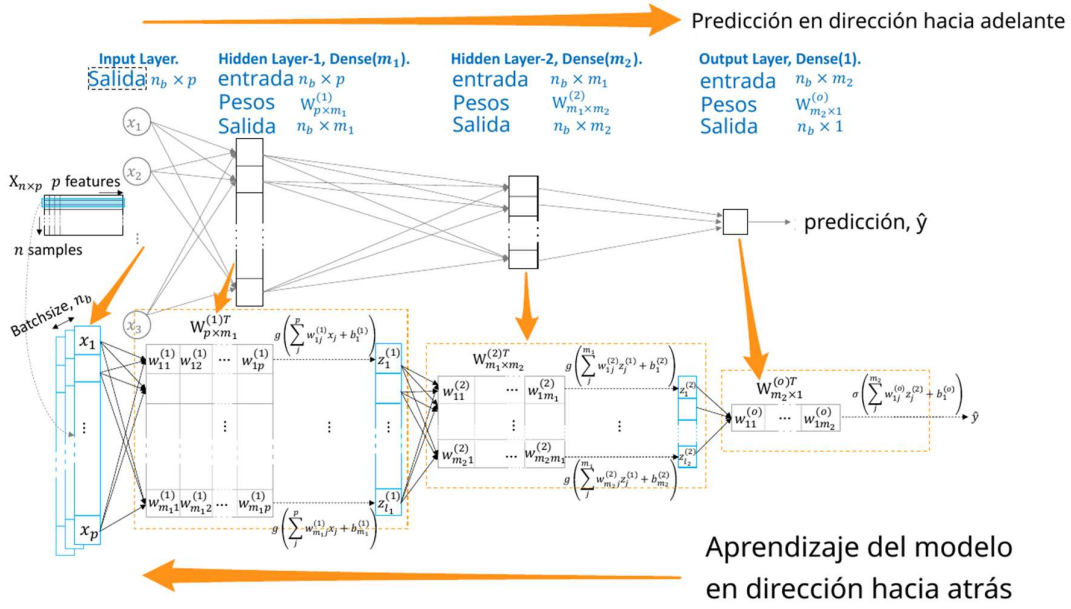

Ilustración 4: *Aprendizaje de una red neuronal*

Una representación típica de un MLP se muestra en la Ilustración 3. Esta representación de alto nivel muestra la naturaleza de avance de la red. En una red de avance, la información entre capas fluye solo hacia adelante. Es decir, la información (*features*) aprendida en una capa no se comparte con ninguna capa anterior.

1. El proceso comienza con un *dataset*. Supóngase que hay un *dataset*, $X_{n\times p} \in \mathbb{R}^{n\times p}$, con n muestras y p *features*.

2. El modelo ingiere un *batch* seleccionado al azar durante el entrenamiento. El lote contiene muestras aleatorias (filas) de X a menos que se indique lo contrario. El tamaño del *batch* se denota como n_b aquí.

3. Las muestras en un *batch* se procesan de forma independiente. Por lo tanto, el orden no es importante.

4. El *batch* de entrada ingresa a la red a través de una capa de entrada. Cada nodo en la capa de entrada corresponde a una *feature* de muestra. Definir explícitamente la capa de entrada es opcional, pero se hace aquí para mayor claridad.

5. La capa de entrada es seguida por una serie de capas ocultas hasta la última (capa de salida). Estas capas realizan operaciones no lineales "complejas". Aunque se perciben como "complejas", las operaciones subyacentes son cálculos aritméticos bastante simples.

6. Una capa oculta es un conjunto de nodos. Cada nodo extrae una *feature* de la entrada. Un nodo, por lo tanto, puede imaginarse como que estuviese resolviendo un problema arbitrario.

7. El conjunto de salidas que provienen de los nodos de una capa se llama mapa de *features* o representación. El tamaño del mapa de *features*, igual al número de nodos, se llama tamaño de la capa.

8. Intuitivamente, este mapa de *features* tiene resultados de varios "subproblemas" resueltos en cada nodo. Proporcionan información predictiva para la siguiente capa hasta la capa de salida para predecir en última instancia la respuesta.

9. Matemáticamente, un nodo es un perceptrón compuesto por pesos y parámetros de sesgo. Los pesos en un nodo se denotan con un vector w y un sesgo b.

10. Todas las *features* de una muestra de entrada van a un nodo. La entrada a la primera capa oculta es la *feature* de datos de entrada $\boldsymbol{x} = \{x_1, \dots, x_n\}$ Para cualquier capa intermedia, es la salida (mapa de *features*) de la capa anterior, denotada como $\boldsymbol{z} = \{z_1, \dots, z_m\}$, donde m es el tamaño de la capa anterior.

11. Considérese una capa oculta l de tamaño m_l. Un nodo j en la capa l realiza una extracción de *features* con un producto interno entre el mapa de *features* de entrada $z^{(l-1)}$ y sus pesos $w_l^{(l)}$, seguido de una adición con el sesgo b_j. Generalizando esto como:

$$z_j^{(l)} = \sum_i^{m_{l-1}} w_{ij}^{(l)} z_i^{(l-1)} + b_j, \qquad j = 1, \dots, m_l \tag{17}$$

donde $z_i^{(l-1)}, i = 1, \dots, m_{l-1}$ es una *feature* producida por la capa anterior $l-1$ de tamaño m_{l-1}.

12. El paso después de la operación lineal en (17) es aplicar una función de activación no lineal, denotada como g. Hay varias opciones para g.

Entre ellas, una función de activación popular es la unidad lineal rectificada (ReLU) definida como:

$$g(z) = \begin{cases} z, & x > 0 \\ 0, & otro\ caso \end{cases} \quad (18)$$

Como se muestra en (18) g no es lineal en 0.

13. Las operaciones en las ecuaciones (17) y (18) se pueden combinar para cada nodo en una capa, al igual que el sesgo.

$$z^{(l)} = g\left(W^{(l)} z^{(l-1)} + b^{(l)}\right) \quad (19)$$

La salida $Z^{(l)}_{n_b \times m_l}$ de la ecuación es la transformación afín activada por g de las *features* de entrada.

14. La ecuación (19) se aplica al *batch* de n_b muestras de entrada. Para una capa densa, esta es una operación de matriz que se muestra a continuación:

$$Z^{(l)}_{n_b \times m_l} = g\left(W^{(l)}_{m_{l-1} \times m_l} Z^{(l-1)}_{n_b \times m_{l-1}} + b^{(l)}_{m_l}\right) \quad (20)$$

La salida $Z^{(l)}_{n_b \times m_l}$ de la ecuación es la transformación afín activada por g de las *features* de entrada.

15. Es la activación no lineal en (20) la que separa el mapa de *features* de una capa de otra. Sin la activación, el mapa de *features* producido por cada capa sería simplemente una transformación lineal de la anterior. Esto implicaría que las capas subsiguientes no agregarían información adicional para mejorar la predicción.

16. Las funciones de activación, por lo tanto, desempeñan un papel importante. Una selección apropiada de la activación es fundamental. Además de ser no lineal, una función de activación también debe tener

un gradiente no decreciente, una región de saturación (provoca derivadas pequeñas que reducen la varianza y, por lo tanto, la información se propaga de manera efectiva a la siguiente capa) y auto-normalización (si sus medias y varianzas a través de las muestras se encuentran dentro de intervalos predefinidos. Por lo tanto, si las entradas están normalizadas, la salida de cada capa se normalizará automáticamente, resolviendo el problema de explosión del gradiente).

17. La operación en (20) se lleva a cabo en cada capa hasta la capa de salida para realizar una predicción. La salida se entrega a través de una activación diferente denotada como σ. La elección de esta activación está dictada por el tipo de respuesta. Por ejemplo, para una activación binaria se usa la función $sigmoidal$.

18. El entrenamiento del modelo comienza con la inicialización aleatoria de los pesos y sesgos en las capas. Se realiza una predicción de la respuesta utilizando estos parámetros. Luego, el error de predicción se propaga hacia atrás al modelo para actualizar los parámetros a un valor que reduzca el error. Este procedimiento de entrenamiento iterativo se llama *backpropagation*.

19. En resumen, *backpropagation* es una extensión del enfoque basado en el descenso de gradiente estocástico (SGD) iterativo para entrenar redes neuronales profundas de múltiples capas. Esto se explica utilizando un perceptrón de una sola capa, también conocido como regresión logística. La ecuación de estimación en la referencia se reformula en un contexto simplificado aquí:

$$\begin{aligned} \theta^{new} &\leftarrow \theta^{old} - \eta \nabla_{\theta} \\ &\leftarrow \theta^{old} - \eta X^{T}(y - \hat{y}) \end{aligned} \tag{21}$$

donde η es un multiplicador, X es una muestra aleatoria, ∇_{θ} es el gradiente de primer orden de la pérdida con respecto al parámetro θ, y θ son los parámetros de peso y sesgo. Como se muestra, el gradiente ∇_{θ}

para la regresión logística contiene $(y - \hat{y})$, que es el error de predicción. Esto implica que este último se propague hacia atrás para actualizar los parámetros del modelo θ.

20. Este enfoque de estimación para la regresión logística se extiende en la *backpropagation* para un MLP. Este proceso se repite en cada capa. Puede imaginarse como la actualización/aprendizaje de una capa a la vez en el orden inverso de la predicción.

21. El aprendizaje se realiza de manera iterativa durante un número definido por el usuario de *epoch*. Un *epoch* es un período de aprendizaje. Dentro de cada uno, el aprendizaje basado en el SGD se realiza de manera iterativa en lotes seleccionados al azar.

22. Después de entrenar a lo largo de todas los *epochs*, se espera que el modelo haya aprendido los parámetros que tienen un error de predicción mínimo. Sin embargo, esta minimización es solo para los datos de entrenamiento y no se garantiza que sea el mínimo global. En consecuencia, el rendimiento en los datos de prueba no es necesariamente el mismo.

1.5.1 Función de pérdida

Sección tomada de [7].

La función de pérdida es un método para evaluar qué tan bien un algoritmo específico modela los datos proporcionados. Si las predicciones se desvían demasiado de los resultados reales, la función de pérdida devolverá un número muy grande. Gradualmente, con la ayuda de alguna función de optimización, la función de pérdida aprende a reducir el error en la predicción.

No existe una función de pérdida única que se adapte a todos los algoritmos. Hay varios factores involucrados en la elección de esta función para un problema específico, como el tipo de algoritmo de aprendizaje automático elegido, la facilidad para calcular las derivadas y, en cierta medida, el porcentaje de valores atípicos en el *dataset*.

En términos generales, las funciones de pérdida se pueden clasificar en dos categorías principales según el tipo de tarea de aprendizaje con la que se está tratando: pérdidas de regresión y pérdidas de clasificación.

Para los problemas de clasificación binaria, la pérdida de entropía cruzada aumenta a medida que la probabilidad predicha se aleja de la etiqueta real. En otras palabras, cuanto mayor sea la diferencia entre la probabilidad predicha de la etiqueta real, mayor será la pérdida de entropía cruzada.

$$\mathcal{L} = -\frac{1}{n}\left(\sum_{i=1}^{n} y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)\right) \tag{22}$$

Para los problemas de regresión, se suele utilizar el MSE (error cuadrático medio, *mean square error*). Como su nombre indica, el error cuadrático medio se mide como el promedio de las diferencias al cuadrado entre las predicciones y las observaciones reales. Solo se interesa por la magnitud promedio del error sin importar su dirección. Sin embargo, debido al cuadrado, las predicciones que están muy lejos de los valores reales se penalizan de manera más severa en comparación con las predicciones con menor desviación. Además, el error cuadrático medio tiene propiedades matemáticas que facilitan el cálculo de los gradientes.

$$MSE = \frac{\sum_{i=1}^{n} (\hat{y}_i - y_i)^2}{n} \tag{23}$$

El error absoluto medio, por otro lado, se mide como el promedio de la suma de las diferencias absolutas entre las predicciones y las observaciones reales. Al igual que el MSE, también mide la magnitud del error sin tener en cuenta su dirección. A diferencia del MSE, el MAE requiere de herramientas más complicadas, tales como la programación lineal, para calcular los gradientes. Además, el MAE es más robusto ante valores atípicos ya que no utiliza el cuadrado de las diferencias.

$$MAE = \frac{\sum_{i=1}^{n} |\hat{y}_i - y_i|}{n} \tag{24}$$

1.5.2 Dropout

La técnica de *dropout* cambió la forma de aprender los pesos. En lugar de aprender todos los pesos de la red juntos, el *dropout* entrena un subconjunto de ellos en una iteración de entrenamiento por lotes. Esta técnica actúa como regularización de la red.

Existen algunas similitudes con la regularización. Al igual que la regularización $\mathcal{L}_1$, que empuja los pesos pequeños a cero, el *dropout* empuja un conjunto de pesos a cero. Sin embargo, existe una diferencia evidente: $\mathcal{L}_1$ realiza una supresión de pesos basada en los datos, mientras que *dropout* lo hace de manera aleatoria.

No obstante, *dropout* es una técnica de regularización que se asemeja más a una regularización $\mathcal{L}_2$ bajo la asunción de linealidad.

$$\mathcal{L} = \frac{1}{2}\left(t - (1-p)\sum_{i=1}^{n} w_i\, x_i\right)^2 + p(1-p)\sum_{i=1}^{n} w_i^2 x_i^2 \tag{25}$$

p representa la tasa de *dropout.*

La tasa de *dropout* es la fracción de nodos que se eliminan en una iteración por lotes. El término de regularización en (25) y (28) tiene un factor de penalización $p(1-p)$.

El factor $p(1-p)$ es máximo cuando $p = 0{,}5$. Por lo tanto, la regularización por *dropout* es la más fuerte cuando $p = 0{,}5$. Por lo general, una tasa de *dropout* de 0,5 es una buena elección para las capas ocultas. Si el rendimiento de un modelo empeora con esta tasa de *dropout*, suele ser mejor aumentar el tamaño de la capa en lugar de reducir la tasa.

1.5.3 Pesos de las clases

La función de pérdida definida en (22) otorga igual importancia (pesos) a las muestras positivas y negativas. Esto dificulta la clasificación en los casos en

que los *datasets* no estén balanceados, es decir, con cantidades similares de muestras de las distintas categorías. Para hacerle frente a esto, se puede dar un mayor peso a las muestras positivas y un peso menor a las muestras negativas. El enfoque de ponderación de clases funciona de la siguiente manera:

- El objetivo de estimación del modelo es minimizar la pérdida. En un caso perfecto, si el modelo pudiera predecir todas las etiquetas perfectamente, es decir, $\hat{y}_i = 1$ cuando $y_i = 1$ y $\hat{y}_i = 0$ cuando $y_i = 0$, la pérdida sería cero. Por lo tanto, el mejor modelo estimado es aquel cuya pérdida se acerca más a cero.
- Con las ponderaciones de clase, $w_1 > w_0$, si el modelo clasifica erróneamente las muestras positivas, es decir, $\hat{y}_i \to 0 | y_i = 1$, la pérdida se aleja más de cero en comparación con si se clasificaran erróneamente las muestras negativas. En otras palabras, el entrenamiento del modelo penaliza más la clasificación errónea de las muestras positivas que la de las negativas.
- Por lo tanto, la estimación del modelo se esfuerza por clasificar correctamente las muestras positivas minoritarias.

En principio, se pueden utilizar pesos arbitrarios siempre que $w_1 > w_0$. Pero una regla general es la siguiente:

- w_1, peso de la clase positiva = número de muestras negativas / número total de muestras.
- w_0, peso de la clase negativa = número de muestras positivas / número total de muestras.

1.5.4 Función de activación

Las funciones de activación son uno de los principales impulsores de las redes neuronales. Una activación introduce propiedades no lineales en una red. Una red con una activación lineal es equivalente a un modelo de regresión simple. Es la no linealidad de las activaciones lo que hace que una red neuronal sea capaz de aprender patrones no lineales en problemas complejos.

Las redes de *deep learning* se entrenan con *backpropagation* que se basan en el gradiente. El aprendizaje se puede generalizar de la siguiente manera:

$$\theta_{n+1} \leftarrow \theta_n - \eta \nabla_\theta \tag{26}$$

Donde n es una iteración de aprendizaje, η es una tasa de aprendizaje y ∇_θ es el gradiente de la pérdida $\mathcal{L}(\theta)$ con respecto a los parámetros del modelo θ. La ecuación muestra que el aprendizaje basado en el gradiente estima θ de manera iterativa. En cada iteración, el parámetro θ se mueve "más cerca" de su valor óptimo θ^*.

Sin embargo, si el gradiente realmente se acercará de θ a θ^* dependerá del propio gradiente.

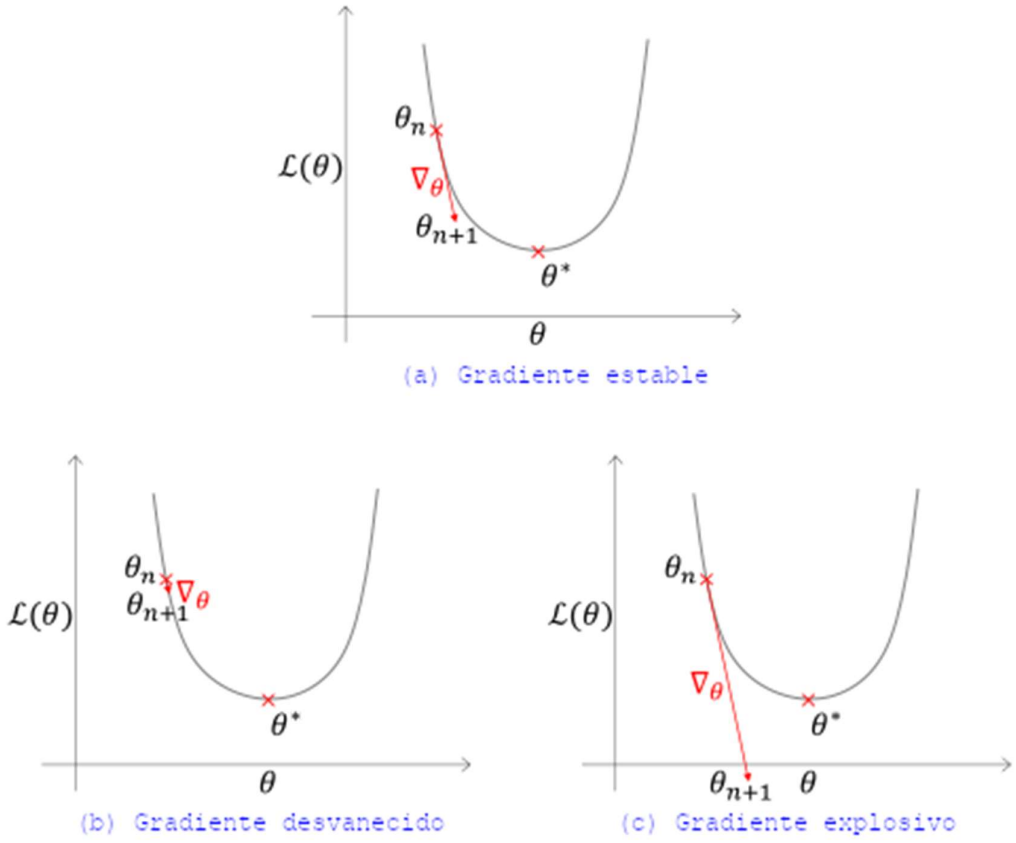

Ilustración 5: *Gradiente estable vs gradiente desvanecido y explosivo*

En estas figuras, el eje horizontal representa el parámetro del modelo θ, el eje vertical representa la pérdida $\mathcal{L}(\theta)$ y θ^* indica el parámetro óptimo en el punto más bajo de la pérdida.

Cuando el gradiente es estable θ_n se acerca a θ^*. Pero si el gradiente es demasiado pequeño, la actualización de θ se vuelve despreciable. Por lo tanto, el parámetro actualizado θ_{n+1} se mantiene alejado de θ^*. Cuando el gradiente es demasiado pequeño, se llama gradiente desvanecido. Este fenómeno se conoce como el problema del gradiente desvanecido.

Por otro lado, a veces el gradiente es masivo. Un gradiente grande aleja θ de θ^*. Este es el fenómeno del gradiente explosivo y hace que alcanzar θ sea bastante difícil.

1.6 Prueba de Weisfeiler-Lehman

Sección tomada de [8].

El algoritmo de la prueba de Isomorfismo Weisfeiler-Lehman produce una forma canónica para cada grafo. Si las formas canónicas de dos grafos no son equivalentes, entonces los grafos definitivamente no son isomorfos. Sin embargo, es posible que dos grafos no isomorfos compartan una forma canónica, por lo que esta prueba por sí sola no puede decidir de manera concluyente de que dos grafos son isomorfos.

El algoritmo funciona de la siguiente manera:

- En cada iteración i del algoritmo, se asigna a cada nodo una tupla $L_{i,n}$ que contiene la etiqueta comprimida antigua del nodo y un multiconjunto de las etiquetas comprimidas de los nodos vecinos. Un multiconjunto es un conjunto donde los elementos pueden aparecer varias veces y el orden no es importante.
- En cada iteración, también se asigna a cada nodo una nueva "etiqueta comprimida", $C_{i,n}$, para el conjunto de etiquetas de ese nodo. Dos nodos con las mismas etiquetas, $L_{i,n}$, obtendrán la misma etiqueta comprimida.

1. Para comenzar, se inicializan las etiquetas comprimidas $C_{0,n} = 1$ para todos los nodos n.

2. En la iteración del algoritmo (comenzando con $i = 1$), para cada nodo n, se establece que la tupla $L_{i,n}$ contiene la etiqueta antigua del nodo, $C_{i-1,n}$, y el multiconjunto de etiquetas comprimidas de todos los nodos vecinos, $C_{i-1,m}$, de la iteración anterior ($i - 1$).

3. Luego, se completa la iteración i asignando a una nueva "etiqueta comprimida", como un hash de $L_{i,n}$. Dos nodos con las mismas etiquetas deben obtener la misma etiqueta comprimida.

4. Se particionan los nodos en el grafo según sus etiquetas comprimidas. Se repiten los pasos 2 y 3 hasta un máximo de N (el número de nodos) iteraciones, o hasta que no haya cambios en la partición de los nodos por etiqueta comprimida de una iteración a la siguiente.

Cuando se utiliza este método para determinar el isomorfismo de grafos, se puede aplicar en paralelo. El algoritmo puede terminar rápido si los tamaños de las particiones de nodos particionados por etiquetas comprimidas divergen entre los dos grafos; si este es el caso, los grafos no son isomorfos.

Esto puede no parecer muy relevante, pero puede ser extremadamente difícil distinguir entre dos grafos grandes. De hecho, no se sabe si este problema se puede resolver en tiempo polinómico ni si es NP-completo.

1.7 Redes neuronales basadas en grafos (GNN)

Sección tomada de [9] y [10].

Sea $G = (V, E)$ un grafo con atributos de nodo X_v para $v \in V$ y atributos de arista e_{uv} para $(u, v) \in E$. Dado un conjunto de grafos $\{G_1, \cdots, G_N\}$ y sus etiquetas $\{y_1, \cdots, y_N\}$, la tarea del aprendizaje supervisado de grafos es aprender

un vector de representación h_G que ayude a predecir la etiqueta de un grafo completo G, $y_G = g(h_G)$.

Las GNNs utilizan la conectividad del grafo, así como las características de nodo y arista para aprender un vector de *features* h_v para cada nodo $v \in G$ y un vector h_G para el grafo completo G. Las GNNs modernas utilizan un enfoque de agregación del vecindario, donde la representación del nodo v se actualiza de manera iterativa mediante la agregación de las representaciones de los nodos vecinos y aristas de v. Después de k iteraciones, la representación de v captura la información estructural dentro de su vecindario de red de k saltos. Formalmente, la k-ésima capa de una GNN es:

$$a_v^{(k)} = AGGREGATE^{(k)}(\{h_u^{(k-1)} | u \in \mathcal{N}(v)\}) \quad (27)$$

$$h_v^{(k)} = COMBINE^{(k)}\left(h_v^{(k-1)} + a_v^{(k)}\right) \quad (28)$$

donde $h_v^{(k)}$ es la representación del nodo v en la k-ésima iteración/capa, euv es el vector de *features* de la arista entre u y v, y $N(v)$ es un conjunto de vecinos de v. Se inicializa $h_v^0 = X_v$.

Para obtener la representación del grafo completo h_G, la función $READOUT$ agrega las *features* de nodo de la última iteración K.

$$h_G = READOUT\left(\left\{h_v^{(K)} | v \in G\right\}\right) \quad (29)$$

La función $READOUT$ es una función invariante a la permutación, como el promedio o una función de agrupación más sofisticada a nivel de grafo.

La función $AGGREGATE$ reúne las *features* de los vecinos en el primer nivel. Por su parte, la función $COMBINE$ fusiona las *features* acumuladas de los vecinos con las actuales del nodo para su actualización. La función $READOUT$ convierte todas las *features* de los nodos en *features* a nivel de grafo, principalmente para la clasificación de grafos.

Las configuraciones comunes de la función $AGGREGATE$ son $suma$, $media$ y $máximo$. La función $suma$ aprende la información estructural, la $media$

tiende a aprender la información de la distribución y el *máximo* tiende a aprenderla de los elementos representativos. Dado que las funciones $media$ y $máximo$ no son inyectivas y, por lo tanto, no pueden distinguir entre distintos grafos, el rendimiento es peor que utilizando la $suma$; por consiguiente, se elige esta última función.

$COMBINE$ es un operador de suma directa u operador proyectivo. Si las funciones $AGGREGATE$, $COMBINE$ y $READOUT$ son inyectivas, GNN puede ser tan poderosa como la prueba de Weisfeiler-Lehman.

La capa fundamental de una GNN se llama convolución de grafos. Aunque existen muchos tipos de convolución de grafos, aquí se introducirán los principios básicos en los que se basa esta idea. Una convolución de grafos es una capa que calcula la siguiente representación latente de los nodos teniendo en cuenta la información estructural de una manera significativa. Para entender la convolución de grafos, primero se deben comprender las invariancias y propiedades que idealmente debería cumplir una GNN:

Invariancia y equivarianza de permutación: Supóngase que se quiere realizar una tarea de clasificación de grafos. Para trabajar con las *features* de los nodos, típicamente se las apila en una matriz $X \in \mathbb{R}^n \times \mathbb{R}^d$, siendo n el número de nodos y d la dimensión de las *features*. Al hacerlo, se ha introducido un orden en los vértices. Para una tarea de clasificación de grafos, está claro que cambiar el orden de los vértices (es decir, permutar los vértices) no debería cambiar el resultado de la clasificación. Por lo tanto, la red neuronal debe ser tal que aún dé el mismo resultado cuando se consideren los nodos en un orden diferente. Una función que cumple esta propiedad se dice que es invariante a la permutación. Un ejemplo de función invariante a la permutación es la suma. En fórmulas, si P es una matriz de permutación, X es la matriz de *features* y A es la matriz de adyacencia del grafo, la invariancia a la permutación se puede escribir como

$$f(PX, PAP^T) = f(X, A) \tag{30}$$

PX indica los vértices permutados, PAP^T indica la matriz de adyacencia permutada correspondiente (en la que, por supuesto, tanto las filas como las

columnas deben estar permutadas). Análogamente, si se quiere realizar una tarea de clasificación de nodos, se desearía que una permutación del orden de los nodos se refleje exactamente en el resultado (se desea que las predicciones de los nodos sigan la misma permutación que la entrada y, por supuesto, no permanezcan iguales). Una función que cumple esta propiedad se llama equivarianza de permutación y en fórmulas se puede escribir como

$$f(PX, PAP^T) = P(f(X, A)) \tag{31}$$

Dependiendo de la tarea específica (por ejemplo, clasificación de grafos o clasificación de nodos), puede ser deseable tener una GNN invariante a la permutación o equivariante a la permutación. La convolución de grafos vista como una forma de calcular una representación latente para cada nodo, debe ser una función equivariante a la permutación.

Localidad: De manera análoga a la convolución en imágenes, la convolución de grafos debe aprovechar el concepto de localidad, lo que significa que las *features* de los nodos en el vecindario $\mathcal{N}(x_i)$ del nodo considerado i deben influir en el cálculo de su representación latente. Los nodos vecinos de un nodo i son aquellos a los que i está conectado a través de una arista.

En relación con las propiedades deseadas que la convolución de grafos debe cumplir, se puede concluir debería ser una función con la siguiente estructura:

$$f(X, A) = \begin{pmatrix} g(x_1, \mathcal{N}(x_1)) \\ \cdots \\ g(x_N, \mathcal{N}(x_N)) \end{pmatrix} \tag{32}$$

Es importante notar que g no debe depender del orden en el que se consideran los nodos vecinos, lo que significa que la restricción de g sobre $\mathcal{N}(x_i)$ debe ser una función invariante a la permutación. Además, al igual que en la convolución clásica de imágenes, la función g es la misma en cada fila, lo que significa que los parámetros que se aprenden son los mismos. Esto reduce

drásticamente el número de parámetros para aprender. Una función f definida como en (32) tiene en cuenta la localidad y es equivariante a la permutación.

La capa funciona de la siguiente manera:

$$X_v^{(\ell+1)} = W_1^{(\ell+1)} X_v^{(\ell)} + W_2^{(\ell+1)} \sum_{w \in \mathcal{N}(v)} X_w^{(\ell)} \tag{33}$$

donde $W_i^{(\ell+1)}$ denota una matriz de pesos que se entrenan con la forma `[nro_features_salida, nro_features_entrada]`, para $i \in \{1,2\}$. Aquí, $X_v^{(\ell)}$ denota la representación latente del nodo v en la capa l. Esta fórmula expresa la función g introducida anteriormente. Es importante destacar que si se suman todos los vectores de *features* en el vecindario del nodo i , lo que aprovecha la localidad y garantiza la invariancia a la permutación de g sobre el vecindario considerado.

1.8 Graph Isomorphism Network (GIN)

El modelo GIN se centra en construir la representación a nivel de grafos. Para la tarea de clasificación de grafos, GIN puede generar una incrustación de todo el grafo a través de la función $READOUT$.

GIN combina $AGGREGATE$ (27) y $COMBINE$ (28) en un paso.

Sea $\mathcal{E}$ un parámetro que se aprende o un escalar constante. Luego, GIN actualiza las representaciones de nodos como se muestra a continuación:

$$h_v^{(k)} = MLP^{(k)}\left((1 + \mathcal{E}^{(k)}) \cdot h_v^{(k-1)} + \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)} \right) \tag{34}$$

1.9 BERT

Tomado de [11].

BERT (*Bidirectional Encoder Representations from Transformers*) es un revolucionario modelo de procesamiento de lenguaje natural (NLP) desarrollado por Google. Representa un cambio en la forma en que las máquinas comprenden el lenguaje, permitiéndoles entender las sutilezas intrincadas y las dependencias contextuales que hacen que la comunicación humana sea rica y significativa. Comprende que la relación impulsada por el contexto entre las palabras desempeña un papel fundamental en la obtención de significado. Captura la esencia de la bidireccionalidad, lo que le permite considerar el contexto completo que rodea a cada palabra, revolucionando la precisión y profundidad de la comprensión del lenguaje.

En su núcleo, BERT está impulsado por una poderosa red neuronal conocida como *Transformers*. Esta arquitectura incorpora un mecanismo llamado autoatención (*self-attention*), que le permite ponderar la importancia de cada palabra en función de su contexto, tanto anterior como posterior. Esta consciencia del contexto le dota de la capacidad de generar incrustaciones de palabras contextualizadas, que son representaciones de palabras considerando sus significados dentro de las oraciones. Es como si BERT leyera y releyera la oración para obtener una comprensión profunda del papel de cada palabra.

BERT necesita que el texto se divida en unidades más pequeñas llamadas *tokens*. La particularidad es que BERT utiliza la tokenización *WordPiece*. Divide las palabras en fragmentos más pequeños. Esto ayuda a manejar palabras difíciles y asegura que BERT no se pierda en palabras desconocidas.

El mecanismo de autoatención funciona observando cada palabra en una oración y decidiendo cuánta atención debe prestarles a otras palabras en función de su importancia. De esta manera, BERT puede centrarse en las palabras relevantes, incluso si están distantes en la oración.

BERT no se basa en una sola perspectiva; utiliza múltiples "cabezas" de atención. Estas “cabezas” serían como diferentes expertos que se centran en diversos aspectos de la oración. Este enfoque ayuda a BERT a capturar diferentes relaciones entre las palabras, haciendo que su comprensión sea más rica y precisa.

Capítulo 2: Machine learning aplicado a Detección de malware

2.1 Análisis dinámico

2.1.1 Introducción

El análisis dinámico consiste en la extracción de *features* ejecutando el programa. Dentro de esta categoría existen varas técnicas:

- Uso de registros y de memoria
- Traza de instrucciones
- Tráfico de red
- Traza de llamadas a las APIs

2.1.2 Uso de registros y memoria

El comportamiento de un programa se puede representar por los contenidos de la memoria en tiempo de ejecución. Es decir, los valores almacenados en los registros del microprocesador mientras el programa está en ejecución pueden distinguir entre el software benigno del malicioso. [12]

En [13] se propone un método basado en similitudes de comportamiento de malware. En principio, se monitorea el comportamiento en tiempo de ejecución y los valores almacenados en los registros para cada llamada a la API interceptada. Luego, se rastrea la distribución y los cambios en los valores de los registros, se los almacena en un vector con los valores de los siguientes registros $(EAX, EBX, EDX, EDI, ESI, EBP)$. Posteriormente, se calcula un *score* de similitud entre el vector obtenido y todo el set de entrenamiento. Finalmente, el vector es catalogado con aquella muestra con la que haya tenido mayor similitud. Este tipo de análisis se denomina VSA (*Value Set Analysis*). Es un método estático que traza la distribución de los valores dentro del ejecutable. Se basa en la premisa de que las diferentes variantes de malware metamórfico y polimórfico permanecen sin cambios. Este tipo de malware cambia en su apariencia, pero el comportamiento malicioso sigue siendo el mismo.

En [14] se propone un método para encontrar similitudes en el comportamiento en tiempo de ejecución basándose en la asunción de que los

comportamientos afectan los valores de los registros de manera diferente. El comportamiento en tiempo de ejecución se guarda y las llamadas a las APIs de algunas librerías estándar son interceptadas. Posteriormente, el sistema analiza los contenidos de memoria y los valores de los registros para construir un *score* de similitud. Cuando un nuevo registro entra al sistema, se obtiene el *score* de similitud más alto contra todos los prototipos. Los prototipos son conjuntos pequeños de muestras que son representativas de todo el *dataset*, que proveen una aproximación aceptable en el análisis de distancia por pares. Finalmente, los dos archivos son similares si alcanzan un umbral mínimo.

2.1.3 Traza de instrucciones

Los enfoques basados en firmas son populares ya que tienen una baja tasa de falsos positivos y poca complejidad computacional. Esto hace que sean muy utilizados hoy en día. Lamentablemente, técnicas como la ofuscación de código hacen la labor más compleja.

Para combatir estas limitaciones se han utilizado varios enfoques, uno de ellos es el análisis estático de n-gramas o la traza de instrucciones.

Para los modelos de n-gramas existen dos parámetros: n que representa el tamaño de las subsecuencias que son analizadas y L que representa el número de n-gramas que se va a analizar. Cabe mencionar que estos valores no escalan y se cae en la “maldición de la dimensionalidad”, es decir, que no existe suficiente espacio para almacenar el modelo. Con valores más pequeños se pierde el poder discriminatorio.

La traza de instrucciones es una secuencia de instrucciones de microprocesador que son llamadas durante la ejecución de un programa. A diferencia de un rastro de instrucciones estático, estas están ordenadas de acuerdo a como son ejecutadas mientras que las estáticas están ordenadas según su orden de aparición en el archivo binario.

El rastro dinámico es una medida más robusta del comportamiento del ejecutable, ya que, los ofuscadores y empaquetadores alteran el orden de las instrucciones desde un análisis estático.

En [15] se propone la utilización de Cadenas de Markov representadas como un grafo dirigido y con pesos en las aristas. Las instrucciones se

representan como los vértices y los pesos de las aristas representan las probabilidades de transición de la Cadena de Markov. Estas últimas se estiman utilizando las trazas de los programas que son recolectadas.

La contribución que aporta es la construcción de una matriz de similitudes (*kernel*) entre los grafos que representan a las Cadenas de Markov y luego utilizarla para clasificación. Se utilizan dos medidas diferentes de similitud para construir la matriz del *kernel*:

- Una medida local que compara las correspondientes aristas en cada grafo
- Una medida global que compara aspectos topológicos del grafo.

Esta combinación habilita a comparar grafos de trazas de instrucciones utilizando criterios muy diferentes en un marco unificado. Finalmente, una vez que la matriz se construye, se utiliza SVM para llevar a cabo la clasificación.

call	[ebp+0x8]
push	0x70
push	0x010012F8
call	0x01006170
push	0x010061C0
mov	eax, fs:[0x00000000]
push	eax
mov	fs:[], esp
mov	eax, [esp+0x10]
mov	[esp+0x10], ebp
lea	ebp, [esp+0x10]
sub	esp, eax
...	...

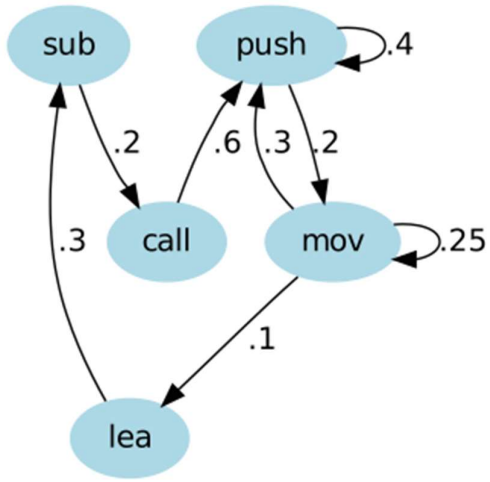

Ilustración 6: *Ejemplo de representación de Cadena de Markov para trazas de instrucciones*

En [2] se presenta una solución que utiliza *VirtualBox* ya que ofrece APIs en comparación con otros virtualizadores. Se usan además herramientas para ocultar la presencia del depurador (*OllyDBG*). El sistema operativo emulado cuenta con *Flash*, *.NET Framework*, elementos en el historial de navegación, elementos en la papelera de reciclaje, etc. Esto sirve para simular lo más posible un equipo real.

El malware es ejecutado automáticamente por nueve minutos. Esta es una solución de compromiso, ya que mayores tiempos pueden dar mejores

resultados, pero se demora mayor cantidad de tiempo en el procesamiento en lotes de archivos.

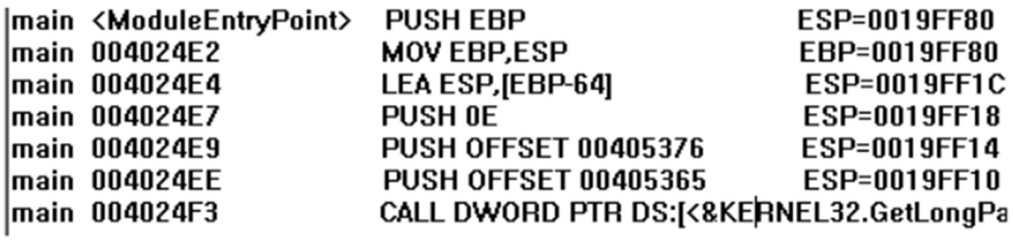

Ilustración 7: *Ejemplo de traza obtenida*

De las trazas obtenidas se crean dos *datasets*:

- *Count-based*: cuenta las ocurrencias de cada *opcode*. Por ejemplo, en el ejemplo anterior: `PUSH`: 4; `MOV`: 1; `LEA`: 1; `CALL`: 1.
- *Sequence-based*: muestra las secuencias ordenadas de *opcodes*. Por ejemplo, `PUSH MOV LEA PUSH PUSH PUSH CALL`

Para el primer *dataset s*e utilizó el algoritmo *Random Forest* (RF) que es fácilmente paralelizable y está recomendado para clases desbalanceadas.

Para el segundo *dataset* se utilizó HMM.

Como resultado se concluyó que la utilización de clústeres previos a la ejecución de ambos algoritmos contribuye a una mejor clasificación.

2.1.4 Tráfico de Red

La detección de tráfico malicioso en una red puede proveer de manera unívoca perspectivas del comportamiento del malware.

Una vez que un equipo es infectado por malware, puede establecer una conexión con un servidor externo para recibir órdenes que se ejecutarán en la víctima; descargar actualizaciones u otro malware o revelar información sensible y/o privada.

Monitorear el flujo de datos que ingresa y sale de una red, así como la actividad interna de los dispositivos en cuestión, proporciona información valiosa para identificar comportamientos maliciosos.

En [16] se propone una solución trasversal basada en niveles y protocolos que clasifique el tráfico utilizando métodos de aprendizaje supervisado. El enfoque planteado examina los protocolos DNS, HTTP y SSL y utiliza diversos métodos de clasificación de redes en múltiples niveles de detalle de las actividades que realizan, con el fin de obtener una mejor diferenciación entre el tráfico malicioso y el tráfico seguro.

La clasificación del tráfico de red se puede realizar de dos maneras:

- **Métodos a nivel de paquete:** examinan las *features* de cada paquete y las firmas de la aplicación.
- **Métodos a nivel de flujo:** se basan en la agregación de paquetes en flujos y la extracción de *features* y análisis estadístico del flujo.

La clasificación del tráfico de red puede basarse en diferentes atributos principales:

- **Atributos basados en puertos:** se basan en los números de puertos TCP o UDP asignados por la Autoridad de Números Asignados de Internet (IANA).
- **Atributos basados en *payload:*** se basan en firmas del tráfico a nivel de la capa de aplicación.
- **Atributos basados en estadísticas:** se relacionan con las *features* estadísticas del tráfico (por ejemplo, la duración del flujo, el tiempo de inactividad, el tiempo y la longitud de llegada de los paquetes). Estos atributos son únicos para ciertas clases de aplicaciones y permiten distinguir diferentes aplicaciones de origen entre sí.

La singularidad de la solución radica en el hecho de que se observa el análisis de flujo de datos en cuatro resoluciones, basadas en las capas de Internet, Transporte y Aplicación, con *features* generadas en consecuencia. Estas son: *Transaction*, *Session*, *Flow* y *Conversation Windows*.

- ***Transaction*:** representa una interacción entre un cliente y un servidor. Es una comunicación bidireccional: el cliente envía una solicitud al servidor y el servidor procesa la solicitud y envía una respuesta de vuelta al cliente.
- ***Session*:** una tupla única que consta de direcciones IP de origen y destino y números de puerto.

- Una sesión TCP comienza con un *handshake* exitoso y termina con un tiempo de espera o con un paquete con el *flag* RST o FIN de cualquiera de los dispositivos.
- Una sesión UDP consta de todos los paquetes enviados desde un cliente a un servidor y desde un servidor a un cliente hasta que se alcanza un tiempo de inactividad de comunicación definido.

- ***Flow*:** un grupo de sesiones entre dos IPs durante el período de agregación. El período de agregación puede ser especificado por un algoritmo como el período preciso de tiempo desde el inicio de la primera sesión en el flujo, hasta el tiempo máximo de inactividad entre dos sesiones. Un nuevo flujo comienza si el tiempo entre el final de una sesión (el último paquete) y el inicio de una nueva sesión (primer paquete) es mayor que el tiempo de inactividad definido. La nueva sesión es entonces parte del nuevo flujo.
- ***Conversation Windows*:** un grupo de flujos entre un cliente y un servidor durante un período de observación. Una conversación puede ser definida entre un par de IPs o un grupo de recursos de red.

El modelo consta de 927 *features.*

Se desarrolló un extractor especializado para procesar el tráfico de red en bruto, extraer las *features* y proporcionarlas como entrada al analizador de aprendizaje automático.

Se eligieron los siguientes algoritmos de clasificación Naïve Bayes, D*ecision Tree* (J48) y *Random Forest*. Para la selección de *features* se utilizó el algoritmo CFS (*Correlation Feature Selection*). Todos los algoritmos fueron ejecutados sobre Weka[1].

Se demostró que diferentes *features*, capas cruzadas y protocolos mejoran el rendimiento de la clasificación. La precisión no se vio afectada por los entornos de red, tanto en redes de prueba como en reales. El rendimiento predictivo mostró una mejora significativa sobre los sistemas modernos de detección de intrusiones en red basados en reglas. Se detectaron softwares

[1] https://www.cs.waikato.ac.nz/ml/weka/

maliciosos desconocidos al menos un mes antes de que se implementara su regla estática. Dado que el método propuesto analiza solo el comportamiento del tráfico y no su contenido, es efectivo incluso para tráfico cifrado y para malware que utiliza recursos de red legítimos. Como no se utiliza el análisis de *payload* para este enfoque, se preserva la privacidad de los usuarios, por lo que estos sistemas de detección de intrusiones en red (NIDS) pueden integrarse en los sistemas de redes empresariales.

En [17] se propone un sistema para detectar infecciones de malware APT (*Advanced Persistent Threat*) basándose en DNS malicioso y análisis de tráfico.

El principal objetivo de este tipo de malware es controlar máquinas remotamente y robar información confidencial en lugar de lanzar ataques DOS, enviar spam o causar daño. Dadas sus características, requieren un alto grado de sigilo durante sus conexiones que suelen ser prolongadas y persistentes. Para logar esto, se suele utilizar el protocolo DNS, ya que, si estuviera la IP del servidor escrita en el código, si por alguna razón esta dejase de estar disponible, generaría un fallo imposible de recuperar. Una vez que el servidor C&C caiga o la IP sea detectada, los objetivos comprometidos pasarían a estar fuera del alcance del atacante. Por lo tanto, el uso del sistema DNS, provee un método flexible para migrar entre distintos servidores C&C. Por último, este método ayuda al atacante a ocultarse detrás de una red de *proxies* más fácilmente.

Es un gran desafío identificar aquellos dominios maliciosos involucrados en la actividad de un malware APT, principalmente en grandes redes donde circulan datos a gran escala. Ejemplos de estas redes pueden ser ISP o grandes corporaciones.

Los ataques ATP no usan dominios DGA o servicios *flux*. El primero hace referencia a *Domain Generation Algorithm* que se utiliza para generar una gran cantidad de nombres de dominio. Por ejemplo, el gusano *Conficker* generaba 50.000 nombres de dominio por día para comunicarse con los servidores C&C. El nombre de dominio para el servidor se seleccionaba de manera aleatoria de una lista. Por otro lado, están los servicios *flux* que son análogos a los servicios CDN (*Content-Delivery Network*). En lugar de proveer el contenido más próximo a fin de reducir la latencia, consiste en una red con gran número de equipos infectados.

El sistema propuesto consiste en dos componentes principales: i) detector de DNS malicioso y, ii) analizador de tráfico de red.

El detector de DNS malicioso extrae unas 14 *features* indicativas de malware APT y de dominios C&C. Por otro lado, el analizador de tráfico combina el sistema basado en firmas, que detecta las infecciones en base a las reglas del conjunto VRT de Snort, y el sistema basado en anomalías que ocurren a nivel protocolo y aplicación. Estos tres subsistemas se combinan para servir como entrada a un *Reputation Engine*. Este asigna un puntaje entre 0 y 1, donde 0 representa una baja reputación, es decir, está infectado por malware y 1 cuando no lo está. Se implementó utilizando una función de reputación como un clasificador estadístico.

Los resultados de la función de reputación y clasificador DNS son de alrededor del 96% y la tasa de falsos positivos de alrededor del 1,5%.

La principal limitación que se encontró con el método planteado es que no es bueno para detectar infecciones que no dependan de dominios, como aquellos troyanos que utilizan direcciones IP directamente para conectarse con el servidor C&C.

En [18] se analizan las anomalías en los *User Agent* de los encabezados del protocolo HTTP en el tráfico generado por malware. Las *botnets* C&C modernas disimulan su actividad mediante el uso de DGA, *flux* y TOR.

Clasifica a las *botnets* C&C en dos tipos:

- **Descentralizada:** evitan nodos únicos de falla mediante el uso de arquitecturas distribuidas donde cada nodo infectado puede ser utilizado por el *botmaster*. Aunque proporcionan una mejor resistencia a la eliminación, las *botnets* descentralizadas son difíciles de operar. Cada host cuando se infecta necesita recibir una lista de bots a los que puede conectarse, mientras que es mucho más fácil redirigir un *bot* a un servidor central de C&C. Los comandos también tardan más tiempo en llegar a todos los bots debido a la latencia dada por la topología distribuida.
- **Centralizada:** son más fáciles de operar ya que dependen de un pequeño conjunto de servidores maestros. Los comandos llegan rápidamente a todos los *bots* de la red, lo que garantiza un tiempo

de respuesta más corto. Son más fáciles de crear y los componentes adecuados para construir dichos *botnets* están fácilmente disponibles para la venta o el alquiler en el mercado negro de internet. Por lo tanto, las arquitecturas de C&C centralizadas son el tipo de *botnets* más extendido en la actualidad. Entre las *botnets* centralizadas, el protocolo HTTP es el más prevalente implementado por *malware*.

El enfoque que se plantea propone analizar aquellos *User Agent* que sean anómalos, en lugar de analizar todo el tráfico HTTP en su totalidad.

El protocolo HTTP no define explícitamente la semántica del campo *User Agent*, sino que solo es útil desde el punto de vista del servidor que maneja las solicitudes, para adaptar las respuestas y restringir a ciertos *User Agent*.

Los *User Agent* de malware pueden considerarse inofensivos si coinciden con aquellos conocidos del tráfico benigno. Estos pueden ser estáticos o dinámicos, siendo los primeros indicados en el código fuente del malware y los últimos proporcionados por el *botmaster* o tomados del navegador del usuario.

Los encabezados estáticos solo pueden ser detectados si el malware utiliza versiones obsoletas de los *User Agent* conocidos. Sin embargo, estos no son lo suficientemente confiables como para usarse como una única firma de detección.

Algunas son utilizadas por el *malware* para implementar funcionalidades maliciosas y son inherentes a la operación del malware.

- **Fuga de información:** el *malware* utiliza el *User Agent* para filtrar información sobre el nodo infectado, como la hora local, la dirección IP, la configuración del *bot* y los datos de usuario privados. Esta información puede ser utilizada por los *botmasters* para coordinar actividades maliciosas, enviar actualizaciones de malware y localizar datos en el host de la víctima para desencadenar ataques.
- **Identificación del bot:** el encabezado de agente de usuario es utilizado por los bots para filtrar información sobre el nodo infectado y especificar identificadores únicos, permitiendo al *botmaster* identificar cada *bot* en la red. Los identificadores de *bot* se implementan como valores hash o claves primarias, o como encabezados compuestos con elementos estáticos comunes e IDs

del bot. Algunos simplemente designan a su familia de la botnet dentro del *User Agent*.

- **Vectores de ataques de *malware*:** el malware utiliza el *User Agent* para atacar aplicaciones web vulnerables.
- ***User Agent* desconocido:** incluye aquellos que son desconocidos y generados al azar por el malware o que contienen errores tipográficos. Estos suelen ser utilizados por herramientas listas para usar. Estos encabezados pueden ser detectados mediante firmas adecuadas que se construyen utilizando un enfoque específico.

De acuerdo con lo mencionado, es impracticable la detección de malware utilizando listas negras de *User Agent* maliciosos.

En el documento se propone agrupar a los *User Agent* en base a patrones similares para construir firmas de detección que enfaticen en patrones comunes entre múltiples agentes, lo que reduce la probabilidad de aleatoriedad. Estas firmas son específicas para detectar tráfico HTTP de malware y también permite detectar aquel que no esté en el *dataset*.

Se propone un proceso de *clustering* de *User Agent* de dos pasos para encontrar patrones similares y generar las firmas de detección específicas. El primer paso consiste en agrupar los *User Agent* con similitudes estadísticas de alto nivel, mientras que el segundo, refina la agrupación para crear grupos más pequeños dentro del mismo clúster.

Los grupos resultantes se utilizan para extraer el conjunto óptimo de firmas para detectar malware. La validación del conjunto de firmas se realiza mediante la *cross-validation* y la prueba se efectúa contra el tráfico HTTP real.

Los experimentos realizados mostraron que la tasa de detección disminuyó a medida que se recopilaron nuevas muestras de malware. Se demostró que las firmas son lo suficientemente genéricas como para coincidir con nuevas variantes de malware en el *dataset*. Por otro lado, no se logró una tasa de detección del 100% para el malware recopilado, ya que algunos utilizaban *User Agent* demasiado cortos. Las firmas creadas con esas cadenas se descartaron ya que se encontró que coincidían con *User Agent* benignos.

En [19] se plantean dos enfoques: clasificación de los encabezados IP y *Deep Packet Inspection* (DPI).

Para el primer enfoque, se etiquetaron y extrajeron *features* de flujo de trazas de red para identificar el tráfico malicioso y el benigno. Es importante mencionar que estas *features* se pueden extraer incluso cuando el tráfico está cifrado, pues se derivan de las cabeceras de los paquetes. Las *features* de flujo se basan en la duración de este, la dirección, tiempos entre paquetes, el número de paquetes intercambiados y el tamaño de estos. Cada flujo se representa mediante un vector de 22 *features*.

Se utilizaron varios algoritmos para clasificación: J48, Naïve Bayes, SVM, Boosted J48 y Boosted Naïve Bayes. El experimento demostró que J48 y Boosted J48 tienen un mejor desempeño en comparación con el resto

Al mismo tiempo, se buscó detectar acciones de malware en el tráfico de red utilizando atributos de flujo y HMMs para corroborar la huella digital (*fingerprint*) de las familias de malware. Por huella digital, se refieren a la detección de tráfico malicioso y a la atribución de la familia de malware. Para ello, se utilizó un enfoque de *clustering* en flujos unidireccionales, y se indexaron las secuencias de flujos maliciosos por familia de malware para construir los modelos HMM que representan el comportamiento de cada familia de malware.

Para etiquetar los diferentes flujos que pertenecen a una secuencia, se adoptó un enfoque de *clustering*. La razón es caracterizar flujos maliciosos de entrada y salida en grupos que representan sus comportamientos de red. Para hacerlo, los flujos se representaron mediante vectores de 45 *features*. Para realizar el *clustering* del tráfico de entrada/salida, se generaron archivos de *features* compatibles con la herramienta de *clustering* CLUTO. Para etiquetar los flujos, se utilizó el algoritmo de partición k-means Repeated Bi-Section implementado en CLUTO. Este algoritmo pertenece a los algoritmos de *clustering* particional. Se sabe que estos algoritmos funcionan en la agrupación de grandes conjuntos de datos debido a su bajo costo computacional. k-means RBS deriva soluciones de *clustering* basadas en una función de criterio global. Este algoritmo crea inicialmente 2 grupos; cada grupo se divide en dos hasta que se optimiza la función de criterio. El algoritmo k-means RBS utiliza el modelo de espacio vectorial para representar cada flujo unidireccional.

Cada flujo se representa mediante un vector de dimensión $f_v = (f_1, f_2, \dots, f_i)$, donde f_i es la i-ésima *feature* del flujo unidireccional. Para calcular la similitud entre los vectores, se utiliza la función coseno. Para agrupar los diferentes flujos unidireccionales, se utiliza una función de criterio híbrida que se basa en una función interna y una función externa.

La función interna intenta maximizar las similitudes promedio entre flujos que se asignan a cada grupo.

La función externa deriva la solución al optimizar una solución que se basa en cómo los diferentes grupos difieren entre sí. La función híbrida combina funciones internas y externas para optimizar ambas simultáneamente. Usando el algoritmo k-means RBS como base, se diseñó un conjunto de experimentos que generan soluciones de *clustering* para el flujo de entrada y el de salida.

Se eligió una solución en la que la métrica de similitud interna (ISIM) es alta y la métrica de similitud externa (ESIM) es moderada.

Una vez realizado el proceso de etiquetado, se entrenaron para las diferentes familias de malware modelos HMM y de esta manera se procedió a asignarlos a las distintas familias.

En el segundo enfoque, DPI, se aplicaron técnicas de aprendizaje automático en paquetes completos de capturas (*pcap*). Cada *pcap* se cargó interpretándolo como una onda. La señal encierra flujos que tienen secciones de cabecera y de *payload*. Es importante mencionar que todos los experimentos de clasificación se realizaron con las configuraciones predeterminadas, es decir, que no se ha considerado el ajuste de la parametría.

Los datos se procesan utilizando n-gramas para construir un valor de amplitud de muestra en la señal. En este caso, se usan di-gramas (dos caracteres o bytes consecutivos) para construir la señal. Las razones detrás del uso de di-gramas es que se ha demostrado su eficacia en la detección de canales C&C; y está adaptado a la forma de onda codificada en el *framework* MARF.

Es importante mencionar que las fases de entrenamiento y prueba se basan en la combinación de varias técnicas de aprendizaje automático para seleccionar las mejores opciones. Se aplicaron técnicas de aprendizaje automático de procesamiento de señales y procesamiento del lenguaje natural (NLP) para analizar y detectar la actividad maliciosa en flujos *pcap*. Se integró

una herramienta de prueba de concepto llamada MARFPCAT, una herramienta de análisis de PCAP basada en MARF, para detectar y clasificar código vulnerable y débil con una precisión relativamente alta. Para entrenar los algoritmos se utilizó el 70% de los diferentes flujos de red de malware y medir la precisión en los flujos restantes. Es importante mencionar que los datos de entrenamiento y prueba están indexados por familia de malware.

En contraste con el primer enfoque, donde el tráfico benigno se recopila de terceros; se considera el tráfico benigno como ruido que se encuentra en las trazas *pcap*. Para separar este tráfico del malicioso se utilizaron filtros. Además, se puede aprender de la señal del tráfico benigno y restarla a del tráfico malicioso (señal maliciosa) para aumentar la precisión de la huella digital. Sin embargo, esta técnica de sustracción de señal disminuye el rendimiento en tiempo de ejecución. Los resultados fueron muy prometedores aún sin la sustracción de tráfico benigno.

Como resultados se tiene que, para el primer enfoque, en particular, para la clasificación, se obtuvieron los mejores resultados utilizando J48 y Boosted J48. Ambos obtuvieron una precisión del 99% y menos del 1% de falsos positivos. Adicionalmente, ambos obtuvieron una alta precisión en varios *datasets*.

El algoritmo SVM obtuvo una precisión entre el 89% y el 95%. En contraste con Naïve Bayes y Boosted Naïve Bayes que no obtuvieron buenos resultados. Dicho esto, se afirma que tanto J48 como Boosted J48 son efectivos para discriminar entre tráfico malicioso y benigno.

Con respecto a la atribución de familias de malware, se utilizaron soluciones de *clustering* k-means RBS para el tráfico de entrada y salida, evaluadas por cohesión y aislamiento, y se seleccionaron soluciones con un promedio de ISIM mayor o igual a 0,95 y un promedio de ESIM menor o igual a 0,5, limitándose a 12-18 clústeres. Se combinaron las soluciones de entrada y salida y se eligió la combinación con la mayor proporción de secuencias no compartidas por familias de malware para entrenar HMMs. Se entrenaron HMMs de 2 estados con secuencias de entrenamiento de longitud 2 para todas las familias de malware, y se recomendó el uso de HMMs de 2 ó 4 estados según la necesidad de expresividad. La distribución de secuencias de entrenamiento se mantuvo adecuada incluso con secuencias cortas.

Para el segundo enfoque, se utilizaron varias medidas estadísticas para evaluar la precisión en la detección de diferentes clases de malware bajo diferentes configuraciones de algoritmos, incluyendo la medida de "segunda suposición". Aunque la precisión general es baja, muchas familias individuales de malware se identifican correctamente. La baja precisión se debe principalmente a la clase de malware "genérico" no fue filtrada en este experimento. Los experimentos se replican utilizando filtros basados en la transformada *wavelet*, pero se observa una disminución en la precisión, lo que plantea la pregunta de si realmente es necesario el preprocesamiento para preclasificar rápidamente un flujo de paquetes. El escaneo global inicial produjo resultados para 1.063 familias de malware, muchas de las cuales se identifican con una precisión de casi el 100%, a menudo incluso usando un solo paquete. Sin embargo, la presencia de clases de malware etiquetadas como "genéricas" afecta negativamente la precisión global.

2.1.5 Trazas de llamadas a la API

En [20] se sugiere la implementación de *clustering* con el fin de identificar nuevas categorías de malware que presenten patrones similares, complementado con la clasificación de comportamiento para asignar malware desconocido a categorías ya conocidas.

El marco de trabajo propuesto ejecuta y monitorea binarios de malware en un ambiente de pruebas y genera informes secuenciales de comportamiento para cada binario. Los informes se incrustan en un espacio vectorial de alta dimensión para evaluar geométricamente la similitud del comportamiento y diseñar métodos de agrupación y clasificación para identificar clases nuevas y conocidas de malware. Se utilizan técnicas de aprendizaje automático para el *clustering* y clasificación y se aplican vectores prototipo para el cómputo eficiente del análisis. El comportamiento del malware se analiza incrementalmente, primero identificando el comportamiento que coincide con clases de malware conocidas y luego agrupando los informes con comportamiento no identificado para descubrir nuevas clases de malware.

Debido a que el formato en XML generado a través del monitoreo no es adecuado para analizar automáticamente el comportamiento, los autores

sugieren el uso de una representación especial llamada MIST (*malware instruction set*). Esta representación está inspirada en los conjuntos de instrucciones empleados en el diseño de microprocesadores. Sin embargo, esa representación no es adecuada para la aplicación de técnicas de análisis eficientes, ya que estas suelen operar sobre vectores de números reales. Para abordar este problema, se introdujo una técnica para incrustar informes de comportamiento en un espacio vectorial, la cual se inspiró en conceptos de NLP y detección de intrusiones basada en host.

Un informe x sobre el comportamiento del malware se corresponde a una secuencia simple de instrucciones. Se mueve una ventana sobre el informe para obtener subsecuencias de longitud q y así obtener q-gramas de instrucciones, que reflejan patrones de comportamiento y capturan algunos aspectos semánticos del programa. Se construye una incrustación de informes utilizando todos los posibles q-gramas de instrucciones del conjunto S.

$$S = \{(a_1, \dots a_q) \mid a_i \in \Lambda, 1 \leq i \leq q\} \tag{35}$$

Donde Λ denota el conjunto de todas las instrucciones posibles. Según el nivel de MIST considerado, la granularidad de Λ y S puede variar desde simples llamadas al sistema (nivel = 1) hasta instrucciones completas que cubren diferentes bloques de argumentos de llamadas al sistema (nivel > 1).

Se puede incrustar un informe x del comportamiento de malware en un espacio vectorial de $|S|$ dimensiones utilizando el conjunto S, donde cada dimensión se asocia con un q-grama de instrucción y, por lo tanto, con un patrón de comportamiento corto. La función φ se asemeja a un indicador de la presencia de q-gramas de instrucción y se puede definir formalmente de la siguiente manera:

$$\varphi(x) = (\varphi_s(x))_{s \in S}, \varphi_s(x) = \begin{cases} 1 \text{ si el reporte } x \text{ contiene } q - \text{gramas } s \\ 0 \text{ en otro caso} \end{cases} \tag{36}$$

A pesar de la complejidad del cómputo y la comparación de vectores en espacios de alta dimensionalidad, se descubrió que el número de q-gramas de instrucciones en un reporte es lineal en su longitud. Esto significa que el vector de *features* $\varphi(x)$ solo tenía un número limitado de dimensiones no nulas, lo que

permitió desarrollar métodos de extracción y comparación de reportes incrustados en tiempo lineal y análisis de comportamientos eficiente. La longitud de los informes dominaba estos factores e introducía un sesgo implícito, lo que hacía problemática la comparación de informes pequeños y grandes. Para compensar este sesgo, se introdujo una función de incrustación normalizada.

$$\hat{q}(x) = \frac{\varphi(x)}{\|\varphi(x)\|} \tag{37}$$

Este tipo de normalización se usa ampliamente en el ámbito de la recuperación de información para comparar documentos de texto, donde generalmente se aplica como parte de la función de similitud coseno.

Para comparar el comportamiento de dos informes x y z, se utilizó la distancia euclídea en $\mathbb{R}^{|S|}$.

Los valores de d variaron desde $d(x,z) = 0$, para un comportamiento idéntico, hasta $d(x,z) = \sqrt{2}$ para informes con desviaciones máximas debido a la normalización.

Se requieren métodos de *clustering* y clasificación para identificar y asignar malware a clases de comportamiento conocidas o nuevas. Sin embargo, la mayoría de los métodos de aprendizaje no son aplicables para procesar grandes cantidades de datos diariamente. Se propone una solución que utiliza prototipos de grupos de comportamientos similares para acelerar el cómputo de los métodos de aprendizaje y mejorar la eficiencia de las técnicas de *clustering* y clasificación.

La extracción de un conjunto pequeño pero representativo de prototipos a partir de un *dataset* no fue una tarea trivial. La mayoría de los enfoques para la extracción de prototipos se basan en el *clustering* o cálculos lineales, por lo que no eran apropiados para una aproximación eficiente. Además, la tarea de encontrar un conjunto óptimo de prototipos puede demostrarse que pertenecen a la clase NP-*hard*. Existe un algoritmo de tiempo lineal que determina un conjunto de prototipos en un tiempo de ejecución dos veces mayor que la solución óptima. El algoritmo procede seleccionando iterativamente prototipos de un conjunto de informes. El primer prototipo es fijo o elegido al azar. Durante cada ejecución, se calcula la distancia desde el conjunto actual de prototipos a

los informes restantes. Se elige el informe más lejano como nuevo prototipo, de modo que el conjunto de datos esté cubierto de forma iterativa por una red de prototipos. Este procedimiento se repite hasta que la distancia desde cada vector a su prototipo más cercano fuera inferior al parámetro d_p.

La complejidad de tiempo de ejecución de este algoritmo es $O(kn)$ donde n es el número de informes y k el número de prototipos. Dado que d_p se elige razonablemente, el algoritmo es lineal en el número de informes, ya que k depende únicamente de la distribución de datos. Si no se puede determinar tal d_p, las iteraciones totales aún son limitadas por un límite fijo k.

El algoritmo para clusterizar utilizando prototipos es el siguiente: se toma a cada prototipo como un clúster individual, el algoritmo procede iterativamente determinando y fusionando el par de clústeres más cercano. Este procedimiento termina si la distancia entre los clústeres más cercanos es mayor que el parámetro d_c.

Para calcular las distancias entre clústeres, el algoritmo considera la distancia máxima de sus miembros individuales. Una vez que se determina un clúster, se propaga a los informes originales. Además, se rechazan y mantienen para un análisis incremental posterior aquellos clústeres con menos de m miembros.

El algoritmo tiene una complejidad de tiempo de ejecución de $O(k^2 \log k + n)$, donde n es el número de informes y k es el número de prototipos. En comparación con el agrupamiento jerárquico exacto con un tiempo de ejecución de $O(n^2 \log n)$, la aproximación proporciona un factor de aceleración de $\sqrt{n/k}$.

El comportamiento de un malware desconocido se clasificó por clases conocidas de comportamiento, donde los datos de entrenamiento iniciales fueron etiquetados utilizando software antivirus. Desafortunadamente, la mayoría de los productos antivirus tienen etiquetas inconsistentes e incompletas y no son lo suficientemente precisas para el entrenamiento. Como solución a este problema, se emplearon las clases de malware descubiertas por agrupamiento como etiquetas para el entrenamiento y, de esta manera, se aprendió una discriminación entre los grupos conocidos de comportamiento de malware. Como estos grupos están representados por prototipos, se hace uso de la aproximación para acelerar el aprendizaje.

El algoritmo clasifica los informes en clústeres al determinar el prototipo más cercano de los datos de entrenamiento. Si está dentro de un radio determinado, se asigna al clúster, de lo contrario se retiene para análisis posterior. La implementación *naïve* tiene un tiempo de ejecución $O(kn)$, pero manteniendo los prototipos en estructuras de árbol especializadas, se puede reducir a $O(n \log k)$. Sin embargo, se prefiere la implementación *naïve* debido a su capacidad de paralelización y mejor rendimiento en sistemas con varios núcleos.

Durante una ejecución incremental, el número de informes disponibles puede ser insuficiente para determinar todos los grupos de comportamiento de malware. Para compensar esta falta de información, se rechazan los grupos con menos de m miembros y se envían los informes correspondientes de vuelta a la fuente de datos. Como resultado, el malware infrecuente se acumula hasta que haya suficientes datos disponibles para aplicarle el *clustering*. El tiempo de ejecución del algoritmo incremental depende del número de prototipos almacenados de ejecuciones anteriores (m) y del número de prototipos extraídos en la ejecución actual (k). Aunque la complejidad temporal es cuadrática en k, el número de prototipos extraídos durante cada ejecución permanece constante para conjuntos de tamaño y distribución iguales. La complejidad del análisis incremental está determinada por m, similar a la complejidad lineal de la detección de firmas en software antimalware.

El análisis incremental reduce los requisitos de memoria en un 94% y proporciona un factor de aceleración de 4, lo que permite procesar 33.000 informes de comportamiento de malware en menos de 25 minutos. La desviación estándar promedio entre las ejecuciones experimentales es inferior al 1,5%, y se considera que la precisión informada es estadísticamente significativa.

Para este experimento, se calculó el *F-Score* para el agrupamiento basado en prototipos y una implementación de agrupamiento jerárquico regular en el *dataset* de referencia.

La clasificación produjo un *F-Score* de más de 0,96. Al incorporar el comportamiento observado en un espacio vectorial, es posible aplicar algoritmos de aprendizaje, como el *clustering* y la clasificación, para el análisis del comportamiento de malware. Ambas técnicas son importantes para el procesamiento automatizado de muestras de malware.

En [21] para extraer el modelo propuesto, primero se realizó un análisis dinámico en un *dataset* de malware relativamente reciente dentro de un entorno virtual controlado y se capturaron las trazas de llamadas a las APIs. Luego, las trazas se generalizaron en *features* de alto nivel a las que se referenció como acciones. La viabilidad de las acciones se evaluó mediante varios algoritmos de clasificación como DT (*Decision Tree*), RF (*Random forest*) y SVM.

El método de análisis dinámico de interceptar llamadas API se conoce como *API hooking*, el cual implica la manipulación de un proceso de manera que cuando se llama a una función API específica para ser interceptada, la ejecución se redirige a otro código. Este código puede registrar información de llamadas a la API en un archivo de registro, analizar sus parámetros o incluso modificarlos. Después de eso, puede redirigir la ejecución de vuelta a la función original. El *API hooking* introduce una penalización de rendimiento en el sistema. Por lo tanto, solo deben interceptarse las funciones más frecuentes utilizadas por el malware. Estas funciones pueden clasificarse en seis categorías: Procesos e Hilos, Gestión de Memoria, Archivos, Registro, Red e Internet.

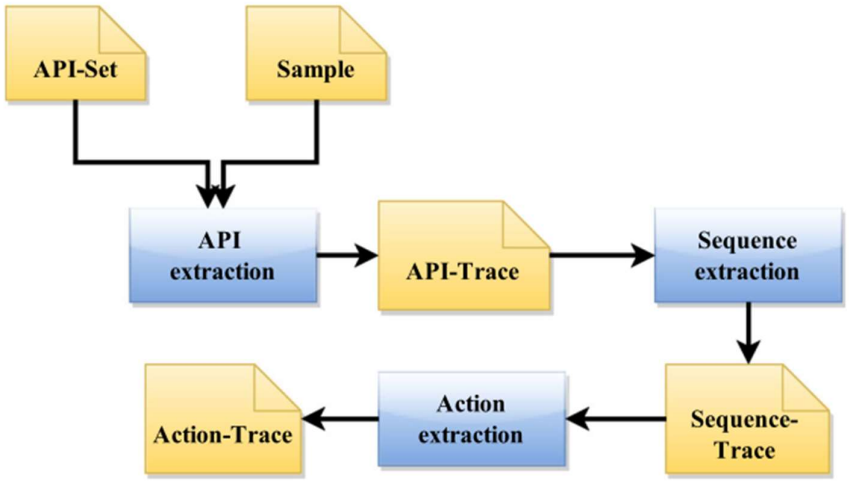

Ilustración 8: *Etapas del análisis dinámico*

La interceptación de información de llamadas API de una muestra se lleva a cabo dentro de una máquina virtual. Una API puede ser llamada por el módulo principal de un proceso, así como por los módulos del sistema importados. Cabe destacar que solo se capturan las APIs invocadas por el módulo principal.

Además, también se capturan las llamadas invocadas por los procesos secundarios creados por la muestra. Las siguientes tareas se llevan a cabo durante esta etapa:

1. Copiar la muestra al sistema operativo invitado de la máquina virtual.
2. Instalar las intercepciones en las categorías previamente descriptas y ejecutar la muestra.
3. Terminar la muestra y sus procesos secundarios asociados luego de cinco minutos.
4. Emitir una lista de las llamadas API interceptadas junto con los valores de los parámetros.
5. Revertir la máquina virtual a una instantánea limpia.

La salida se denomina API-Trace que consiste en una tupla con el formato $(\mathrm{l}, \mathrm{a}, \mathrm{r}, \mathrm{p}_1, \ldots \mathrm{p}_\mathrm{n})$, donde l es el número de línea; a es el nombre de la función; r es el valor devuelto y p_i es el i −ésimo parámetro.

Se identificaron dependencias de flujo de datos entre llamadas a la API para crear secuencias de llamadas, dividiéndolas en cuatro categorías:

- La categoría de creación de *Handles* consta de funciones de la API que abren o crean un objeto en el sistema operativo Microsoft Windows. El valor de retorno de estas funciones es el identificador que se utiliza para identificar el objeto.
- La categoría que depende del *Handle* consta de funciones que operan en objetos y requieren el valor del *Handle* como parámetro de entrada.
- La categoría de liberación del *Handle* consta de funciones que liberan los recursos asociados con el objeto obtenido cuando ya no hay más operaciones que realizar.
- La categoría simple consta de funciones que no requieren de un *Handle*.

Se usó el valor del *Handle* para identificar la dependencia de datos entre las llamadas de las tres primeras categorías y las llamadas de la última categoría

se representaron en secuencias separadas. El conjunto de secuencias se llama *Sequence-Trace*.

Para extraer acciones de las trazas de secuencia, se utilizaron un conjunto de funciones heurísticas que infieren acciones únicas. La función heurística selecciona una secuencia de llamadas a la API según su categoría.

Las acciones consisten en campos con un valor semántico que describe el comportamiento de una muestra. La colección de acciones para una muestra determinada se conoce como *Action-Trace*, donde cada acción se describe como $(N, F_i = V_i, \dots)$ tal que N es el nombre, F_i y V_i son el campo y el valor i-ésimo, respectivamente.

La siguiente tabla muestra los resultados que se obtuvieron:

ALGORITMO	SENSIBILIDAD	ESPECIFICIDAD	PRECISIÓN	AUC
DT	93,3%	96,53%	97,19%	97,65%
RF	97,19%	96,35%	96,84%	99,48%
SVM	92,28%	96,35%	93,98%	98,55%

Tabla 1: *Resultados del experimento*

En [22] se propuso una metodología que consta de dos etapas:

- La Etapa 1 comprende dos pasos: capturar las llamadas API de muestras benignas y maliciosas y aplicar la técnica de selección de *features* a las secuencias de API para obtener secuencias distintas de APIs.
- La Etapa 2 describe el uso de algoritmos de aprendizaje automático y algoritmos de clasificación para clasificar una muestra en maliciosa o benigna.

En el paso A, se utilizó un programa premodelado para realizar un seguimiento de la ejecución de las muestras de entrada y capturar las llamadas a la API de estas muestras para obtener las secuencias de llamadas.

En el paso B, se seleccionaron las secuencias de APIs distintas en dos partes; generando los N-gramas de llamadas, calculando el *odd ratio*[2] de cada uno y por último generando un vector de *features*.

Las secuencias del paso anterior no se pueden utilizar con fines de clasificación. Por lo tanto, propusieron un algoritmo de selección de *features* basado en N −gramas de llamadas y en el algoritmo de selección *odd ratio (OR)*.

- Generar los N-gramas de llamadas: se genera un 1-grama de API utilizando las secuencias de llamadas obtenidas en el paso A y desplazando la ventana se pueden obtener 2, 3 y hasta 6-gramas. El análisis experimental demostró que los 2-gramas ofrecen una mejor precisión en comparación con 1 -grama y además proporciona una mejora adicional respecto con los 3-gramas. En general, los 4-gramas son los que ofrecen mejores resultados.
- Calcular el OR para cada N-grama y generar un vector de *features*: hubo que seleccionar aquellos N-gramas distintos y útiles, de los muchos obtenidos en el paso anterior. Cada N-grama obtenido se consideró como una *feature* X_j y el OR para una *feature* X se puede calcular como $OR(X) = \log\frac{\boldsymbol{Pr}(x|c)(1-\boldsymbol{Pr}(X|\bar{c})}{\boldsymbol{Pr}(x|\bar{c})(1-\boldsymbol{Pr}(X|c)}$; donde $\boldsymbol{Pr}(X|c)$ es la probabilidad de que la *feature* $\boldsymbol{X}$ aparezca en la clase $\boldsymbol{c}$ (análogamente con $\boldsymbol{Pr}(X|\bar{c})$)**.** c y $\bar{c}$ representan a dos clases, por ejemplo, *benigna* y *maliciosa*. Por último, queda *rankear* las *features* de acuerdo con el OR calculado y seleccionar aquellas N principales para formar el vector.

Para construir el modelo para la clasificación se utilizaron varios algoritmos: Naïve Bayes, Random, Decision Tree y SVM.

Se extrajeron distintas secuencias API utilizando el algoritmo de selección de *features* propuesto y se alimentaron a todos los mencionados algoritmos de clasificación. El estudio concluyó que SVM ofrece los mejores resultados en comparación con los algoritmos utilizados.

[2] El odd ratio es una medida de asociación entre dos variables que indica la fortaleza de la relación entre dos variables.

En [3] se propone un mapeo de la minería OOA y la detección de *malware*.

En el problema que se plantea, DB está compuesta por un conjunto de muestras benignas y maliciosas. Cada muestra puede ser vista como un registro, que se representa como llamadas al sistema extraídas de la cabecera PE de un ejecutable. Por lo tanto, un elemento i_k en el problema es una llamada al sistema. Un conjunto de elementos I está compuesto por un conjunto de diferentes llamadas al sistema. Dado que el objetivo es reconocer malware, se definió solo un atributo objetivo (denominado $label - r$), $label - r = 1$ para malware y $label - r = 0$ para archivos benignos.

Según la definición de OOA, se tiene $class^{+}(label - r) = \{label - r = 1\}$, $class^{-}(label - r) = \{label - r = 0\}$, $class^{+}(DB) = \{label - r = 1(u_{r-label=1})\}$ y $class^{-}(DB) = \{label - r = 0(u_{r-label=0})\}$.

La utilidad de una regla de asociación se puede calcular según las fórmulas de la sección 1.3. La parte clave de la minería OOA es cómo definir los valores de utilidad de $label - r$.

En su trabajo, se obtienen las llamadas a la API extrayéndolas de la cabecera PE. Como mucho del *malware* está empaquetado, primero, se desempaqueta. En caso de que no sea posible, no se considera. Aunque todas las instancias de malware empaquetado fueran posible desempaquetarlas, aún no se podría confirmar que las APIs fueran realmente llamadas por el *malware*. En algunos casos, los creadores podían agregar APIs nunca utilizadas en la tabla de importaciones para dificultar el análisis. Con el fin de contrarrestar este efecto, se desensambló el ejecutable y, a partir de la información de referencia cruzada correspondiente en la sección *.idata*, se descubrió no solo cuántas veces se llamaba una API, sino también en dónde se llamaba. Solo se consideraron aquellas APIs que eran realmente llamadas por un ejecutable.

Las APIs se seleccionaron de acuerdo con los siguientes criterios:

1. Tomar aquellas APIs que tengan valores de distribución altos porque es más probable que sean elementos frecuentes.
2. Elegir aquellas APIs que tengan una gran capacidad de clasificación. Para ello, se calculó la ganancia de información (IG)[3]

[3] La ganancia de información (IG) es la reducción de la entropía causada por la partición de una colección de muestras según una feature. La IG es una medida de la efectividad de una feature en la clasificación de muestras.

para cada API del *dataset* y se seleccionaron las 1000 APIs principales que tengan una alta IG.

Si se calcula la utilidad positiva y negativa de DB para I, se encontrará que la utilidad de DB para I estará sesgada hacia aquella categoría que tenga un mayor número de muestras. El atributo objetivo $label - r$ se define como la categoría de un registro.

Por medio de las fórmulas (14) y (15) de la sección 1.3, es posible ver que el valor de utilidad de DB para I, ($u_{+D}\ (I)$ y $u_{-DB}(I)$), depende directamente del número de muestras en cada categoría. Tanto los archivos maliciosos como los benignos suelen tener un gran número de llamadas al sistema comunes. Para hacerle frente a este sesgo, se propuso el apoyo objetivo de un conjunto de elementos I. El apoyo objetivo positivo para I ($supp_{+DB}(I)$) y el correspondiente apoyo objetivo negativo para I ($supp_{-DB}(I)$) se definen como:

$$supp_{+DB}(I) = \frac{u_{+DB}(I)}{count(\{r - label = 1\}, DB)} \tag{38}$$

$$supp_{-DB}(I) = \frac{u_{-DB}(I)}{count(\{r - label = 0\}, DB)} \tag{39}$$

Se puede observar que el apoyo objetivo es un valor promedio. En este caso, se definen $u_{r-label=1} = 1$ y $u_{r-label=0} = 1$; el apoyo objetivo de I es la probabilidad de que I ocurra en una cierta categoría de registros.

Se define la utilidad de una regla de asociación $I \rightarrow (r - label = 1)$ como:

$$u = (supp_{+DB}(I) - supp_{-DB}(I)) \times 100\%$$

Para mantener un número suficiente de reglas de asociación, se estableció que la utilidad mínima (u_{min}) sea cero. Esto implica que se espera que la probabilidad de que un conjunto I ocurra en archivos malignos sea mayor que la probabilidad de que ocurra en archivos benignos.

Se ha encontrado que ciertos conjuntos de elementos son comunes tanto en *malware* como en archivos benignos. Este hallazgo implica que los

mencionados conjuntos de elementos tienen un alto valor de utilidad (u), por ende, las reglas de asociación no pueden distinguirlos. En consecuencia, se ha establecido un umbral mínimo llamado "soporte objetivo" (min_obj_supp) para determinar la frecuencia de aparición de estos conjuntos en ambas categorías. Si se supera ese umbral ($supp_{+DB}(I) \geq min\_obj\_sup \vee supp_{-DB}(I) \geq min\_obj\_sup$), se eliminan esas reglas de asociación.

En este estudio, se utilizó el algoritmo *OOA_FP-Growth* para extraer las reglas de asociación, aprovechando una estructura de árbol FP para agilizar el proceso.

Se propusieron criterios para la selección de API y criterios para la selección de reglas de asociación, con el objetivo de reducir el número de reglas y mejorar la calidad de estas.

Debido a la disminución en el número de reglas de asociación, la velocidad de detección del método propuesto es aproximadamente dos veces más rápida que la del algoritmo de minería OOA original. Como resultado, el método propuesto es efectivo para mejorar la velocidad de ejecución de la minería OOA para la detección de malware.

En [23] se propone un método para generar un conjunto de *features* robusto basado en una combinación de argumentos y valores de retorno de las APIs utilizadas por el *malware*.

La solución propuesta consiste en tres fases: en la primera, se ejecutan los binarios y se monitorea su comportamiento. Se vigilan las llamadas a la API, los argumentos de entrada y salida, y los valores de retorno. Los resultados son preprocesados y se generan conjuntos de *features* basados en combinaciones de llamadas de API y los argumentos y/o valores de retorno. Dado que se genera un gran conjunto de *features*, este conjunto se reduce mediante técnicas de selección. Finalmente, se construyen modelos para realizar la clasificación. En la tercera fase, se evalúa cuán predictivo es el modelo construido utilizando binarios que no habían sido vistos en la fase de entrenamiento.

Los conjuntos de *features* modelan comportamientos maliciosos y benignos. Estos conjuntos tienen valores binarios.

El primer conjunto es una representación de las APIs llamadas junto con sus valores de retorno, llamado API-RET. El segundo se llama API-ARG y

representa el nombre de la API junto con sus argumentos. En este conjunto, el nombre de la función se corresponde con el primer elemento, el segundo elemento indica el orden del argumento de entrada y el tercer elemento representa el valor correspondiente del argumento. Por último, el tercer conjunto es una combinación de API-RET y API-ARG, por lo que lo llaman API-ARG-RET. Puede parecer que este último es un superconjunto de los dos primeros. Sin embargo, el solo contiene ciertas combinaciones de API-RET y API-ARG. Por lo tanto, no es un superconjunto.

Para todas las llamadas al sistema, se pueden extraer los valores de los parámetros de entrada y el valor de retorno, y se construyen las tres categorías de *features*. Después de construirlas, se verifica la presencia de cada una en cada archivo binario y se muestra en un vector. Si la *feature* seleccionada está disponible en el archivo de registro, el valor se establece en 1; de lo contrario, se establece en 0.

Durante la supervisión del comportamiento se monitorean muchos parámetros. El preprocesamiento reduce el error de generalización de los modelos aprendidos, ya que se eliminan aquellas *features* irrelevantes y redundantes. En la etapa de supervisión se generan demasiadas *features*. Por ejemplo, una técnica de polimorfismo llamada inserción de código basura crea muchas *features* innecesarias que no son discriminativas para la detección de *malware*. Además, se pueden insertar varias *features* engañosas en binarios maliciosos para dificultar su detección.

El hecho de invertir mayores esfuerzos en la selección de *features* tiene como resultado una mejor obtención de resultados. En la primera fase, se utiliza la escala de *Fisher*. En la segunda etapa, se utiliza un algoritmo de selección de *features* basado en SVM (SVM-RFE). Aquellas *features* resultantes del proceso, se consideran mucho más discriminativas y conllevan a un margen de separación de clases mayor.

Los algoritmos de clasificación ayudan a descubrir patrones discriminativos en los conjuntos de datos. Estos patrones descubiertos se utilizan para clasificar muestras no vistas.

Los vectores binarios generados en la etapa anterior se utilizan como entrada para los algoritmos de clasificación. En este estudio se utilizan varios

clasificadores conocidos, como *Random Forest* (RF), Árboles de Decisión J48, Optimización Secuencial Mínima (SMO) y Regresión Logística Bayesiana (BLR).

Los clasificadores se evaluaron utilizando el procedimiento de *cross-validation* de 10 *folds* (pliegues) en todos los experimentos para evitar el *overfitting*. Se calculó el *F-Measure* en cada ejecución y, finalmente, se tomó el promedio como criterio de evaluación.

En este estudio, no se monitorizaron todas las llamadas a API; solo se registró un subconjunto de 126 llamadas a API provenientes de las seis DLL más importantes. Las DLL importantes son: *advapi32.dll*, *kernel32.dll*, *ntdll.dll*, *user32.dll*, *wininet.dll* y *ws2_32.dll*. Estas llamadas se seleccionaron de una investigación anterior. Se recopilaron en función de las categorías de Acceso al Sistema de Archivos, Información del Sistema, Redes, Acceso al Registro y Procesos.

Los resultados sugieren que el conjunto API-RET es más efectivo en *datasets* más grandes, mientras que el conjunto API-ARG-RET funciona mejor dentro de un mismo *dataset*. Esto se explica porque al considerar tanto la combinación de los argumentos como los valores de retorno, se puede modelar el comportamiento de manera más precisa. Sin embargo, es importante tener en cuenta que los argumentos de entrada pueden variar ampliamente entre nuevas familias con comportamientos diferentes, mientras que los valores de retorno tienen un rango más limitado y son más estables. En resumen, el conjunto API-RET ofrece un modelo más general del comportamiento, mientras que el conjunto API-ARG-RET es más específico. En situaciones de aprendizaje incremental, API-ARG-RET proporciona resultados más sólidos al basarse en menos parámetros. Por otro lado, el uso de API-RET sería más adecuado cuando los conjuntos de *features* no se actualizan, ya que pueden detectar automáticamente algunas muestras de las nuevas familias.

Los resultados demostraron que las *features* creadas fueron altamente discriminatorias y robustas, independientemente del *dataset* que se utilice.

Capítulo 3: Deep learning aplicado a Detección de malware

3.1 Representación en vectores de features

En [24] se plantea el uso de proyecciones aleatorias sobre cada vector de entrada para hacer el problema más manejable.

Se extraen tres tipos de *features*, que incluyen patrones terminados en nulo (mayormente cadenas) que se observan en la memoria, trigramas de llamadas a la API del sistema y distintas combinaciones de una única llamada a la API y un parámetro de entrada. Los trigramas de API y sus parámetros se utilizan para analizar el comportamiento dinámico del archivo. Estos consisten en tres llamadas consecutivas a la API, los cuales se pueden utilizar para identificar familias individuales de *malware*. Dado que los datos en bruto son generados por un motor antimalware en producción, se utilizan atributos que se pueden extraer eficientemente en tiempo real.

Se proyectó cada vector de entrada en un espacio de dimensionalidad mucho más baja utilizando una matriz de proyección dispersa $\mathcal{R}$ con entradas muestreadas independientemente de una distribución en $\{-1, 0, 1\}$. Las entradas de 1 y -1 tienen igual probabilidad, y $\boldsymbol{Pr}(R_{ij} = 0) = 1 - \sqrt{1/d}$, donde d es la dimensionalidad de entrada original. Otra forma de ver la proyección aleatoria en el contexto del entrenamiento de redes neuronales es que el paso de proyección aleatoria forme una capa con unidades ocultas lineales en la cual la matriz de pesos no se aprende y, en cambio, simplemente se establece como $\mathcal{R}$.

Para el entrenamiento, se utilizó un 10% del total de las muestras y los sistemas se entrenaron con 40 *epochs*.

Los tamaños de todas las capas ocultas (*hidden layers*) para las redes neuronales fueron de 1536, 2048 y 1024. En los resultados de la red neuronal, se utilizó un *momentum* de entrenamiento con un valor de 0,9 y la tasa de aprendizaje (η) se estableció en 0,03. El número de proyecciones aleatorias se fijó en 4000, lo cual tuvo un rendimiento constante. Asimismo, se incluyó una etapa adicional de preentrenamiento que suele utilizarse en arquitecturas de *deep learning*. Esta etapa tiene como objetivo inicializar los pesos de la capa oculta antes de llevar a cabo el entrenamiento estándar de *backpropagation* de

la red neuronal con SGD para mejorar el aprendizaje. En este trabajo, se implementó el preentrenamiento utilizando una máquina de Boltzmann restringida (RBM) de tipo Gaussiano-Bernoulli.

Empíricamente, se detectó que añadir más capas ocultas no mejoró los resultados. Con la excepción del FNR para la red neuronal de una capa con preentrenamiento, todas las métricas de error son ligeramente peores para las redes preentrenadas en comparación con las respectivas versiones sin preentrenamiento.

Efectuar el preentrenamiento después de las proyecciones aleatorias no es algo que se esperaría que sea útil, ya que requiere el uso de RBM Gaussianas-Bernoulli, las cuales son más difíciles de manejar y podrían aprender incorrectamente correlaciones artificiales introducidas por la mezcla de *features* de las proyecciones aleatorias.

A medida que se utilizan más dimensiones, el clasificador tiene más pesos aprendidos y más información sobre la entrada, lo que refleja una mejora en su rendimiento. Aumentar el número de dimensiones de la proyección aleatoria aumenta la capacidad del modelo al proporcionarle más parámetros ajustables y también proporciona más información sobre los datos. Estos dos efectos están estrechamente relacionados en la regresión logística en comparación con las redes neuronales con número variable de unidades ocultas, por lo que se espera que el número óptimo de dimensiones de la proyección aleatoria para una red neuronal sea como máximo el número óptimo para la regresión logística.

Se encontró que los resultados fueron relativamente insensibles al número de unidades ocultas en la capa oculta de las redes neuronales. Siempre y cuando la capa oculta fuera lo suficientemente grande se logran resultados muy buenos.

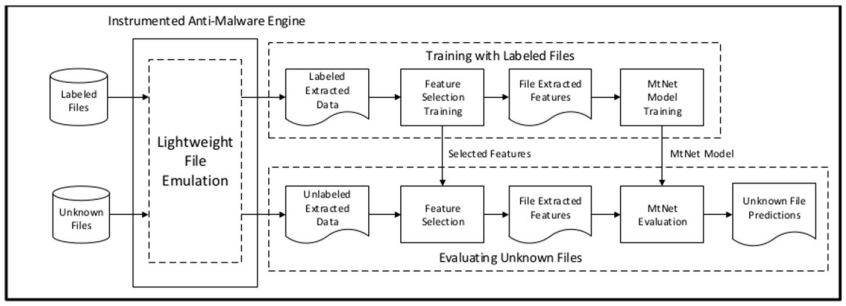

Ilustración 9: *Vista a alto nivel de MtNet*

En [25] se propone un sistema denominado MtNet basándose en el esquema propuesto anteriormente. La **Ilustración 9:** Vista a alto nivel de MtNet muestra una visión general del proceso de entrenamiento del sistema MtNet y la evaluación de archivos desconocidos con el modelo entrenado. La fila superior presenta los pasos necesarios para identificar las *features* seleccionadas y entrenar el modelo MtNet, mientras que la fila inferior indica el proceso para evaluar un archivo desconocido dado un conjunto de *features* seleccionadas y el modelo MtNet entrenado.

Para el entrenamiento, se extrajeron los datos sin procesar de archivos etiquetados durante el análisis dinámico mediante una versión modificada de un motor antimalware. A diferencia de la emulación en profundidad ejecutada en una máquina virtual, el motor antimalware utilizado en este estudio solo proporciona una emulación ligera del sistema operativo e intenta provocar la ejecución del archivo. Dado que los motores antimalware están diseñados para escanear rápidamente archivos desconocidos en busca de virus, se pueden evaluar muchos más archivos con este método que utilizando máquinas virtuales completas. Una vez que se ha recopilado los datos sin procesar de los archivos etiquetados para el conjunto de entrenamiento, se realiza el entrenamiento de selección de *features* para producir las *features* binarias dispersas finales necesarias para el entrenamiento de MtNet. A continuación, se entrenó el modelo MtNet para dos tareas, incluida la clasificación binaria que determina si un archivo desconocido es malicioso o benigno, y la clasificación de familias de 100 clases que ubica el archivo entre una de las 98 familias importantes, o una clase genérica de malware o en la clase benigna. Para obtener las categorías, los analistas proporcionaron etiquetas para decenas de miles de familias de

malware individuales. Sin embargo, se seleccionaron 98 familias para el clasificador de familias en función de su gravedad y prevalencia de infección. Los archivos pertenecientes a las familias restantes se asignan a la clase genérica. Todos los archivos que no son malware pertenecen a la clase "Benigno". Luego del entrenamiento, las *features* seleccionadas se utilizan para limitar aquellas que fueron extraídas mediante la emulación de archivos desconocidos, y estas pueden evaluarse mediante el modelo MtNet entrenado. La puntuación de predicción binaria de MtNet se utiliza para clasificar automáticamente el archivo desconocido como malicioso o benigno. Del mismo modo, el clasificador de familias intenta asignar una etiqueta de familia específica al archivo desconocido.

Para cada archivo ejecutable que es emulado por el motor antimalware, se extrajeron dos conjuntos de información: una secuencia de eventos de llamadas a la API junto con sus parámetros, y una secuencia de objetos terminados en nulo recuperados de la memoria del sistema durante la emulación. Un gran porcentaje de archivos maliciosos están comprimidos. Durante el proceso de descompresión, el malware a menudo escribe objetos terminados en nulo en la memoria del sistema. Se descubrió que la mayoría de los objetos terminados en nulo son, de hecho, cadenas descomprimidas, pero algunos corresponden a fragmentos de código individuales.

Para el flujo de API y parámetros, se utiliza una asignación de uno a muchos para representar los eventos de API. En total, hay 114 eventos de API de alto nivel en los datos.

Se derivan tres conjuntos de *features* binarias dispersas de las dos fuentes de datos. Una *feature* binaria dispersa se establece si la misma está presente en los datos; no se utilizan recuentos de *features* en MtNet para evitar detecciones perdidas debido a que los atacantes varían polimórficamente el número de *features* críticas. La presencia o ausencia de los objetos terminados en nulo se utiliza directamente como uno de los conjuntos de *features*. Dos conjuntos adicionales se derivan del flujo de API y parámetros. El primer conjunto se deriva de cada combinación única de evento de API de alto nivel y una configuración de parámetros de entrada individual. Como resultado, se generan varias *features* binarias dispersas a partir de cada llamada de API. El segundo

conjunto consiste en trigramas de eventos de API que se generan mediante la combinación única de tres eventos de API consecutivos.

Al comparar los resultados del modelo propuesto por [24] con esta implementación, se observa que el modelo de línea de base de mejor rendimiento en la implementación utiliza tres capas ocultas en comparación con una en el estudio original. Varios factores cambiaron entre estos dos experimentos. Los tamaños de los conjuntos de entrenamiento y prueba se duplicaron, el número de *features* y familias disminuyó y la implementación subyacente cambió por completo. Además, en [24] solo se entrenaron modelos basados en familias y los resultados de clasificación binaria se calcularon en función de si la familia predicha era maliciosa o no. En este estudio, los modelos de 2 clases se entrenaron con las etiquetas binarias verdaderas. Es interesante que las tasas de error de prueba más bajas para las dos implementaciones sean esencialmente idénticas (es decir, 0,49% para su implementación y 0,4845% para esta). Además del tamaño de la capa oculta, esta versión de una sola tarea de MtNet difiere de [24] en dos aspectos: la función de activación sigmoidal se reemplaza por la función ReLU y se incluye *dropout*. Tanto para la clasificación binaria como la de familias, estos modelos de una sola tarea mejoran significativamente los resultados de la línea de base en un 23,98% y un 19,21%, respectivamente. Estos resultados indican que cambiar a la función de activación de ReLU y agregar *dropout* ayuda al modelo a aprender una mejor representación de las *features* del archivo. Se descubrió también que, aunque agregar *dropout* a las capas ocultas generalmente aumenta la cantidad de *epochs* de entrenamiento requeridas, ReLU acelera la convergencia del proceso de SGD para la clasificación binaria. En comparación con las funciones de activación sigmoideas, ReLU reduce significativamente el número de iteraciones necesarias para entrenar un clasificador binario.

En [26] se propone un *framework* de clasificación. Este consta de tres componentes principales: el primero extrae cuatro tipos diferentes de *features* complementarias de los binarios estáticos benignos y maliciosos; el segundo componente es un clasificador basado en redes neuronales que consta de una capa de entrada, dos capas ocultas y una capa de salida; el último es un calibrador de puntuación, que traduce las salidas de la red neuronal en una

puntuación que puede interpretarse de manera realista como una aproximación de la probabilidad de que el archivo sea realmente malware.

Los primeros conjuntos de *features* que se calculan para los binarios de entrada son los valores binarios de un histograma bidimensional de entropía de bytes que modela la distribución de bytes del archivo. Para extraer el histograma de entropía de bytes, se desliza una ventana de 1024 bytes sobre un binario de entrada, con un paso de 256 bytes. Para cada ventana, se calcula la entropía de la ventana y cada ocurrencia de byte individual en la ventana (1024 valores no únicos) con el valor de entropía calculado, almacenando los 1024 pares en una lista. Finalmente, se calcula un histograma bidimensional sobre la lista de pares, donde el eje de entropía del histograma tiene 16 contenedores de tamaño uniforme en el rango [0, 8], y el eje de bytes tiene 16 contenedores de tamaño uniforme en el rango [0, 255]. Para obtener un vector de *features* (en lugar de la matriz), se concatenan cada vector de fila en este histograma en un solo vector de 256 valores. Estas *features* permiten modelar el contenido del archivo de manera independiente al formato y pueden distinguir entre diferentes tipos de datos, como instrucciones x86 y datos comprimidos.

El tercer conjunto se deriva de la tabla de direcciones de importación del archivo binario. Se utiliza una matriz de 256 enteros para contar el número de apariciones de las tuplas de nombre de DLL y función de importación. Estas *features* capturan la semántica de las llamadas de función externa y pueden ayudar a detectar archivos sospechosos o aquellos que coinciden con una familia de malware conocida.

El último conjunto se extrae de los campos numéricos del PE. Cada campo numérico se agrega a una matriz de longitud 256. Estas *features* permiten identificar aspectos sospechosos en la estructura del programa binario y aprender firmas específicas de familias de malware.

Para construir el vector de *features* final, se concatenan los cuatro conjuntos de 256 dimensiones, lo que resulta en un vector de 1024 dimensiones. Esta reducción de datos en un vector de tamaño fijo proporciona resultados precisos y permite una carga y entrenamiento eficientes del modelo.

Para etiquetar los archivos binarios como malware o software legítimo, se ejecutan a través de VirusTotal[4] y se utiliza una estrategia de votación. Los archivos con una alerta del 30% o más de los motores antivirus se etiquetan como malware, mientras que los archivos sin ninguna alerta se etiquetan como software benigno. Se descartan los archivos con una clasificación incierta, es decir, aquellos en los que menos del 30% de los motores antivirus los consideran malware. No se filtran los archivos en función de su contenido real para evitar sesgos en los resultados.

Para la clasificación, se utiliza una red neuronal compuesta por cuatro capas. Las tres primeras capas tienen 1024 nodos y se les aplica la técnica de *dropout*, seguidas de una capa densa con una función de activación de unidad lineal rectificada paramétrica (PReLU) en las dos primeras capas y la función sigmoide en la última capa oculta. La cuarta capa es la capa de predicción. Estas decisiones de diseño se basan en la eficiencia y la prevención del *overfitting*.

Dado que el número de muestras binarias en el conjunto de datos es relativamente pequeño en comparación con todos los posibles archivos binarios que se pueden observar en una red empresarial grande, se busca aumentar la expresividad de la red manteniendo un tamaño manejable.

Para prevenir el *overfitting*, se utiliza la técnica de *dropout*, que ha demostrado ser un enfoque simple y eficiente. A diferencia de las regularizaciones de peso estándar, el *dropout* busca pesos en cada nodo que sean complementarios a los pesos en otros nodos, lo que evita la coadaptación y mejora la clasificación correcta en toda la red.

Para acelerar el aprendizaje, se utilizaron funciones de activación ReLU en lugar de funciones sigmoidales tradicionales como la tangente hiperbólica. Las ReLU han demostrado acelerar significativamente el entrenamiento de la red al evitar la desaceleración en la tasa de convergencia del descenso de gradiente. Además, se utilizó la función de activación PReLU para evitar el rendimiento deficiente cuando los valores de entrada son inferiores a 0, lo que mejora aún más la velocidad de convergencia.

La inicialización de los pesos antes del entrenamiento también puede afectar significativamente la convergencia del algoritmo de *backpropagation*. En

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

este caso, se utilizó una distribución gaussiana normalizada según el tamaño de entrada y salida de las capas. Además, se aplica el logaritmo en base 10 a los valores de las *features* antes de la inicialización de los pesos, lo que ha demostrado mejorar sustancialmente el rendimiento del entrenamiento en la práctica.

El sistema propuesto no solo detectó malware en un sentido binario, sino que también proporcionó probabilidades precisas de que un archivo dado sea malware. Para lograr esto, se utilizó un enfoque de calibración del modelo bayesiano que combina la creencia empírica sobre el "riesgo" de una red de clientes determinada e información empírica sobre el perfil de error de la red neuronal en los datos de prueba.

Para convertir la puntuación proporcionada por el clasificador en un puntaje de "amenaza", que se define como la probabilidad de que el archivo sea realmente malware ($\boldsymbol{P}(C = m|S = s)$) teniendo en cuenta la puntuación s y la categoría $C = \{m, b\}$. Dicho puntaje refleja en qué medida el clasificador cree que un binario observado es malware.

Supóngase que hay *pdfs* (función de densidad de probabilidad, por sus siglas en inglés) para los puntajes benignos y de malware para un clasificador dado, $p(S = s|C = m)$ y $p(S = s|C = b)$.

Dada la definición anterior del puntaje de amenaza, es necesario obtener las *pdfs* $p(s|m)$ y $p(s|b)$. Hay dos enfoques comúnmente utilizados:

i. el enfoque paramétrico, donde se asume alguna distribución para las *pdfs* y se ajustan los parámetros de esa distribución en función de las muestras observadas.
ii. el enfoque no paramétrico, como el estimador de densidad de núcleo (KDE, por sus siglas en inglés), donde se aproxima el valor de la *pdf* dada C tomando un promedio ponderado de la vecindad.

Dado que no es razonable esperar que la salida del clasificador de ML siga alguna distribución estándar, se utiliza KDE con el núcleo Epanechnikov. En las pruebas, se obtuvo una puntuación de validación mejor que el núcleo

gaussiano estándar. Dado que las *pdfs* solo pueden tomar valores en $[0, 1]$, se reflejaron las muestras a la izquierda de 0 y a la derecha de 1 antes de calcular el valor estimado de la *pdf* en un punto específico. El tamaño de la ventana se estableció empíricamente en 0,01 para aproximar mejor el extremo de la distribución, donde las muestras son menos densas.

Como conclusiones, se logra una tasa de detección del 95% y una tasa de falsos positivos del 0,1% en un conjunto experimental de más de 400.000 archivos binarios. Además, se ha demostrado que el enfoque propuesto requiere una computación modesta para realizar la extracción de *features* y puede lograr una buena precisión con una única GPU en tiempos razonables.

Los autores creen que el enfoque en capas de redes neuronales profundas y las *features* de histograma bidimensional proporcionan una categorización implícita de los tipos de archivos binarios, lo que les permite entrenar directamente con todos los binarios sin separarlos en función de características internas, como tipos de empaquetadores, entre otros.

Las redes neuronales también tienen varias propiedades que las hacen candidatas adecuadas para la detección de malware. En primer lugar, permiten un aprendizaje incremental, lo que significa que no solo pueden entrenarse en lotes, sino que también pueden ser reentrenadas eficientemente a medida que se recopilan nuevos datos de entrenamiento. En segundo lugar, les permite combinar datos etiquetados y no etiquetados mediante el preentrenamiento de capas individuales. En tercer lugar, los clasificadores son muy compactos, lo que permite realizar predicciones rápidamente utilizando poca memoria.

3.2 Trazas de llamadas a la API

La metodología descripta en [27] consiste en reunir y preprocesar los datos de entrada, extraer las *features* y clasificarlas por medio de una red neuronal.

En cuanto a la adquisición de datos, se recolectó un conjunto de muestras de malware de diversas fuentes durante varios meses. Para la extracción de datos a gran escala, se utilizó "*malware zoo*" que consume servicios REST para obtener datos de varias herramientas de análisis de malware. Se emplearon herramientas como *objdump* y *PEInfo* para obtener información de los encabezados PE y los *opcodes* de las secciones de código.

Luego, se realizó el preprocesamiento para extraer las *features* de los archivos. Se obtuvieron las de los metadatos, de los importados y de los *opcodes*. Luego, se convirtieron en vectores numéricos para ser utilizadas en el proceso de clasificación.

Para la clasificación, se diseñó una arquitectura de red neuronal híbrida que combina capas de redes neuronales de avance y convolucionales. Se utilizaron capas de avance para procesar los datos sin estructura que se beneficiarían de los filtros convolucionales. Las capas de convolución se aplicaron para aprender características de alto nivel a partir de las secuencias de *opcodes* de malware. La red neuronal se entrenó utilizando el algoritmo de SGD con momento de Nesterov.

El momento de Nesterov es un cambio sencillo con respecto al momento normal. En este último se computa el gradiente con base a la posición actual, mientras que en Nesterov se calcula con base a un punto intermedio entre el actual y el próximo punto en el tiempo. Esto es importante pues el gradiente siempre apunta en la dirección correcta, mientras que el momento no.

Además, se realizó un proceso de *clustering* de firmas para obtener etiquetas más confiables para la clasificación. Se agruparon las firmas de los archivos de malware en función de su presencia o ausencia y se seleccionaron aquellas familias de malware que sean más representativas para la clasificación.

En resumen, este estudio utilizó una combinación de técnicas de recolección de datos, preprocesamiento, extracción de *features* y clasificación mediante una red neuronal híbrida para la detección y clasificación de malware. Los resultados indicaron que este enfoque puede ser efectivo en la identificación de patrones de comportamiento en muestras de malware y su clasificación en familias específicas.

Se realizaron experimentos de validación cruzada para evaluar el rendimiento de las redes neuronales de avance, las redes neuronales convolucionales y la red neuronal híbrida combinada. Los resultados de clasificación se evaluaron utilizando métricas como precisión, *recall* y *F1-Score*.

Los resultados mostraron que la red neuronal híbrida proporciona una mejora en el rendimiento en comparación con las redes neuronales de avance y las redes neuronales convolucionales, así como con el clasificador de máquina

de soporte vectorial (SVM). La red neuronal híbrida logra un *F1-Score* de 0,92, con valores de precisión y *recall* de 0,93.

Los experimentos demostraron que el enfoque propuesto clasifica con precisión los ejecutables maliciosos que se comportan de manera similar en la misma familia. Sin embargo, se observó cierta confusión en la clasificación de ciertas clases de malware que comparten firmas comunes.

Además, se expuso que la red neuronal es resistente al reordenamiento de instrucciones, una técnica utilizada por los atacantes para crear variantes de malware y evadir la detección de firmas. La red neuronal conserva su capacidad de clasificación incluso cuando se reordenan las secuencias de *opcodes* hasta un 50%.

En resumen, los resultados mostraron que la red neuronal híbrida es efectiva en la clasificación de malware y que es robusta frente a técnicas de evasión, lo que la convierte en una herramienta prometedora para la detección y clasificación de amenazas de seguridad informática.

En el trabajo descripto en [28], los autores proponen usar una red neuronal recurrente estándar (RNN) [7] o una *echo state network* (ESN) como el modelo de lenguaje. Cabe señalar que la estructura de una ESN es similar a la de una RNN. La diferencia es que los pesos se inicializan al azar y no son entrenados. Luego, se entrena un clasificador de regresión logística o un MLP basado en las representaciones de las *features* generadas por el modelo de lenguaje. Además, se propone el uso de agrupamiento máximo temporal para ambos modelos de lenguaje recurrentes para combinar una larga secuencia de *features* temporales, lo que mejora los resultados.

El objetivo principal del trabajo es encontrar arquitecturas neuronales que brinden un mejor rendimiento. En particular, se utilizan arquitecturas de RNN que sean capaces de capturar mejor las dependencias que las RNNs estándares. En este estudio, se revisa la arquitectura del modelo de lenguaje para el malware para investigar si estos modelos de lenguaje mejorados proveen mejores resultados en la clasificación de malware. En particular, se experimentó con el modelo LSTM y la unidad recurrente con puertas (GRU). Se compararon todas las variantes de modelos con agrupamiento máximo temporal y un mecanismo de atención propuesto recientemente. Por último, también se consideró si la red

neuronal convolucional a nivel de caracteres (CHAR-CNN) puede mejorar el rendimiento de la clasificación de malware.

Se utilizó una versión modificada del motor antimalware de Microsoft para realizar análisis dinámicos de 75,000 archivos en formato PE, divididos equitativamente entre malware y archivos benignos. El *dataset* se dividió en tres conjuntos de manera aleatoria. Estos incluyen un conjunto de entrenamiento con 50.000 archivos; un conjunto de validación con 10.000 archivos y, por último, un conjunto de prueba con 15.000 archivos.

Antes de permitir que un archivo se ejecutase en una computadora, fue analizado mediante emulación de archivos ligera por el motor antimalware. Dicha emulación es un tipo de análisis dinámico y produce la secuencia de llamadas al sistema ejecutadas por el archivo. El motor antimalware modificado registra esta secuencia para entrenar y probar el clasificador. Estas llamadas al sistema que se registraron son eventos de alto nivel. En total, hay 114 llamadas. Cada evento se convierte primero en una secuencia de enteros que van del 0 al 113, donde cada entero es el índice de un evento de llamada al sistema individual. Esta secuencia de enteros luego es ingresada en el sistema de clasificación de malware.

Todos los modelos se implementaron utilizando *Keras*[5] con el motor de aprendizaje profundo Theano.

En el experimento, se dividieron todas las secuencias de eventos en subsecuencias más pequeñas con una longitud máxima de 50. Se entrenó el modelo utilizando un tamaño de lote de 50 y un máximo de 15 *epochs*.

Para los clasificadores de regresión logística y MLP, se escanearon secuencias de hasta 200 eventos para obtener las *features*. Los clasificadores se entrenan durante un máximo de 20 *epochs*.

El modelo CHAR-CNN se entrenó utilizando una secuencia de caracteres de longitud máxima de 1.014.

En cuanto al rendimiento de los modelos, se compararon las variantes de modelos basados en ESN y RNN con el mecanismo de atención y el agrupamiento máximo temporal.

[5] https://keras.io/

En general, los clasificadores basados en regresión logística mostraron resultados ligeramente mejores que las versiones MLP con el mismo modelo de lenguaje.

El mejor modelo en general en este estudio es el que utiliza una red LSTM con agrupamiento máximo temporal y regresión logística como clasificador. El modelo CHAR-CNN tiene un rendimiento significativamente peor en bajos FPR, pero se convierte en el tercer mejor modelo a un FPR de 1,9%.

La arquitectura que obtiene mejores resultados es una ESN con agrupamiento máximo temporal para generar la representación de *features*, combinada con un clasificador de regresión logística.

Desafortunadamente, los autores encontraron que la RNN no fue capaz de aprender las *features* relevantes de los archivos y tuvo un rendimiento inferior en comparación con la ESN no entrenada.

Este estudio presenta varias contribuciones en arquitecturas de aprendizaje profundo para clasificar malware, incluidos modelos de lenguaje basados en LSTM y GRU, así como una red CHAR-CNN. Se muestra que las *features* aprendidas de un modelo de lenguaje LSTM ayudan a mejorar el rendimiento en comparación con arquitecturas con pesos aleatorios, y que el modelo LSTM con agrupamiento máximo temporal supera a otros modelos competidores.

3.3 Representaciones basadas en secuencias de bytes

En [29] se plantea un enfoque de análisis estático en lugar de uno dinámico para detectar malware en archivos binarios. El enfoque utiliza una red neuronal que procesa secuencias de bytes del archivo para determinar cuán malicioso es. El uso de redes neuronales ha demostrado tener éxito en otros dominios, como procesamiento de imágenes y señales y problemas de texto. Sin embargo, el dominio de detección de malware presenta desafíos únicos, como la naturaleza contextual de los bytes en el malware, la correlación espacial en el contenido del archivo, la longitud extensa de las secuencias de bytes y la evolución constante del malware.

La contribución principal es que se desarrolló una arquitectura de red que puede procesar largas secuencias de bytes. Se identificaron los desafíos de

entrenar un modelo de detección de malware a partir de bytes sin conocimiento previo del dominio, y se demostró que este enfoque aprende una amplia variedad de tipos de información en comparación con otros métodos que requieren del conocimiento del dominio.

Para el trabajo, se utilizaron *datasets* de entrenamiento y prueba de dos grupos diferentes, que representan archivos de malware y archivos benignos. Además, se obtiene un corpus de entrenamiento más grande para mostrar que la nueva arquitectura sigue mejorando con más datos de entrenamiento, en comparación con un enfoque basado en gramáticas de bytes.

En resumen, el estudio propone un enfoque basado en redes neuronales para detectar malware a partir de secuencias de bytes, superando desafíos específicos del dominio. Los resultados demostraron la capacidad de generalización del modelo en *datasets* distintos y revelaron mejoras en el rendimiento utilizando un corpus de entrenamiento más grande con la nueva arquitectura propuesta.

Los autores deseaban diseñar un modelo que tuviera tres características principales:

i) capacidad para escalar adecuadamente con la longitud de la secuencia.
ii) habilidad para considerar tanto el contexto local como global al examinar un archivo completo.
iii) capacidad explicativa para facilitar el análisis del malware detectado.

El modelo resultante, denominado *MalConv*, utiliza una arquitectura de red convolucional para abordar la alta variación posicional presente en los archivos ejecutables.

En los archivos PE, existe una gran cantidad de reorganización de contenido, lo que representa un desafío para el modelo. Para abordar esto, se optó por utilizar una arquitectura de red convolucional y una operación de *max-pooling* global para permitir que el modelo produzca sus activaciones independientemente de la ubicación de las *features* detectadas. En lugar de realizar convoluciones directamente sobre los valores de bytes brutos (que van de 0 a 255), se empleó una capa de *embedding* que asigna cada byte a un vector

de *features* aprendidas de longitud fija. Esto le proporciona al modelo robustez ante alteraciones menores en los valores de bytes y una mayor capacidad para activarse ante diversos patrones de entrada.

Uno de los desafíos con los que se enfrenaron en el diseño fue la memoria consumida por la GPU debido a la longitud de las secuencias de entrada. Para abordar esto, se eligió una arquitectura superficial con filtros de convolución amplios y pasos agresivos. Sin embargo, la regularización se terminó convirtiendo en un desafío debido a la tendencia del modelo hacia el *overfitting*. Se implementó una regularización denominada *DeCov*, que penaliza la correlación entre las activaciones en la penúltima capa.

Se realizaron pruebas con numerosos diseños de arquitectura alternativos para abordar el problema, incluyendo hasta 13 capas de convolución, el uso de RNNs (bidireccionales) y diferentes modelos de atención[6].

El modelo *MalConv* tiene acceso al archivo completo, lo que le permite detectar las pocas *features* informativas independientemente de su ubicación, lo cual es esencial en casos donde programas generalmente benignos pueden tener malware inyectado en ellos.

La elección de usar *max-pooling* temporal en lugar de promedio fue otra consideración importante debido a la escasez de información en los datos. *Max-pooling* proporcionó una interpretación más clara y un mejor rendimiento en comparación con el promedio.

El uso de RNNs después de las convoluciones resultó en una disminución del rendimiento debido a la imposición de una suposición de que los datos convolucionales deben producir patrones de activación consistentes en frecuencias fijas, lo cual no era factible de aprender para el modelo.

En cuanto a los resultados, el modelo *MalConv* obtuvo un alto rendimiento y AUC, y tuvo la menor diferencia de rendimiento entre los distintos conjuntos de prueba, lo que indica una buena generalización de las *features* aprendidas.

Se realizó un análisis manual utilizando la arquitectura diseñada para obtener un mapa de activación para cada clase de salida, lo que permitió obtener información sobre las regiones del archivo son indicativas de cada una de las

[6] Los mecanismos de atención permiten que el modelo se centre selectivamente en las partes de la entrada que son más importantes para hacer una predicción e ignore las partes menos relevantes. Esto puede ayudar al modelo a hacer predicciones más precisas y funcionar de manera más eficiente.

etiquetas. El modelo *MalConv* demostró una diversidad más amplia de tipos de información utilizados en comparación con el enfoque de n-grama de bytes.

Por último, se destacó que la *batch-normalization* falló en el modelo *MalConv*, lo cual fue inusual ya que esta técnica generalmente mejora la convergencia y generalización en otros dominios. Los autores sugieren que esta falta de efectividad puede deberse a la naturaleza multimodal de los datos binarios, que viola las suposiciones fundamentales de la normalización de lotes, causando un rendimiento degradado.

En [30] se propone y evalúa una arquitectura simple de redes neuronales convolucionales para detectar archivos PE maliciosos a partir del aprendizaje de sus secuencias de bytes y etiquetas, sin la necesidad de extracción de *features* específicas del dominio ni preprocesamiento. Se utilizó un *dataset* de 20 millones de archivos PE desempaquetados de medio megabyte, y esta aproximación de extremo a extremo logró un rendimiento casi equivalente al de los enfoques tradicionales de aprendizaje automático.

La arquitectura consistió en una capa de incrustación seguida de cuatro convoluciones con pasos y una capa de agrupamiento máximo en el medio, seguida de una agrupación promedio global y cuatro capas totalmente conectadas. Se destacó que este enfoque es la contraparte del análisis estático de malware, donde la red solo recibe la secuencia de bytes del ejecutable. Se sugirió que arquitecturas similares podrían tener buenos resultados en el análisis dinámico de malware, donde la red recibiría el código de máquina u otra representación de bajo nivel que desempaquetaría un emulador o un entorno aislado.

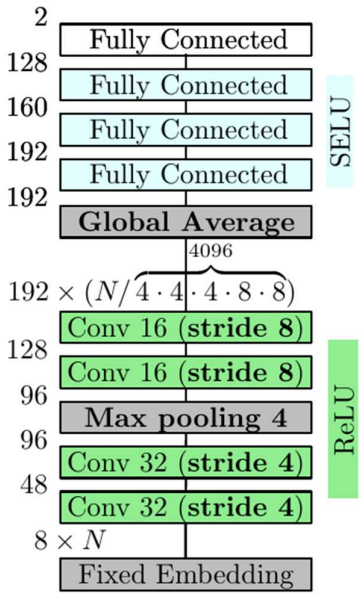

Ilustración 10: *Modelo propuesto*

Además, se mencionó que los archivos PE representan un desafío para el aprendizaje profundo debido a su estructura compleja y la diversidad y dependencia contextual de sus símbolos de bytes. Sin embargo, se considera que el uso de grandes *datasets* en esta área no explorada podría ser de gran interés para la comunidad.

El *dataset* utilizado en el estudio consistió en archivos PE recolectados en un período de 16 meses y se dividió en conjuntos de entrenamiento, validación y prueba. Los archivos fueron etiquetados como "limpio" o "malware" y se buscó lograr bajas tasas de falsos positivos.

La arquitectura de la red neuronal consiste en incrustaciones fijas, convoluciones con pasos para reducir la carga computacional y capas totalmente conectadas. El entrenamiento se realizó utilizando el optimizador *Adam* y se detuvo después del tercer *epoch* cuando los resultados en el conjunto de validación fueron satisfactorios.

El objetivo del estudio fue lograr un bajo índice de falsos positivos, lo que se logró al adaptar la arquitectura para este propósito.

Se compararon las *features* aprendidas por la red convolucional con un conjunto de 538 *features* estáticas diseñadas manualmente utilizadas en el análisis de malware. Los resultados muestran que el aprendizaje a partir de

ejecutables en bruto puede ser un proceso de ingeniería de *features* valioso y complementario a las diseñadas a mano.

En conclusión, aunque el enfoque de aprendizaje de extremo a extremo aún se encuentra ligeramente detrás del enfoque de referencia de aprendizaje automático en este *dataset* específico, se espera que el *deep learning* continúe mejorando en el campo del análisis de malware, especialmente con el uso de arquitecturas y *datasets* más grandes y mejor diseñados. Al mismo tiempo, se señala la posibilidad de incorporar conocimientos del dominio en futuras investigaciones y la importancia de la velocidad de entrenamiento para la aplicabilidad en entornos productivos. Además, se destaca que el *deep learning* podría beneficiar a los expertos en seguridad al observar y localizar patrones previamente desconocidos tanto en archivos benignos como en maliciosos.

En [31] se propone un método de *deep learning* de *features* no lineales para la clasificación de software malicioso basado en su entropía estructural. El enfoque utiliza CNN para aprender representaciones de *features* a partir de series temporales de entropía.

El enfoque se basa en la observación de que las series de tiempo de entropía de una familia de malware son visualmente similares pero distintas a las de otras familias, lo que sugiere que la reutilización de código podría estar presente. Para abordar esto, se utilizaron CNN para aprender representaciones de *features* de las series de tiempo de entropía en los dominios de tiempo y frecuencia.

El proceso de transformación de un archivo ejecutable en una entrada reconocible consta de dos pasos principales:

i) calcular la entropía estructural dividiendo la representación hexadecimal del archivo en fragmentos de tamaño fijo
ii) aplicar la Transformada Discreta de Onda para comprimir la señal y reducir el ruido.

La arquitectura general de la red consiste en tres capas convolucionales seguidas de dos capas completamente conectadas. Las capas convolucionales

realizan el aprendizaje de características en las series de tiempo univariadas, y luego se realiza la clasificación mediante una red de alimentación normal.

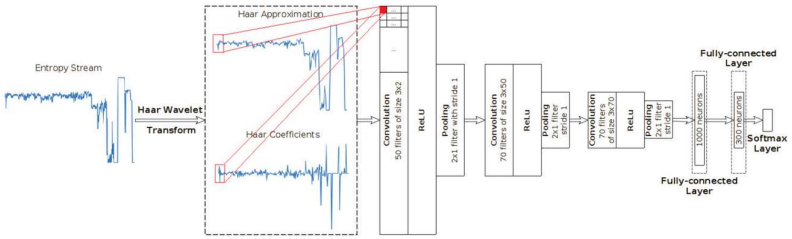

Ilustración 11: *Modelo propuesto*

El enfoque se ha evaluado utilizando un conjunto de datos proporcionado por Microsoft, que incluye 21.741 muestras de malware agrupadas en 9 familias diferentes. Se utilizó validación cruzada con K = 10 para estimar el rendimiento general del método, y se seleccionó aquel modelo con mejor *F1-Score* para evitar la paradoja del *Accuracy*[7].

Se han realizado tres fases de experimentación:

i) se ha estudiado cómo el tamaño de fragmento influye en el rendimiento del enfoque.

ii) se ha analizado cómo la aproximación Haar de la señal entrópica afecta al clasificador

iii) se ha comparado el mejor modelo con otros métodos del estado del arte.

Los resultados demostraron que el enfoque propuesto supera a otros métodos del estado del arte en términos de rendimiento de clasificación. Además, se ha demostrado que las *features* aprendidas por las CNN son invariantes a la traslación, lo que las hace resistentes a las técnicas de ofuscación comunes utilizadas en el desarrollo de malware.

[7] Esta paradoja establece que ciertos modelos predictivos con un nivel dado de *accuracy* pueden tener un mejor desempeño que otros con un *accuracy* mayor. Es por este motivo que métricas como *precision*, *recall* o *F1* son favorecidas por encima del *accuracy*.

3.4 Tráfico de red

El análisis del tráfico de red de una organización complementa el software antivirus descentralizado que se ejecuta en las estaciones de trabajo. Esto permite que las organizaciones apliquen una política de seguridad de manera consistente en toda la red y minimicen la carga de gestión. Además, esta aproximación permite incorporar la detección de malware en dispositivos de red o servicios en la nube. En combinación con soluciones antivirus, el análisis del tráfico de red puede ayudar a detectar malware polimórfico y desconocido basado en patrones de tráfico de red.

El análisis de cargas HTTP puede ser evitado mediante el uso del protocolo HTTPS, en el cual el tráfico es cifrado. Para que el análisis de tráfico siga siendo efectivo, se debe trabajar con el tráfico HTTPS. En la capa de aplicación, HTTPS utiliza el protocolo HTTP, pero todas las comunicaciones están cifradas a través del protocolo TLS o su predecesor, el protocolo SSL.

En [32] se desarrolla un método para detectar malware en computadoras cliente basado en información observable de la comunicación HTTPS. Se utilizó un enfoque de aprendizaje automático, que depende de grandes cantidades de datos de entrenamiento etiquetados. Se recopiló información de entrenamiento y evaluación utilizando un cliente VPN que es capaz de observar las asociaciones entre archivos ejecutables y el tráfico HTTPS en un gran número de computadoras cliente.

Para extraer las *features* del nombre de dominio, se exploraron modelos de lenguaje neural. Se desarrolló un clasificador de detección de malware basado en LSTM y se compara con un método de referencia basado en *Random Forests* (RF).

Se realizó una recopilación de datos en un entorno de aplicación específico para proteger las computadoras cliente de una organización utilizando un servicio de seguridad web en la nube (CWS). El servicio CWS actúa como intermediario entre la red privada de la organización e Internet, aplicando la política de seguridad y detectando malware.

Los resultados mostraron que el modelo basado en LSTM supera al método de referencia basado en RF y puede detectar una proporción significativa de malware, incluso aquellos que no fueron detectados por un antivirus basado

en firmas. Esto proporciona una herramienta valiosa para complementar los enfoques existentes de detección de malware.

En [33] se propone un modelo eficiente de detección de anomalías para proteger entornos de Control de Sistemas Industriales Ciberfísicos (IICS) contra actividades maliciosas. El modelo utiliza técnicas de *deep learning* y consta de dos etapas principales: entrenamiento y prueba.

En la etapa de preprocesamiento de datos, se realizó una transformación y normalización de las *features* para seleccionar información importante de los datos a gran escala en un entorno IIoT. La transformación de *features* convierte los valores simbólicos en numéricos, y la normalización asegura que las *features* tengan la misma escala para evitar sesgos en los pesos del modelo de aprendizaje profundo.

Posteriormente, se utilizó un modelo DAE-DFFNN para reducir la dimensionalidad de los datos de red, lo que permitió diseñar una técnica de detección de anomalías ligera y escalable. La reducción de *features* se realizó sin conocimiento humano, y el objetivo fue encontrar representaciones bien diseñadas y reducidas en términos de *features* aprendidas de alto nivel.

Para detectar actividades sospechosas, se aplicó un proceso de aprendizaje no supervisado para estimar los parámetros de inicialización de los pesos y sesgos. Luego, los parámetros se ajustan en el proceso de aprendizaje supervisado utilizando datos etiquetados (normales y maliciosos). El modelo final se evaluó utilizando nuevos datos durante la fase de prueba.

La colocación de un IDS en el entorno IIoT es crucial para garantizar la seguridad contra actividades maliciosas. El IDS propuesto se implementa en el *gateway* IoT, que monitorea y recopila el tráfico de red y utiliza una base de datos para almacenar datos de tráfico, el comportamiento y los registros. El componente analizador procesa los datos y el modelo de detección clasifica las observaciones como normales o ataques en función del comportamiento aprendido. El componente de respuesta alerta al administrador del sistema cuando se detecta una actividad anómala.

En cuanto a los conjuntos de datos y métricas de evaluación, el estudio utilizó los *datasets* NSL-KDD y UNSW-NB15 para evaluar el rendimiento del

modelo propuesto. Se utilizaron métricas como precisión, tasa de detección y tasa de falsos positivos para evaluar el rendimiento del modelo.

En las pruebas realizadas, el modelo propuesto mostró un rendimiento superior en el conjunto de datos NSL-KDD en comparación con UNSW-NB15. El modelo alcanzó una precisión del 98,4% para NSL-KDD y aproximadamente 92,5% para UNSW-NB15. También demostró buenas tasas de detección para varios tipos de registros en ambos *datasets*.

Además, se realizó un estudio comparativo con otros ocho modelos recientes de detección de anomalías. El modelo que propusieron superó a estos otros modelos en términos de detección de ataques y tasas de falsos positivos.

En conclusión, el modelo propuesto de detección de anomalías basado en *deep learning* mostró un rendimiento prometedor en la detección de actividades maliciosas en entornos IIoT. Su capacidad para reducir la dimensionalidad y extraer *features* automáticamente lo hizo adecuado para aplicaciones en entornos industriales con grandes cantidades de datos no etiquetados y no estructurados. Aunque presentó algunas desventajas en la elección de parámetros, su eficiencia y efectividad general lo hicieron una solución práctica para la detección de actividades intrusivas en entornos IIoT del mundo real.

3.5 GNN

El sistema propuesto para detección de malware en [34] puede extraer directamente información de grafos basados en CFG de archivos ejecutables y luego comprimir la información en un vector de características utilizando GIN (*Graph Isomorphism Network*). En última instancia, el sistema clasifica los vectores de *features* utilizando un MLP.

El CFG es una notación de estructura de programa basada en grafos bien conocida en el campo de la informática. El programa se divide con instrucciones de bifurcación (incluidas las instrucciones de bifurcación para el uso de llamadas a funciones) y los bloques separados entre las instrucciones de bifurcación se llaman bloques básicos. Un típico final del bloque básico está conectado a un bloque básico o a varios bloques básicos. Por lo tanto, se puede usar la notación de conexión de bloques básicos para expresar la estructura del programa; este

grafo de conexión se convierte en el CFG. El CFG es la base de muchas optimizaciones de compiladores y herramientas de análisis estático.

Para el *dataset* utilizado se incluyeron muestras benignas obtenidas del proyecto DikeDataset[8] y las muestras maliciosas se obtuvieron de BODMAS[9]. Se recopilaron un total de 1000 muestras, que incluyeron 500 muestras maliciosas y 500 benignas. Se seleccionaron 8 familias de malware. De cada una de las familias se tomó una pequeña muestra. Debido al tamaño reducido del *dataset*, se utilizó una división especial para evitar que todas las métricas de evaluación se acercaran o igualaran a 1, lo que dificultaría la comparación intuitiva de los resultados experimentales. Se aplicó una validación cruzada de permutaciones aleatorias (*Shuffle & Split*) con 5 iteraciones de mezcla y división. La división resultó en un conjunto de entrenamiento y un conjunto de prueba, cada uno representando el 50% del *dataset* original, a diferencia del enfoque de validación cruzada tradicional que utiliza una proporción fija de entrenamiento y prueba.

La función principal del módulo GFE (*Graph Feature Extraction*) es extraer la información de CFG de las muestras, mantener la información de la estructura y transformarla en un grafo FCG (*Function Call Graph).*

La diferencia entre CFG y FCG es que el CFG contiene el código ensamblador de cada bloque básico, mientras que el FCG lo elimina. En el FCG, solo se almacena información que incluye la dirección de inicio del bloque básico, el nombre de la API y el nombre de la función. Específicamente, el CFG muestra todas las rutas que un programa recorrerá durante el proceso de ejecución. Además, representa gráficamente el flujo posible de todos los bloques básicos ejecutados dentro de un proceso y también refleja los bloques básicos que se ejecutan realmente durante la ejecución. Algunos softwares maliciosos incluyen código muerto (que no se ejecuta en ninguna ruta de ejecución) como ofuscación, pero esta técnica de análisis puede hacerle frente. Debido a que el CFG de una muestra única contiene una gran cantidad de información, el preprocesamiento consume mucho tiempo. Por lo tanto, solo se mantiene la información de la relación estructural, se le asigna un nombre unificado a cada bloque y este nombre es usado para representar el bloque. El código ensamblador en cada

[8] https://github.com/iosifache/DikeDataset
[9] https://github.com/whyisyoung/BODMAS

bloque no se utiliza como *feature*. Cada bloque en el CFG generalmente tiene dos tipos de contenido: la API del sistema y las funciones internas del programa. Cuando un bloque es una API del sistema, su nombre se usa directamente como el nombre del bloque. En cambio, si es una función interna, se usa el *offset* de la función como el nombre del bloque. Para generar el CFG a partir del código binario se utilizó la librería *angr*[10] que combina análisis simbólico estático y dinámico. El grafo generado es un grafo dirigido. Para la manipulación de estos grafos, se utilizó la librería *NetworkX*[11].

Este módulo, además, inserta los nombres de los bloques a través de un modelo de lenguaje preentrenado a gran escala. El modelo de generación utilizado se llama "*all-MiniLM-L12-v2*". Este tiene un conjunto de entrenamiento de mil millones de elementos y está diseñado como un modelo general. El modelo *MiniLM* basado en BERT, lanzado por Microsoft, es un transformador de 12 capas con un tamaño oculto de 384 que contienen alrededor de 33 millones de parámetros. Mapea oraciones y párrafos a un espacio vectorial denso de 384 dimensiones y se puede utilizar para diversas tareas, como el agrupamiento o la búsqueda semántica. Este modelo es el mejor en velocidad de generación y rendimiento entre todos los modelos preentrenados que pueden generar representaciones de 384 dimensiones.

Dado que la dimensión del vector de *features* generado es 384, el vector correspondiente a cada nodo se puede expresar como $X_v\,(384D)$. Para mantener la dimensionalidad de las *features* al mínimo, en lugar de usar la función $READOUT$ original concatenando los resultados de todas las iteraciones/capas, solo se utiliza la salida de la última capa. Las representaciones finales del grafo son la tercera capa y se pueden expresar como h_G. La función $READOUT$ utiliza la sumatoria.

Se diseñó una estructura de 3 capas, donde cada capa es una capa *GINConv*. El módulo GDG genera $h_v^0(384D)$ como la entrada de $GINConv_1$ y la salida es $h_v^1(64D)$. El modelo $GINConv_1$ tiene una dimensión de entrada de 384 y una dimensión de salida de 64. Los resultados de salida de la primera capa deben normalizarse y aplicarse la función de activación ReLU antes de ingresar a la segunda capa. $GINConv$ utiliza el mecanismo de paso de mensajes para

[10] https://angr.io/
[11] https://networkx.org/

actualizar la representación de cada nodo. $GINConv_2$ y $GINConv_3$ son idénticas, con dimensiones de entrada y salida de 64. La dimensión de salida de la función $READOUT$ es 64, por lo que la representación final del grafo correspondiente a cada muestra es un vector de 64 dimensiones. En el estudio se utilizó la configuración predeterminada de GIN. Para escribir y entrenar la red neuronal basada en grafos (GNN) se utilizó la librería *PyTorch Geometric* (*PyG*)[12], basada en *PyTorch*[13]

Por último, para clasificar las distintas muestras se utilizó un MLP. Este tiene una entrada de 64 dimensiones y una salida de 2 dimensiones con un total de tres capas. La capa de entrada y la capa oculta tienen ambas 64 dimensiones, mientras que la capa de salida tiene 2 dimensiones. Además, la salida de la capa oculta se normaliza por lotes y la función de activación es ReLU.

[12] https://www.pyg.org
[13] https://pytorch.org/

Capítulo 4: Sistema de detección de malware basado en GIN propuesto

4.1 Ideas previas y dificultades iniciales

Tomando las ideas propuestas en [34] y complementando con [35] se logró obtener un clasificador binario de archivos ejecutables que corren sobre la plataforma Windows.

Originalmente, la idea iba a ser combinar varias técnicas descriptas en este trabajo tales como: trazas de llamadas a la API, cálculo de entropía del archivo, análisis de *strings* y análisis de N-gramas (tanto de opcodes como de bytes). Todos estos enfoques fueron tratados en este trabajo y en el anterior. Combinando todas esas *features*, el objetivo era armar un vector para luego ser utilizado como entrada para una red neuronal.

Recolectando información se llegó a [34] donde parecía un enfoque interesante. Utilizar la estructura que proveen los CFG, así también como los bloques de código, resultaron de gran interés. En una primera instancia, se intentó replicar el trabajo. Fue una ardua tarea, ya que no se contaba con conocimientos de ninguna de las librerías utilizadas. Así también como la falta de detalles o de algún código fuente para tomar como base.

Finalmente, se utilizó el *dataset* de una competencia en *Kaggle*[14], ya descripto con anterioridad. El CFG en cierta manera ya estaba armado en cada uno de los archivos de texto con instrucciones en ensamblador. Se creó un *parser* para convertir esa entrada en texto en un grafo CFG. En cada nodo se guardó el nombre de la función, nombre de la API si era una función externa, y del *offset* del salto, así también como el listado de las instrucciones en ensamblador. La idea era seguir con el primer enfoque, pero tomando como base el grafo CFG.

Luego, de muchas dificultades, se logró preprocesar una gran cantidad de muestras. El problema era que todas las muestras eran malignas. Como prueba de concepto servía, pero iba a ser insuficiente para generar un sistema de detección binario. Se pensó en adaptarlo a la detección de familias (clasificador

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

multimodal), en lugar de un clasificador binario. Esa idea no prosperó, ya que se apartaba del objetivo inicial del clasificador binario.

Luego, de releer una vez más la información obtenida, se prosiguió a utilizar los *datasets* utilizados en [34]. En particular, se utilizó DikeDataset, ya que contenía tanto muestras benignas como malignas.

Tomada esta decisión, se procedió a experimentar con la librería *angr* para generar el CFG. Una vez logrado este hito, se intentó hacer una prueba de concepto pequeña, de alrededor de 10 grafos, para probar PyG. Luego de varios intentos, la gran mayoría con errores, se logró. No fue posible clasificar dado que para cada uno de los nodos había que indicar dos propiedades: *X* e *y*. La primera hace referencia al vector de *features* (X_v); la segunda la etiqueta.

Como ya había habido suficientes avances, se decidió experimentar con la librería *Sentence-BERT*. Con esto, fue posible lograr el vector de dimensión 384, tal como se explicó en 3.5. Con ese vector ya generado a partir del nombre de la función y la etiqueta ya conocida, se asignaron a las propiedades *X* e *y*. Con todo esto, fue posible imitar a escala muy pequeña (10 grafos) el trabajo propuesto en [34].

4.2 Modelo propuesto

La diferencia con el trabajo original fue la redefinición del modelo GIN. Se utilizó un ejemplo tomado de https://towardsdatascience.com/how-to-design-the-most-powerful-graph-neural-network-3d18b07a6e66.

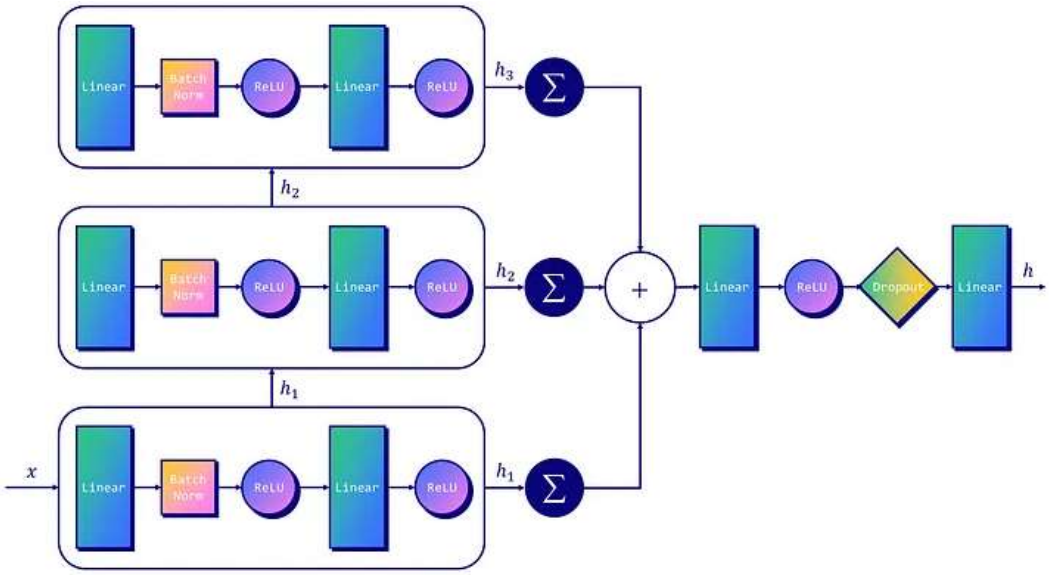

Ilustración 12: *Modelo GIN utilizado.*
Tomado del sitiotowardsdatascience.com

Los vectores ocultos de cada capa son concatenados en lugar de solo considerar el último.

Al igual que en el trabajo de referencia se utilizaron 3 capas ($GINConv_1$, $GINConv_2$ y $GINConv_3$). En lugar de que $GINConv_1$ tenga una salida de dimensión 64 tiene una salida de dimensión 32. Las entradas y salidas de las últimas dos capas se configuraron también en 32. El vector que representa a cada muestra es un vector de dimensión 96. A diferencia de [34] la representación final del grafo es el resultado de la concatenación, de acuerdo con el modelo GIN original.

El parámetro $\mathcal{E}$ de cada una de las capas $GINConv$ se mantuvo con su valor predeterminado (0) y no fue entrenado.

4.3 Preprocesamiento de las muestras

Se tomaron 2300 muestras aleatorias del conjunto *DikeDataset*. La única salvedad que se tuvo fue que únicamente se utilizaron archivos ejecutables, en el *dataset* original existen archivos OLE, y además se mantuvo una proporción del 50% de muestras benignas y malignas (650 muestras en cada categoría). No se tomaron en consideración las familias de malware.

Se siguió con la misma idea que el trabajo original. Se ejecutó un análisis rápido CFG (*CFGFast*) con la librería *angr*. Luego, se procesó cada CFG utilizando un método específicamente diseñado. Este método consiste en iterar sobre todas las funciones que el analizador haya detectado. Cada una de las funciones contiene un conjunto de bloques. Cada bloque representa un nodo del grafo con instrucciones en ensamblador dentro.

En la función en cuestión, se guardó en un diccionario con el objeto dentro. Este objeto contiene información de interés, como el nombre y los bloques asociados. Para cada bloque, se guardó el nombre y se concatenaron en una única cadena de texto todas las instrucciones en ensamblador que contenía. Luego, a diferencia del trabajo original, se creó el vector de 384 dimensiones utilizando el modelo *MiniLM* con esa cadena de instrucciones. El hecho de utilizar este modelo sobre las instrucciones, da una semántica de lo que hace cada uno de los bloques. De esta manera, se evitan los análisis de N-gramas ya que la información está contenida en el vector. Luego, en el nodo que representa a la función, que contiene los bloques con código, se guarda el vector de 384 dimensiones generado a partir del nombre. No discrimina entre función interna y llamada a la API. Por consiguiente, el análisis de las APIs queda implícito en el vector. Otro punto de interés es que el análisis CFG es resistente al código muerto (*dead code*) o la ofuscación, lo cual, permitiría hacerle frente a esta problemática.

Es importante remarcar que para que el modelo funcione, todos los nodos han de tener la misma dimensión del vector de *features* ($X_v(384D)$). Para la etiqueta de cada muestra (y_v), se indicó 0 para software benigno y 1 para malware.

Luego, una vez obtenido el grafo, se convirtió a un tensor de *PyG* para finalmente guardar el archivo preprocesado para su posterior uso.

Aquellas muestras que tardaron más de 20 minutos en fueron descartadas.

4.4 Resultados

Para entrenar el modelo se utilizó el 80% de las muestras. Para la validación cruzada el 10% y para evaluar la precisión el 10% restante.

Los *epochs* se configuraron en 100

Se intentó utilizar CUDA para agilizar el entrenamiento, pero por limitaciones de hardware, no se pudo continuar. Se utilizó la CPU para el entrenamiento, validación y prueba del modelo propuesto.

Con las 2300 muestras se logró un *accuracy* de 97,40, un *recall* de 98,92 y un *F1-Score* de 78,23.

A priori, no se podría decir que mejoró el modelo original, pues como no se sabe qué muestras utilizaron en el original, no se puede comparar. Si se hubieran tenido las mismas muestras utilizadas para evaluar el modelo original, hubiese sido posible ejecutar el modelo propuesto y obtener las métricas que permitieran su comparación.

Conclusiones y trabajo futuro

La solución propuesta demostró tener buenos resultados. Dada la limitación del hardware sobre el que se ejecutaron las pruebas, quedan pendientes pruebas de entrenamiento con mayor cantidad de muestras. El número elegido no es lo suficientemente grande. Asimismo, habría que probar cuán efectivo es para detectar *zero-days* y eventualmente, cambiar la clasificación binaria a una con varias etiquetas para ser capaz de detectar también las familias.

Bibliografía

[1] Juang y Rabiner, «Hidden Markov Models for Speech Recognition,» *American Statistical Associtation and the American Society for Quality Control*, pp. 251-272, 1991.

[2] D. Carlin, A. Cowan, P. O'Kane y S. Sezer, «The Effects of Traditional Anti-Virus Labels on Malware Detection Using Dynamic Runtime Opcodes,» *IEEE Access*, vol. V, pp. 17742-17752, 2017.

[3] Y. Ding, X. Yuan, K. Tang, X. Xiao y Y. Zhang, «A fast malware detection algorithm based on objective-oriented association mining,» *Computers & Security*, vol. 39, pp. 315-324, 2013.

[4] Y.-D. Shen, Q. Yang y Z. Zhang, «Objective-Oriented Utility-Based Association Mining,» de *IEEE International*, 2002.

[5] P. O'Kane, S. Sezer y K. McLaughlin, «Obfuscation: The Hidden Malware,» *IEEE Security & Privacy*, pp. 41-47, 2011.

[6] C. Ranjan, Understanding Deep Learning, 2020.

[7] R. Parmar, «medium.com,» medium.com, 02 09 2018. [En línea]. Available: https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23. [Último acceso: 11 09 2023].

[8] M. Bronstein, «towardsdatascience.com,» 26 6 2020. [En línea]. Available: https://towardsdatascience.com/expressive-power-of-graph-neural-networks-and-the-weisefeiler-lehman-test-b883db3c7c49. [Último acceso: 2 9 2023].

[9] [En línea]. Available: https://colab.research.google.com/github/VisiumCH/AMLD-2021-Graphs/blob/master/notebooks/workshop_notebook.ipynb. [Último acceso: 2 9 2023].

[10] W. Hu, B. Lui, J. Gomes, M. Zitnik, P. Liang, V. Pande y J. Leskovec, «Strategies for Pre-Training Graph Neural Networks,» de *ICLR 2020*, 2020.

[11] R. Shaikh, «medium.com,» medium.com, 26 08 2023. [En línea]. Available: https://medium.com/@shaikhrayyan123/a-comprehensive-guide-to-understanding-bert-from-beginners-to-advanced-2379699e2b51. [Último acceso: 11 09 2023].

[12] D. Gibert, C. Mateu y J. Planes, «The rise of machine learning for detection and classification of malware: Research developments, trends and challenges,» 2019.

[13] M. Ghiasi, A. Sami y Z. Salehi, «Dynamic Malware Detection Using Registers Values Set Analysis,» de *Information Security and Cryptology (ISCISC)*, 2012.

[14] M. Ghiasi, A. Sami y Z. Salehi, «Dynamic VSA: a framework for malware detection based on register contents,» de *Engineering Applications of Artificial Intelligence*, 2015.

[15] B. Anderson, D. Quist, J. Neil, C. Storlie and T. Lane, "Graph-based malware detection using dynamic analysis," *Computer Virology*, pp. 247-258, 2011.

[16] D. Berkerman, B. Shapira, L. Rokach y A. Bar, «Unknown Malware Detection Using Network Traffic Classification,» *IEEE Conference on Communications and Network Security*, pp. 134-142, 2015.

[17] G. Zhao, K. Xu, L. Xu y B. Wu, «Detecting APT Malware Infections Based on Malicious DNS and Traffic Analysis,» *IEEE*, vol. 3, pp. 1132-1142, 2015.

[18] N. Kheir, «Behavioral classification and detection of malware through HTTP user agent anomalies,» *Journal of Information Security and Applications,* vol. 18, nº 1, pp. 2-13, July 2013.

[19] A. Boukhtouta, S. Mokhov y M. Debbabi, «Network Malware Classification Comparison Using DPI and Flow Packet Headers,» *Journal of Computer Virology and Hacking Techniques,* pp. 1-32, 2015.

[20] K. Rieck, P. Trinius y T. Holz, «Automatic analysis of malware behavior using machine learning,» *Journal of Computer Security,* pp. 639-668, 2011.

[21] H. Galal, «Behaviour-based features model for malware detection,» *Journal of Computer Virology and HackingTechniques,* 2015.

[22] D. Uppal, V. Mehra y V. Jain, «Malware Detection and Classification Based on Extraction of API Sequences,» *International Conference on Advances in Computing, Communications and Informatics (ICACCI),* pp. 2337-2342, 2014.

[23] Z. Salehi, A. Sami y M. Ghiasi, «MAAR: Robust features to detect malicious activity based on API calls, their arguments and return values,» *Engineering Applications of Artificial Intelligence,* vol. 59, pp. 93-102, 2017.

[24] «Large-Scale Malware Classification using Random Projections and Neural Networks,» de *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, Vancouver, BC, Canada, 2013.

[25] W. Huang y J. Stokes, «MtNET: A Multi-Task Neural Network for Dynamic Malware Classification,» *DIMVA 2016: Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 9721, p. 399–418, 2016.

[26] J. Saxe y K. Berlin, «Deep Neural Network Based Malware Detection Using Two Dimensional Binary Program Features,» de *10th International Conference on Malicious and Unwanted Software (MALWARE)*, 2015.

[27] B. Kolosnjaji, G. Eraisha, G. Webster y A. Zarras, «Empowering convolutional networks for malware classification and analysis,» de *International Joint Conference on Neural Networks (IJCNN)*, 2017.

[28] B. Athiwaratkun y J. Stokes, «CNN, Malware classification with LSTM and GRU language models and a character-level,» de *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017.

[29] E. Raff, J. Barker y J. Sylvester, «Malware Detection by Eating a Whole EXE,» 2017.

[30] M. Krčál, O. Švec, O. Jašek y M. Bálek, «Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only,» de *ICLR 2018*, 2018.

[31] D. Gibert, C. Mateu y J. Planes, «Classification of Malware by Using Structural Entropy on Convolutional Neural Networks,» de *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[32] P. Prasse, L. Machlica y T. Pevny, «Malware Detection by Analysing Network Traffic with Neural Networks,» de *IEEE Security and Privacy Workshops (SPW)*, 2017.

[33] M. Al-Hawawreh, N. Moustafa y E. Sitnikova, «Identification of malicious activities in industrial internet of things based on deep learning models,» *Journal of Information Security and Applications,* nº 41, 2018.

[34] Y. Gao, H. Hasegawa, Y. Yamaguchi y H. Shimada, «Malware Detection by Control-Flow Graph Level Representation Learning With Graph Isomorphism Network,» *IEEE Access 10,* vol. 10, pp. 111830-111841, 2022.

[35] Y. Guo, P. Li, Y. Luo, X. Wang y Z. Wang, «Exploring GNN Based Program Embedding Technologies for Binary Related Tasks,» de *30th IEEE/ACM International Conference on Program Comprehension*.