UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería Informática

Señales Electromagnéticas para la detección y clasificación de malware

Electromagnetic signals for malware classification and detection

Realizado por
Adrián Meis Asensi

Tutorizado por
Antonio Muñoz Gallego

Departamento
Lenguajes y Ciencias de la Computación

MÁLAGA, septiembre de 2023

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADO EN INGENIERÍA INFORMÁTICA

Señales Electromagnéticas para la detección y clasificación de malware

Electromagnetic signals for malware classification and detection

Realizado por
Adrián Meis Asensi

Tutorizado por
Antonio Muñoz Gallego

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, SEPTIEMBRE DE 2023

Fecha defensa: septiembre de 2023

Abstract

With this Bachelor's Final Project we checked the validity of a new method, by analysing the electromagnetic signals emitted by the processor (CPU) of IoT (Internet of Things) devices, to discover and classify possible malware threats that could be affecting them, while at the same time doing it in an undetectable and non-invasive manner, trying to not consume the valuable and limited resources available to those devices. We have used an oscilloscope to obtain more than 30GB of traces from different types, families and variants of malware that are commonly found in recent attacks against these types of devices and we have trained and validated two types of networks, machine and neural, while also using different methods of classification for each of them. In the obtained results we have found an accuracy of over 90 % in the detection and classification of types of malware, confirming that this method of malware detection can be a very effective defensive tool against cyberattacks while also being able to be expanded upon in future research.

Keywords: **side-channel attacks, malware detection, deep learning**

Resumen

Con este trabajo de fin de grado se trata de comprobar la validez de un método, mediante el análisis de las señales electromagnéticas emitidas por los procesadores (CPU) de dispositivos IoT (Internet of Things), para descubrir amenazas malware que puedan estar afectando a dichos dispositivos, clasificarlas y a su vez realizarlo de una manera indetectable y no invasiva, tratando de no consumir los valiosos y limitados recursos de los que disponen. Se han capturado con un osciloscopio más de 30GB de trazas de diversos tipos, familias y variantes de malware comúnmente utilizados en la actualidad en ataques masivos contra dispositivos IoT y se ha entrenado y validado con ellas dos tipos de redes, de machine learning y neuronal, además de usar para cada una de estas varios métodos. En los resultados obtenidos se ha podido comprobar una exactitud superior al 90 % en la detección y clasificación de tipos de malware, confirmando así que este método de detección puede ser altamente efectivo como medida de protección contra ciberataques y que puede ser expandido para obtener resultados aún mejores en futuras investigaciones.

Palabras clave: ataques de canal lateral, detección de malware, deep learning

Índice

1 Introducción

1.1. Motivación

Vivimos en un mundo con cada vez más dispositivos inteligentes, pasando de 3 mil millones en 2015 a una proyección de cerca de 17 mil millones para finales de 2023 [19], lo que equivale aproximadamente a 2 objetos inteligentes por cada ser humano en la Tierra. Además, estos cuentan cada vez con mayores capacidades de procesamiento y algunos de ellos ejecutan sistemas operativos (SO) completamente funcionales con procesadores multinúcleo, lo que aumenta la probabilidad de que estos sean atacados por malware.

Los sistemas de análisis que se basan en características estáticas y dinámicas todavía presentan varias desventajas para los analistas de malware. En particular, las características estáticas pueden ser fácilmente manipuladas mediante técnicas de empaquetado u ofuscación, mientras que el monitoreo basado en software dinámico puede ser detectable (por ejemplo, mediante huellas digitales de entornos controlados), llevando a la finalización de la ejecución del malware y, por lo tanto, dificultar la posibilidad de realizar análisis de comportamiento. Además, a diferencia de los sistemas informáticos y servidores, los sistemas ciberfísicos integrados pueden no tener suficientes recursos o la accesibilidad necesaria para asignar a soluciones de análisis de malware. Todos estos factores dificultan que los analistas de malware obtengan automáticamente información adecuada sobre las muestras de malware recopiladas (por ejemplo, naturaleza, evolución, etc.) para poder mitigar los riesgos de seguridad.

En esta investigación y en la que nos basamos [22] nos centramos en el campo electromagnético (EM) de un dispositivo integrado como fuente para el análisis de malware, lo cual ofrece varias ventajas. De hecho, la emanación EM que se mide desde el dispositivo es prácticamente indetectable por el malware. Por lo tanto, las técnicas de evasión de malware no se pueden aplicar directamente, a diferencia del monitoreo de software dinámico. Además, dado que el

malware no tiene control sobre eventos externos a nivel de hardware (por ejemplo, emanación EM, disipación de calor), no se puede desactivar un sistema de protección basado en características de hardware, incluso si el malware tiene los máximos privilegios en la máquina. Por lo tanto, con la emanación EM, es posible detectar malware sigiloso (por ejemplo, rootkits a nivel de kernel), que pueden evitar los métodos de análisis basados en software. Otra ventaja es que el monitoreo de la emanación EM no requiere alterar el dispositivo para analizarlo. En otras palabras, no depende de la arquitectura y el sistema operativo del dispositivo, ni implica ninguna carga computacional adicional.

Con este trabajo se pretende expandir sobre investigaciones anteriores [22] mediante el análisis de la capacidad de estos métodos para detectar distintos tipos de malware no explorados previamente.

1.2. Objetivos

Este trabajo tiene como principal objetivo el estudio de la capacidad de las técnicas de análisis de señales electromagnéticas para la detección y clasificación de malware avanzado en dispositivos IoT (Internet of things). En particular, se centra en técnicas para la detección de malware con la capacidad de inhabilitarse al detectar su monitorización mediante el análisis de la actividad electromagnética en los componentes hardware. Con los ensayos realizados sobre un dispositivo RaspBerry Pi en los que se han ejecutado diversos tipos de malware, con o sin ofuscación, se han obtenido las lecturas de las señales electromagnéticas emitidas por dicho dispositivo mediante un osciloscopio que a su vez estará conectado al ordenador de control. Con dichas lecturas y con la ayuda de diversas tecnologías de Deep Learning y de Machine Learning se pretende comprobar la capacidad de una red neuronal para detectar si un malware está presente en un dispositivo a través de las señales electromagnéticas del mismo.

También existen una serie de objetivos que se podrían considerar de carácter secundario, como son:

- **Comprobar que se puede replicar todo lo realizado en la investigación anterior** [22]. Con este objetivo se pretende verificar que es posible seguir todos los pasos indicados en dicha investigación y obtener resultados similares.

- **Documentar todos los procedimientos realizados**. Es de vital importancia dejar

constancia de todos los pasos y procesos que se realicen a lo largo de este trabajo.

- **Verificar cuáles de las redes y sus clasificadores/arquitecuras son las más eficientes**. Al emplear dos tipos de redes (*machine* o *deep learning*) con varios métodos para cada una se intenta verificar cuál es la combinación red/método que obtiene la mejor proporción entre exactitud y empleo de recursos de computación.

- **Expandir el repertorio de muestras de malware**. Mediante el empleo de muestras de nuevas variantes y familias, se aspira a obtener una capacidad de detección de malware más avanzada y completa.

1.3. Estructura del documento

Este documento se dividirá principalmente en tres partes:

- Una primera sección detallando la metodología empleada, así como el proceso realizado de manera resumida.

- Una segunda sección donde se explican los fundamentos tecnológicos en los que se basa este trabajo, como las redes (*machine* y *deep learning*) empleadas para la detección del malware y las investigaciones anteriores relacionadas con este mismo método de análisis.

- Una tercera sección para discutir los resultados obtenidos, explicar el por qué de los mismos y qué se puede obtener a partir de ellos.

- Una cuarta sección donde se numeran y detallan los distintos tipos de malware y de ofuscación que se han utilizado en las pruebas y qué se pretende detectar.

- Una quinta sección en la que se comentan los distintos pasos realizados durante todo el proceso de captura de las muestras, entrenamiento y validación de la red neuronal y obtención de los resultados.

- Una última sección con las conclusiones generales de este trabajo y de la investigación realizada.

También se incluyen en el apéndice el código de scripts generados para la investigación, un manual de instalación y manuales de entrenamiento y validación de las redes de *machine* y *deep learning*.

1.4. Tecnologías usadas

En el proceso de captura de las muestras se ha utilizado un osciloscopio *RIGOL MSO5074*, encargado de capturar las señales electromagnéticas emitidas por el procesador del dispositivo y el cual posee 4 canales analógicos para captura, un ancho de banda de 70MHz y una frecuencia de muestreo de hasta 8GS (Giga Samples) por segundo además de la capacidad de conexión con USB la cual es vital para el envío de los comandos de captura desde el ordenador al que se conecte. En adición a este se emplea una sonda, una Raspberry Pi 3B+ utilizando un sistema operativo *Raspberry OS* con una versión de **Linux** 4.19.57-v7+ a la que se le han mandado comandos de *shell* mediante **SSH** y también se han empleado diversos scripts escritos en el lenguaje de programación **Python**. Para el proceso de análisis de las muestras se han utilizado modelos tanto de *Machine Learning*, siendo los clasificadores **Naive Bayes** (NB) y **Support Vector Machine** (SVM), como de *Deep Learning* utilizando las arquitecturas **Multilayer Perceptron** (MLP) y **Convolutional Neural Network** (CNN).

También se ha utilizado el leguaje TEX en combinación con el sistema de preparación de documentos LATEX para realizar el trabajo de redacción de esta memoria.

2

Metodología

La metodología utilizada en esta investigación se basa en el método científico, con el objetivo de desarrollar una herramienta de software capaz de identificar y clasificar software malicioso en dispositivos IoT. La hipótesis planteada es que mediante la captura de datos electromagnéticos del procesador con un osciloscopio, es posible detectar y clasificar diferentes tipos de malware utilizando una red neuronal y una red de *machine learning*.

El enfoque principal de esta metodología es el análisis de patrones anómalos en las señales electromagnéticas emitidas por el procesador de un dispositivo. Para llevar a cabo esta tarea, se ha utilizado una Raspberry Pi como elemento hardware y se han capturado las medidas de las señales electromagnéticas utilizando una sonda unida a un osciloscopio *RIGOL MSO 5074*. Estas medidas se han primero procesado y posteriormente empleado como datos de entrada para el entrenamiento y validación de la red neuronal y de la red de *machine learning*.

El proceso consiste en realizar pruebas lanzando mediante comandos SSH varios tipos de malware (ransomware, rootkits, botnets y malware ofuscado) sobre la RPi, y a su vez registrar las señales electromagnéticas generadas en el procesador por cada uno de ellos con una sonda conectada a un osciloscopio. Estas señales, tras ser guardadas en la máquina que actúa como dispositivo de mando (en nuestro caso un ordenador portátil), posteriormente se procesan empleando scripts de **Python**, realizando un espectrograma de las mismas que permite descubrir y extraer de ellas las secciones de banda ancha donde hay anomalías, reduciendo así el tamaño de las trazas y eliminando el ruido generado de manera normal por la CPU. Después, estas son divididas en diversos grupos para el entrenamiento y posterior validación de las redes, también empleando scripts del lenguaje **Python** y en el caso de la red neuronal la librería *tensorflow*, mientras que en la red de *machine learning* estás son sometidas a un procesado adicional para ajustarlas a las necesidades de dicho tipo de red. Siendo empleadas las trazas procesadas como datos de entrada, se entrenan ambas redes y sus respectivos métodos,

permitiéndolas aprender y reconocer los patrones característicos presentes en las señales de cada tipo de malware. Con las redes ya entrenadas, se procede a validar la efectividad de las mismas frente a muestras desconocidas para ellas. El objetivo principal de esto siendo evaluar la capacidad de cada red para detectar y clasificar correctamente según tipo, familia y si está ofuscado o no el malware desconocido y además comparándolas entre ellas para encontrar la combinación de red/arquitectura más eficiente.

Los resultados obtenidos con este este trabajo permiten determinar si la redes desarrolladas pueden ser utilizadas en entornos de peligros reales para obtener información confiable sobre la presencia de malware, tanto ofuscado como sin ofuscar, en dispositivos IoT, así como identificar el tipo de malware que está afectando a dicho dispositivo y hacerlo de la manera más eficiente posible, manteniendo unos niveles de precisión aceptables y sin ningún tipo de efecto negativo sobre el funcionamiento de dichos dispositivos.

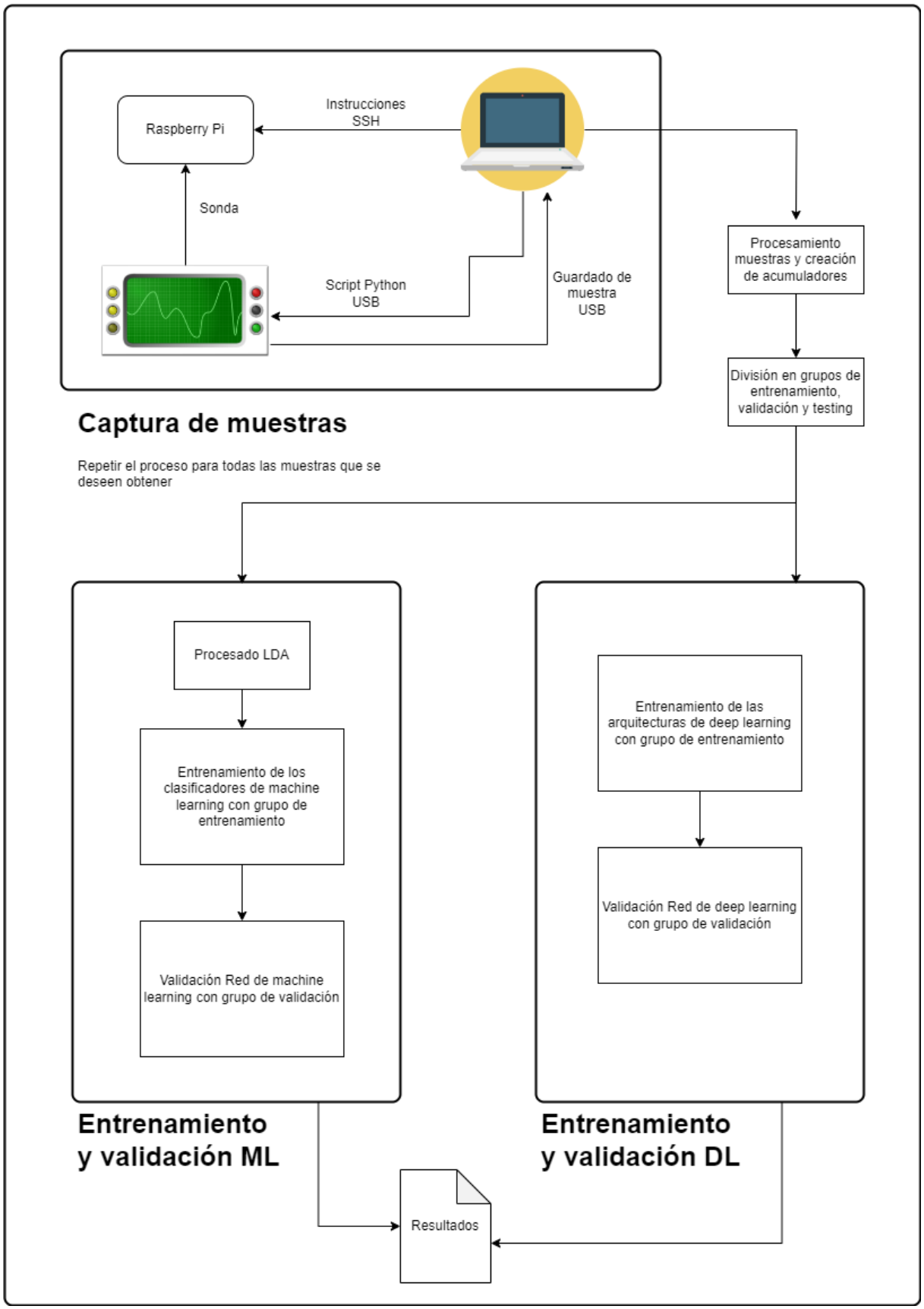

Figura 1: Diagrama a alto nivel del proceso realizado

3

Fundamentos tecnológicos

Para esta investigación se utilizaran dos tipos de red distintos, con dos clasificadores/arquitecturas en cada uno, comprobando así cuál es la combinación que obtiene un mejor equilibrio entre coste computacional y precisión de los resultados.

3.1. Red de machine learning

El propósito principal del modelo de machine learning es aprender de los datos sin ser programado explícitamente. En lugar de seguir instrucciones específicas, las máquinas utilizan algoritmos y datos para aprender y mejorar su rendimiento en tareas específicas. Cuando se refiere a grandes cantidades de datos que son difíciles de interpretar manualmente, el machine learning se aplica para conseguir que las máquinas extraigan y manejen la información relevante de los datos de manera más eficiente.

El machine learning se basa en diversos algoritmos para resolver problemas de datos, lo que que implica que no existe un algoritmo único que sea la mejor solución para todos los problemas; la elección del algoritmo depende del tipo de problema a resolver, el número de variables involucradas y el tipo de modelo que se adapte mejor a la situación. En nuestro caso utilizaremos algoritmos de aprendizaje supervisado, que consiste en aprender una función que asigna una entrada a una salida en función de ejemplos de pares de entrada-salida. En otras palabras, se proporciona a la máquina un conjunto de datos de entrenamiento etiquetado, que contiene ejemplos de entradas junto con sus respectivas salidas deseadas. El conjunto de datos de entrada se divide en conjuntos de entrenamiento y prueba, mientras que el conjunto de entrenamiento contiene las variables de salida que deben predecirse o clasificarse [17].

En el contexto de esta investigación, este tipo de red es una opción menos costosa en térmi-

nos de recursos y de tiempo empleado en el análisis en comparación con las redes neuronales. Aunque con estas ventajas también conlleva que los resultados obtenidos sean, normalmente, menos precisos que aquellos resultantes de análisis con redes neuronales. Dentro de esta red también se comprueban dos algoritmos distintos con el mismo propósito de descubrir cuál de ellos es el más eficiente.

3.1.1. Naive Bayes (NB)

Naive Bayes es un algoritmo clasificador simple que utiliza la Regla de Bayes junto con la suposición de que los atributos son independientes entre sí dado la clase. Aunque esta suposición de independencia a menudo se viola en la práctica, Naive Bayes sigue entregando con frecuencia una precisión de clasificación competitiva. El objetivo de Naive Bayes es estimar la probabilidad posterior P(y | x) de cada clase y dado un objeto x utilizando la información de los datos de muestra. Una vez que se tienen estas estimaciones, se pueden utilizar para clasificación u otras aplicaciones de soporte de decisiones.

Las características de Naive Bayes incluyen:

- Eficiencia computacional: el tiempo de entrenamiento es lineal con respecto al número de ejemplos de entrenamiento.
- Baja varianza: debido a que Naive Bayes no utiliza búsquedas complicadas, tiene baja varianza, aunque esto se logra a costa de un sesgo alto.
- Aprendizaje incremental: Naive Bayes opera a partir de estimaciones de probabilidades de bajo orden que se derivan de los datos de entrenamiento. Estas estimaciones pueden actualizarse fácilmente a medida que se adquieren nuevos datos de entrenamiento.
- Robustez frente al ruido: Naive Bayes siempre utiliza todos los atributos para todas las predicciones y, por lo tanto, es relativamente insensible al ruido en los ejemplos a clasificar. Debido a que utiliza probabilidades, también es relativamente insensible al ruido en los datos de entrenamiento.
- Robustez frente a valores faltantes: debido a que Naive Bayes siempre utiliza todos los atributos para todas las predicciones, si falta un valor de atributo, se sigue utilizando la

información de otros atributos, lo que resulta en un degradado suave en el rendimiento [28].

3.1.2. Support Vector Machine (SVM)

Support Vector Machine (SVM) es un algoritmo clasificador desarrollado por Cortes y Vapnik en 1995 para clasificación binaria. En comparación con Naive Bayes este es un tanto más complejo por lo que puede requerir más recursos y a la vez devolver resultados más precisos.

Su funcionamiento se puede resumir de la siguiente manera:

- Separación de clases: SVM busca el hiperplano de separación óptimo entre dos clases, maximizando así el margen entre los puntos más cercanos de las clases. Los puntos que se encuentran en los bordes del margen se llaman vectores de soporte, siendo el hiperplano que pasa por el centro del margen el hiperplano de separación óptimo.

- No linealidad: cuando no es posible encontrar un separador lineal, los puntos de datos se proyectan en un espacio (generalmente de mayor dimensión) donde se pueden volver linealmente separables mediante técnicas de kernel.

- Solución al problema: toda la tarea se puede formular como un problema de optimización cuadrática que se puede resolver mediante técnicas conocidas [18].

3.2. Red neuronal (Deep learning)

Deep learning es normalmente como se le llama a una clase de redes neuronales que típicamente consisten en más de una capa oculta, organizadas en arquitecturas de redes profundamente anidadas. Estas redes suelen contener neuronas avanzadas que pueden utilizar operaciones complejas (como convoluciones) o múltiples activaciones en una neurona, en contraste con las redes neuronales artificiales (ANN) simples, que solo pueden emplear una función de activación simple. Estas características permiten que las redes neuronales profundas se alimenten con datos de entrada sin procesar y descubran automáticamente representaciones necesarias para la tarea de aprendizaje correspondiente.

El deep learning es especialmente útil en dominios con datos grandes y de alta dimensionalidad, lo que explica por qué las redes neuronales profundas superan a los algoritmos

de machine learning poco profundos en la mayoría de las aplicaciones que involucran procesamiento de texto, imágenes, videos y audio. Sin embargo, para datos de entrada de baja dimensionalidad, especialmente cuando hay una disponibilidad limitada de datos de entrenamiento, el aprendizaje superficial (machine learning) aún puede producir resultados superiores y, a menudo, más interpretables [12].

Esta sería la opción más costosa pero a su vez la que otorga los resultados más precisos. Dentro de esta red también se comprobarán dos arquitecturas diferentas con la intención de descubrir la mejor en términos de empleo de recursos y calidad de los resultados.

3.2.1. Convolutional neural network (CNN)

La Convolutional Neural Network (CNN) o Red Neuronal Convolucional es una de las arquitecturas más populares utilizadas en tiempos recientes. Su función principal es codificar de manera adaptativa la composición de características de datos poco reconocidos en características más significativas, combinando el proceso de extracción de características con los procesos de calibración.

Una típica arquitectura de CNN consta de varias capas, incluyendo la capa de entrada, la capa de convolución, la capa de pooling (submuestreo) y la capa completamente conectada. Generalmente, el formato de los datos de entrada procesados por la red es una imagen plana (datos bidimensionales) o una imagen a color (datos de alta dimensión). Los modelos se entrenan con varios filtros para producir múltiples mapas de características apilados en la capa de convolución, lo que resulta en mapas de características de alta dimensión. Estos mapas de características necesitan reducir su dimensión en la capa de pooling. Luego, las características se convierten en un vector unidimensional para el modelado en la capa completamente conectada.

Esta arquitectura ha sido exitosamente aplicada en diversos campos de análisis de imágenes, incluyendo reconocimiento facial, procesamiento de lenguaje o predicción de edad [7].

En cuanto a la estructura empleada en esta investigación, esta se compone de un *stack* de tres bloques atómicos en donde cada uno de ellos contiene una o dos capas de convolución seguidas de una capa *Pooling* del tipo *Max-Pooling*.

3.2.2. Multilayer perceptron (MLP)

El Multilayer perceptron (MLP) es una arquitectura consistente en un sistema de neuronas interconectadas que representa una asignación no lineal entre un vector de entrada y un vector de salida. Las neuronas están conectadas por pesos y señales de salida, que son una función de la suma de las entradas a la neurona modificada por una función de transferencia no lineal y simple, también llamada función de activación. El MLP puede aproximar funciones extremadamente no lineales gracias a la superposición de muchas de estas funciones de transferencia no lineales y simples.

La estructura del MLP es variable, pero generalmente consta de varias capas de neuronas. La capa de entrada no tiene un papel computacional, sino que simplemente pasa el vector de entrada a la red. El MLP puede tener una o más capas ocultas (así son llamadas las capas que no tienen conexión con el exterior) y, finalmente, una capa de salida. A esta arquitectura se la clasifica como alimentación directa porque la información se procesa en una sola dirección, desde la capa de entrada a través de las capas ocultas hasta la capa de salida. Además, el MLP se define como totalmente conectado, lo que significa que cada neurona está conectada a todas las neuronas de la capa siguiente y la capa anterior [10].

En nuestro caso, el diseño de MLP empleado es el más simple de todos, consta solo de la capa de entrada y de la de salida.

3.3. Trabajos relacionados

Se han realizado diversas investigaciones anteriores sobre este método de detección de malware mediante señales emitidas por un dispositivo (también llamado análisis de canal lateral):

- Una de las primeras investigaciones realizadas sobre este tema fue presentada en la *USENIX Workshop* de 2013. En esta se empleaba la consumición de electricidad de dispositivos médicos para detectar si estos estaban infectados con malware [8].

- En otra investigación, esta vez presentada durante el evento *SPIE Defense + Security* en 2018, también se emplea el consumo eléctrico de dispositivos como Arduino, Raspberry Pi y Siemens PLC. Estas lecturas son después utilizadas como datos de entrada para

entrenar y validar una arquitectura de deep learning MLP (Multilayer Perceptron) que posteriormente se encarga de detectar la presencia de malware en dichos dispositivos. Además, como arquitectura adicional, también se emplea un tipo de red neuronal recurrente llamada LSTM (Long Short-Term Memory) para así comparar la eficacia de ambas [27].

- En la conferencia *ASIA CCS '20* se presentó un estudio en donde se analizaba, como en los demás explicados anteriormente, las señales eléctricas referentes al consumo de un dispositivo para así detectar posibles infecciones malware. En este también se empleaba la red neuronal LSTM para detectar las anomálias en las señales. Los dispositivos empleados eran todas cámaras de seguridad con conexión a internet, como por ejemplo la cámara inalámbrica *D-Link DCS-934L* [9].

- También existen investigacion que emplean señales electromagnéticas para la detección de malware. En un artículo publicado en *IEEE Transactions on Dependable and Secure Computing* se presenta un método de análisis de amenazas en dispositivos monitorizando de manera constante las señales electromagnéticas emitidas por estos y al detectar algún tipo de anomalía (una fluctiación que se encuentre fuera de los patrones observados normalmente) se avisa de esta como posible actividad maliciosa [15].

Todas estas investigaciones presentan problemas, algunas no utilizan muestras reales de malware si no que imitan el comportamiento de estas. Las hay que dependen de fluctuaciones que pueden resultar en falsos positivos y otras se han llevado a cabo en entornos muy controlados y alejados de casos reales.

En el trabajo en el que nos basamos [22] se corrigen muchos de estos problemas, se emplean muestras de malware auténticas, en un entorno similar a la realidad y además haciendo uso de redes de diversos tipos para conseguir detectar los patrones de estos actores maliciosos.

4

Resultados

Hay varias categorías en las que se ha medido la precisión de las redes, también conocidas como *tagmaps*. Estas son *executable classification* que se refiere al plano más bajo, es decir, detectar la muestra de malware específica, *family classification* en la que se intenta detectar la familia a la que pertenece el malware, *novelty classification* donde se quiere conocer el tipo del malware, *obfuscation classification* para detectar si el malware está utilizando alguna técnica de ofuscación y seleccionar cúal de ellas, *type classification* que se refiere al tipo del malware a un nivel más alto (ransomware, ddos, benigno o rootkit).

4.1. Resultados de machine learning

	LDA + NB			
Categoría	#BD	Exactitud	Precisión (macro/peso)	Exhaustividad (macro/peso)
Executable	24	0,6302	0,6430/0,6470	0,6291/0,6302
Family	24	0,953	0,8455/0,9541	0,8473/0,9530
Novelty	24	0,9589	0,9471/0,9618	0,9692/0,9589
Obfuscation	22	0,569	0,5562/0,5702	0,5552/0,5690
Type	24	0,9741	0,9812/0,9743	0,9837/0,9741

Cuadro 1: Tabla con los resultados del clasificador *Naive Bayes*. Se han escogido solo los valores de banda ancha (bd) con la exactitud (accuracy) más alta. También se muestran la precisión y exhaustividad (recall)

4.1.1. Naive Bayes

La exactitud conseguida por este clasificador para la mayoría de las categorías es superior a 0.9, a excepción de la categoría de *executable* en donde ha una exactitud de 0.63 ya que esta

	LDA + SVM			
Categoría	#BD	Exactitud	Precisión (macro/peso)	Exhaustividad (macro/peso)
Executable	24	0,6282	0,6351/0,6401	0,6263/0,6282
Family	24	0,9571	0,8474/0,9575	0,8466/0,9571
Novelty	24	0,9738	0,9721/0,9754	0,9805/0,9738
Obfuscation	24	0,5626	0,5533/0,5697	0,5497/0,5626
Type	24	0,981	0,9882/0,981	0,9874/0,9810

Cuadro 2: Tabla con los resultados del clasificador *Support Vector Machine*. Se han escogido solo los valores de banda ancha (bd) con la exactitud (accuracy) más alta. También se muestran la precisión y exhaustividad (recall)

es la de más bajo nivel y en donde se debe detectar la variante exacta de malware. También se puede observar que la ofuscación es la categoría más complicada de detectar, con un valor de 0.569, aunque esto puede ser una indicación de que se necesitan más muestras de este tipo.

4.1.2. Support Vector Machine

Este clasificador consigue también muy buenos resultados de exactitud, manteniendo valores superiores a 0.95 en tres de las cinco categorías, pero también sufre de los mismos problemas con las categorías de *executable* y *obfuscation*, en donde obtiene 0.628 y 0.56 respectivamente, lo cúal solo reafirma la necesidad de aumentar el número total de muestras si se desea aumentar la exactitud de estas.

4.1.3. Comparaciones entre clasificadores

Como era de esperar, conociendo las carácteristicas de ambos clasificadores, SVM ha obtenido mejores resultados en casi todas las categorías en comparación con NB, pero no por márgenes significativos (una diferencia de 0.02 o menos en todos los casos). La única excepción a este hecho es en cuanto a las dos categorías de exactitudes más bajas, *executable* y *obfuscation*, en las cuales NB ha obtenido mejores resultados aunque no por mucho. Como conclusión, teniendo en cuenta que SVM es relativamente más costoso en términos de recursos, se podría decir que ambos clasificadores están a la par y que si no se dispone o no se desea emplear muchos recursos entonces NB es la mejor opción a utilizar.

4.2. Resultados de deep learning

	MLP	
Categoría	#BD	Exactitud
Executable	24	0,6893
Family	28	0,968
Novelty	20	0,9084
Obfuscation	20	0,6761
Type	28	0,9927

Cuadro 3: Tabla con los resultados de la arquitectura *Multilayer Perceptron*. Se han escogido solo los valores de banda ancha (bd) con la exactitud (accuracy) más alta.

	CNN	
Categoría	#BD	Exactitud
Executable	16	0,6723
Family	28	0,9906
Novelty	28	0,9654
Obfuscation	28	0,7096
Type	20/24	0,9967

Cuadro 4: Tabla con los resultados de la arquitectura *Convolutional neural network*. Se han escogido solo los valores de banda ancha (bd) con la exactitud (accuracy) más alta.

4.2.1. Convolutional neural network

Esta arquitectura ha obtenido resultados de exactitud muy elevados, superando el 0.96 para casi todas las categorías e incluso consiguiendo valores aceptables para el resto. También cabe destacar que a diferencia de la mayoría de casos, en este la banda ancha para *executable* es de 16.

4.2.2. Multilayer perceptron

Los datos de exactitud obtenidos por esta arquitectura también son altamente positivos, destacando el valor casi perfecto de 0.9927 de *type* y el valor bastante aceptable de 0.6893 para la difícil categoría de *executable*.

4.2.3. Comparaciones entre arquitecturas

La arquitectura CNN, al ser más compleja que MLP, ha obtenido mejores resultados en casi todos los aspectos aunque, como ha ocurrido con los clasificadores de *machine learning*, no ha sido por una diferencia considerable a excepción de *novelty* en donde ha conseguido un valor de 0.9654 (0.06 de margen con MLP). La única categoría en donde MLP ha superado a CNN ha sido en una de las más complicadas, *executable*, con un valor de 0.68 comparado con el 0.67 de CNN, un aumento no demasiado importante en la exactitud. Para concluir, si se desean obtener los mejores resultados (a expensas de una mayor consumición de recursos y un sistema más complejo) la mejor opción es decantarse por la arquitectura CNN.

4.3. Comparaciones finales

Si se compara el método de *machine learning* frente al de *deep learning* queda claro que este último obtiene mejores resultados, aunque lo importante a discutir es si la diferencia entre los resultados de ambos métodos es lo suficientemente importante como para asumir el elevado gasto de recursos que conlleva utilizar las arquitecturas de CNN y MLP frente a los clasificadores NB y SVM. Comprobando estas diferencias se puede observar como en *obfuscation* es donde se encuentran los mayores márgenes e incluso como en *novelty* tanto SVM como NB obtienen una mejor exactitud que MLP. Para el resto de categorías la diferencia suele ser menor que 0.06. Habiendo explicado esto, se puede concluir que si se desea detectar casos de malware ofuscado con una exactitud parcialmente alta y además de ello conseguir los mejores resultados en casi todas las categorías restantes entonces la mejor opción es usar CNN (y si se dispone de menos recursos se puede optar por MLP), pero si lo que se busca es obtener exactitudes mayormente fiables y a un coste de recursos bajo en comparación con el resto, la mejor opción es el clasificador de *machine learning* NB ya que ofrece resultados muy similares a otros métodos pero con una consumición de medios mucho menor.

5

Tipos de Malware utilizados

Hay diversos tipos y familias de malware que son comúmente empleados en ciberataques contra dispositivos IoT de sistema operativo linux, entre estos se han seleccionado aquellos más presentes en la actualidad y que plantean los mayores problemas a entidades gubernamentales y empresas de todo tipo. En la investigación en la que nos basamos [22] se utilizan muestras de dichos tipos de malware para poder comprobar la capacidad de detección de los métodos empleados, además de mezclar entre ellas procesos normales (sin malware) encontrados comunmente en estos dispositivos para así también probar la detección de falsos positivos. Los tipos empleados son principalmente tres: Botnets, Ransomware y Rootkits, y para cada uno de estos tipos de base se utilizan muestras de una o dos familias, aunque en esta investigación se han añadido varias familias más. También se emplean muestras de malware ofuscado, es decir, aquel que emplea técnicas para intentar no ser detectado por sistemas antivirus.

5.1. Botnets

Las *botnets* son colecciones de ordenadores o dispositivos IoT comprometidos que son controlados de forma remota por su creador (BotMaster) a través de una infraestructura de Comando y Control (C&C) común. Proporcionan una plataforma distribuida para poder realizar varias actividades ilegales, como el lanzamiento de ataques de denegación de servicio (DDoS) contra objetivos críticos, la distribución de malware adicional, el phishing y el fraude por clics [29]. Para estas pruebas se usaron varias muestras de malware de las familias Mirai y Bashlite, dos de las más utilizadas en ataques de este tipo.

5.1.1. Mirai

La familia de botnet Mirai fue altamente impactante y comenzó a afectar a dispositivos IoT en 2016. Fue diseñado para aprovechar la falta de seguridad en dispositivos como cámaras de seguridad, enrutadores y grabadoras de video digital, que generalmente tenían contraseñas débiles o predeterminadas. El botnet Mirai se propagó mediante un contínuo escaneo de Internet en busca de dispositivos vulnerables y posteriormente la explotación de las debilidades conocidas en esos dispositivos para tomar su control.

Una vez que un dispositivo estaba comprometido, se convertía en un "bot" y se unía a la red del botnet. La característica distintiva del botnet Mirai era su capacidad para reclutar un gran número de dispositivos comprometidos, comenzado con unos 65 mil en sus primeras horas de vida y llegando a un número estable entre 200 mil y 300 mil más tarde.

El objetivo principal de Mirai era realizar ataques de denegación de servicio distribuido (DDoS) de muy gran tamaño contra objetivos seleccionados. Estos ataques DDoS implicaban abrumar los sistemas y recursos de los objetivos con un tráfico de red masivo, lo que les dificultaba o impedía su funcionamiento normal. Los ataques DDoS del botnet Mirai causaron interrupciones significativas en servicios en línea populares, incluyendo sitios web y plataformas de juego.

Esta familia de botnet demostró la vulnerabilidad generalizada de los dispositivos IoT y resaltó la importancia de implementar medidas de seguridad adecuadas en estos dispositivos. También generó conciencia sobre la necesidad de cambiar las contraseñas predeterminadas y débiles en los dispositivos IoT, así como de mantener los dispositivos actualizados con los últimos parches de seguridad [2].

Una de las muestras adicionales que se han añadido de Mirai es el famoso IRGC botnet (basado en Mirai), relacionado con distintos ataques de denegación de servicio (DDoS) por parte de Cuerpos de la Guardia Revolucionaria Islámica iraní contra el sector financiero estadounidense. Estos ataques dejaron inutilizadas durante largos periodos de tiempo páginas web de distintos bancos, impidiendo así a los clientes de los mismos poder acceder a sus fondos y costando millones de dólares en pérdidas para estas entidades [24].

5.2. Ransomware

El ransomware es un tipo de software malicioso que utiliza la encriptación para secuestrar los datos de las víctimas y exigir un rescate para su liberación. Algunas variantes de ransomware utilizan técnicas de intimidación, como afirmar falsamente ser fuerzas del orden o mostrar imágenes ilegales para asustar a las víctimas.

El primer programa de ransomware, conocido como "AIDS", fue creado en 1989 por Joseph Popp. Este programa se distribuyó como un troyano a través de disquetes, utilizando encriptación de clave simétrica. Sin embargo, esta forma de encriptación presentaba una debilidad, ya que la clave de encriptación estaba presente en el propio ransomware, lo que permitía el análisis y desarrollo de contramedidas. Esta primera versión cifraba los archivos en la unidad "C:" del ordenador, exigiendo un pago de rescate a una dirección de correo en Panamá.

A lo largo de los años, el ransomware ha evolucionado y se han utilizado diferentes métodos de encriptación y técnicas de propagación. En 2006, se adoptó la encriptación asimétrica, que utiliza dos claves diferentes para encriptar y desencriptar los datos. Esto hizo que fuera casi imposible determinar la clave de desencriptación a partir del ransomware mismo. Además, se han utilizado técnicas de bloqueo de dispositivos, amenazas falsas y ataques dirigidos a empresas para aumentar el impacto del malware. En los últimos años ha habido un aumento considerable en el ransomware a nivel mundial. El crecimiento se ha visto facilitado por factores como la conectividad de Internet, la disponibilidad de herramientas de cifrado y el surgimiento de monedas digitales como Bitcoin, que permiten a los criminales recibir rescates de forma anónima.

Para llegar a infectar a un ordenador o dispositivo inteligente se utilizan tácticas de ingeniería social, como correos electrónicos de phishing, ataques a sitios web populares, aplicaciones maliciosas disfrazadas como herramientas útiles y la explotación de vulnerabilidades en los sistemas de los usuarios. Una vez infectada la víctima, el ransomware cifra los datos y muestra un mensaje amenazante que exige un rescate. En cuanto a los métodos de pago de dicho rescate, los delincuentes han utilizado diversas formas de monetización. Algunos métodos incluyen el uso de números de tarificación especial, tarjetas de regalo, servicios de pago en línea, servicios prepagados y criptomonedas como Bitcoin. Bitcoin ha ganado popularidad entre los delincuentes debido a su relativo anonimato y facilidad de transferencia de fondos

ilícitos.

El ransomware se ha convertido en un negocio lucrativo para los delincuentes, con un gran mercado para la venta de herramientas de ransomware, como por ejemplo los kits de explotación (Exploit Kits) que facilitan la entrega y gestión del ransomware. En este negocio también se controla la distribución de ataques y el pago de rescates. Se estima que el valor de este mercado alcanza los millones de dólares anualmente, con cifras alarmantes de víctimas y pérdidas económicas [20]. Para las pruebas se usaron varias muestras de ransomware de la familia Gonnacry, también conocido como WannaCry.

5.2.1. Gonnacry (Wannacry)

El ransomware WannaCry, también conocido como Wana Decrypt0r, WCry, WannaCry, WannaCrypt y WanaCrypt0, fue observado por primera vez durante un ataque masivo en múltiples países el 12 de mayo de 2017. Según informes de varios proveedores de seguridad, un total de 300,000 sistemas en más de 150 países resultaron severamente dañados. El ataque afectó a una amplia gama de sectores, incluyendo salud, gobierno, telecomunicaciones y producción de petróleo/gas.

La dificultad en la protección contra WannaCry radica en su capacidad para propagarse a otros sistemas utilizando un componente de gusano (worm). Esta característica hace que los ataques sean más efectivos y requiere mecanismos de defensa que puedan reaccionar rápidamente y en tiempo real. Además, WannaCry tiene un componente de encriptación basado en criptografía de clave pública.

Durante la fase de infección, WannaCry utiliza las vulnerabilidades EternalBlue y DoublePulsar:

- EternalBlue explota la vulnerabilidad del servidor de mensajes SMB que fue bloqueada por Microsoft el 14 de marzo de 2017. Esta vulnerabilidad permite a los adversarios ejecutar un código remoto en las máquinas infectadas enviando mensajes especialmente diseñados a un servidor SMB v1, conectándose a los puertos TCP 139 y 445 de sistemas Windows que no fueron actualizados. En particular, esta vulnerabilidad afecta a todas las versiones no actualizadas de Windows, desde Windows XP hasta Windows 8.1, excepto Windows 10.

- DoublePulsar es una puerta trasera persistente que puede ser utilizada para acceder y ejecutar código en sistemas previamente comprometidos, lo que permite a los atacantes instalar malware adicional en el sistema. Durante el proceso de distribución, el componente de gusano de WannaCry utiliza EternalBlue para la infección inicial a través de la vulnerabilidad SMB, sondeando activamente los puertos TCP apropiados y, si tiene éxito, intenta implantar la puerta trasera DoublePulsar en los sistemas infectados [1].

5.3. Rootkits

Cuando un sistema es atacado, y los perpetrantes desean que el administrador del mismo no descubra su presencia, se emplean medidas para camuflar su existencia y la de las actividades que puedan realizar en él. Una de estas medidas es una colección de programas llamados *Rootkits*.

Los *Rootkits* aparecieron a finales de la década de 1980 como colecciones de herramientas utilizadas para manipular los archivos de registro de UNIX y ocultar la presencia de ciertos usuarios. Con el tiempo, los atacantes comenzaron a reemplazar programas como "ls","ps" o "netstat" para ocultar sus actividades. Estos programas pronto se empaquetaron junto con versiones manipuladas de programas de inicio de sesión similares para registrar contraseñas en secreto.

En la década de 1990, surgieron rootkits del kernel de sistemas Linux. Estos se cargaban como módulos en tiempo de ejecución para manipular el núcleo del propio sistema operativo, consiguiendo una obfuscación muy completa de las actividades del atacante. Para finales de esa década ya existían rootkits de kernel prácticamente para todos los sistemas operativos UNIX modernos. Al mismo tiempo, aparecieron rootkits para Microsoft Windows.

Para principios de la década de los 2000, los métodos utilizados por los atacantes se habían refinado. Se desarrollaron nuevas formas de inyección de código que permitían plantar un rootkit en el kernel en ejecución sin utilizar módulos. Otros rootkits manipulaban módulos existentes o imágenes del kernel en disco para instalarse.

Los rootkits modernos se han vuelto más sofisticados y difíciles de detectar haciendo que, por ejemplo, el flujo de ejecución se desvíe en diferentes lugares y a niveles más profundos dentro del kernel. Además, se emplean técnicas de ofuscación en los binarios del rootkit para evitar el análisis si se encuentra el malware [6]. En la pruebas realizadas se han empleado

muestras de dos familias de *Rootkits*: “keysniffer” y “maK_It”.

5.3.1. Keysniffer

Keysniffer es un rootkit del tipo LKM (Loadable Kernel Module) que puede clasificarse como tal debido a sus capacidades de intercepción del kernel. Es un keylogger, es decir, registra los eventos del teclado en debugfs, un sistema de archivos en memoria utilizado para la depuración del kernel de Linux. No cuenta con mecanismos específicos para prevenir su detección, excepto que impide el acceso al archivo de registro por parte de usuarios sin privilegios de root, y el módulo se denomina “kisni.ko” de forma predeterminada para evitar sospechas inmediatas al listar los módulos del kernel. Aunque esta familia puede no cumplir con los criterios de qué se considera un rootkit debido a su falta de funcionalidades específicas de ocultamiento, sí cumple con los criterios establecidos por Harley y Lee [11] [13].

5.4. Ofuscación

La ofuscación se refiere a la técnica que consiste en ocultar información de manera que otras personas no puedan encontrar su verdadero significado. Los proveedores de software normalmente la utilizan para hacer más difícil el realizar ingeniería inversa sobre sus programas. Los creadores de malware aprovechan esta técnica para ocultar programas maliciosos, utilizando diversas transformaciones de ofuscación para que el malware sea difícil de analizar, lo que complica el descubrir sus intenciones maliciosas.

La teoría de la ofuscación implica tomar un programa original P y aplicar una función de transformación T para generar un programa P’ que cumpla con ciertas propiedades:

- P’ es difícil de ingeniería inversa.
- P’ mantiene la funcionalidad de P.
- P’ tiene un rendimiento comparable a P.

Muchos programas maliciosos utilizan técnicas de ofuscación, como metamorfismo y polimorfismo, para evadir la detección basada en firmas. Estas técnicas pueden cambiar fácilmente las firmas comúmente asociadas a dicho malware, lo que dificulta su detección. Un ejemplo de técnica de ofuscación consiste en agregar instrucciones basura (como instrucciones nop) al

código original. Esto modifica las firmas y hace que el código obfuscado sea menos detectable por los escáneres de malware, aumentando la tasa de falsos negativos [26].

5.4.1. Flatten

La técnica del *flattening* consiste en modificar la estructura del código fuente de un programa para que las metas de las ramas (branch targets) no puedan ser fácilmente determinadas mediante análisis estático, dificultando así la comprensión del programa.

El proceso básico para llevar a cabo el *flattening* en una función es el siguiente: primero, se descompone el cuerpo de la función en bloques básicos (basic blocks) y luego se colocan todos estos bloques, que originalmente estaban en diferentes niveles de anidamiento, uno al lado del otro. Estos bloques básicos ahora de igual nivel se encapsulan en una estructura selectiva (como un bloque switch en el lenguaje C++), con cada bloque en un caso separado, y esta selección se encapsula a su vez en un bucle. Finalmente, se garantiza el flujo correcto de control mediante una variable de control que representa el estado del programa, la cual se establece al final de cada bloque básico y se utiliza en los predicados del bucle y la selección que lo envuelven [16].

5.4.2. BCF (Bogus control flow)

La técnina de inserción de flujo de control falso (Bogus Control Flow) consiste en modificar el esquema de flujo de control de una función, agregando a la misma una estructura de salto condicional que apunta tanto al bloque básico original como a un bloque básico falso que retrocede al bloque de salto condicional. Un predicado opaco (ver la sección 5.4.5) [4] se encarga de asegurar que en tiempo de ejecución solo se ejecute el bloque básico original y a la vez también garantiza que el optimizador no pueda simplificar el listado de llamadas resultante identificando el código inactivo [14].

5.4.3. Virtualize

Cada función en el programa se convierte en un intérprete que tiene su propio conjunto único de *bytecode*. Cuando se ejecuta, el programa interpretará cada *bytecode*, creando una máquina virtual que luego ejecuta las instrucciones contenidas en el interior del *bytecode*, representando así la función original [4].

5.4.4. Sub (Instruction Substitution)

La sustitución de instrucciones (instruction substitution) se implementa utilizando una biblioteca de instrucciones equivalentes. Se reemplaza una instrucción en el cuerpo del código por otra que es equivalente, consiguiendo así cambiar drásticamente la firma del código y haciéndolo mucho más difícil de desofuscar, especialmente si la biblioteca de instrucciones equivalentes no está disponible [3].

5.4.5. Opaque Predicates (Addopaque)

En esta técnica se utilizan los predicados opacos (Opaque Predicates), un predicado opaco es una condición o rama (predicado) que se evalúa como un valor booleano. El resultado de este predicado está predeterminado por el programador pero es desconocido para cualquier otra persona (opaco). Cuando se alcanza este valor predefinido, se ejecuta el código original [4].

5.4.6. Upx packing

El *packing* es una técnica de ofuscación avanzada para crear variantes de malware más complejas y sofisticadas, dificultando el análisis estático. En esta técnica, el código real del malware se comprime o cifra en una forma diferente pero semánticamente idéntica. El nuevo archivo ejecutable que se obtiene como resultado consta de un código binario de malware empaquetado o envuelto (comprimido o cifrado) y un código de desempaquetado. Este código de desempaquetado define el punto de entrada del nuevo archivo de malware, que es invocado por el sistema operativo. Luego, se ejecuta el código de desempaquetado, desempaquetando así el código de malware original en la memoria en tiempo de ejecución. En otras palabras, la rutina de desempaquetado representa el punto de entrada original (OEP) [25].

En este caso es llamado UPX ya que se emplea el empaquetador o *packer* UPX, que es un programa *open source* de compresión de ficheros ejecutables [23].

6

Descripción detallada del escenario de capturas y análisis

Esta investigación experimental se puede dividir en distintas etapas, una primera en la que se capturan las muestras con las que se va a trabajar y otras en las que se procesan dichas muestras y se entrenan y validan las distintas redes.

6.1. Captura de las señales

El proceso de captura de las trazas se realiza conectando la Raspberry Pi via SSH y el osciloscopio (*MSO 5074*) via USB al ordenador, que funcionará como dispositivo de mando durante el procedimiento. Posteriormente, se enciende el osciloscopio y se activa para recibir órdenes por parte del ordenador, se conecta la sonda a uno de los canales del osciloscopio y se coloca sobre el procesador (CPU) de la Raspberry.

Una vez se tiene todo el equipamiento en posición como se muestra en la figura 2, y asumiendo que ya se han pasado las muestras de malware a la propia Rpi, se procede a ejecutar el malware cuya traza se quiere capturar via comando SSH y a su vez el script de python, que comenzará a mandar las órdenes necesarias al osciloscopio para que capture las señales electromagnéticas que emanan del procesador siguiendo una serie de parámetros de voltaje, frecuencia y duración de la señal, y guarde la traza en el dispositivo de mando.

El proceso de ejecución del malware en la Raspberry y el script de captura de las señales por parte del osciloscopio se repite para todas las muestras que se quisieran obtener.

Figura 2: Posicionamiento de los componentes involucrados

6.2. Procesamiento de las trazas

Una vez se dispone de las trazas capturadas (en este caso por el osciloscopio *PicoScope® 6000 Series*) se tienen que procesar para poder distinguir entre el ruido generado de manera normal por el procesador del dispositivo y las distintas fluctuaciones anómalas que puedan indicar la presencia de malware.

Se computa el espectrograma de la señal guardada en cada traza y a partir de este se extraen mediante el método NICV (Normalized Inter-Class Variance for Detection of Side-Channel Leakage) [5] las secciones del ancho de banda de la frecuencia en donde existan picos o descensos no naturales.

En el caso de las muestras procesadas que se vayan a utilizar para entrenar la red de *machine learning* se realiza un paso más de procesamiento de las mismas mediante el algoritmo LDA (Linear Discriminant Analysis) [21] para reducir en gran cantidad la dimensión de estas ya que

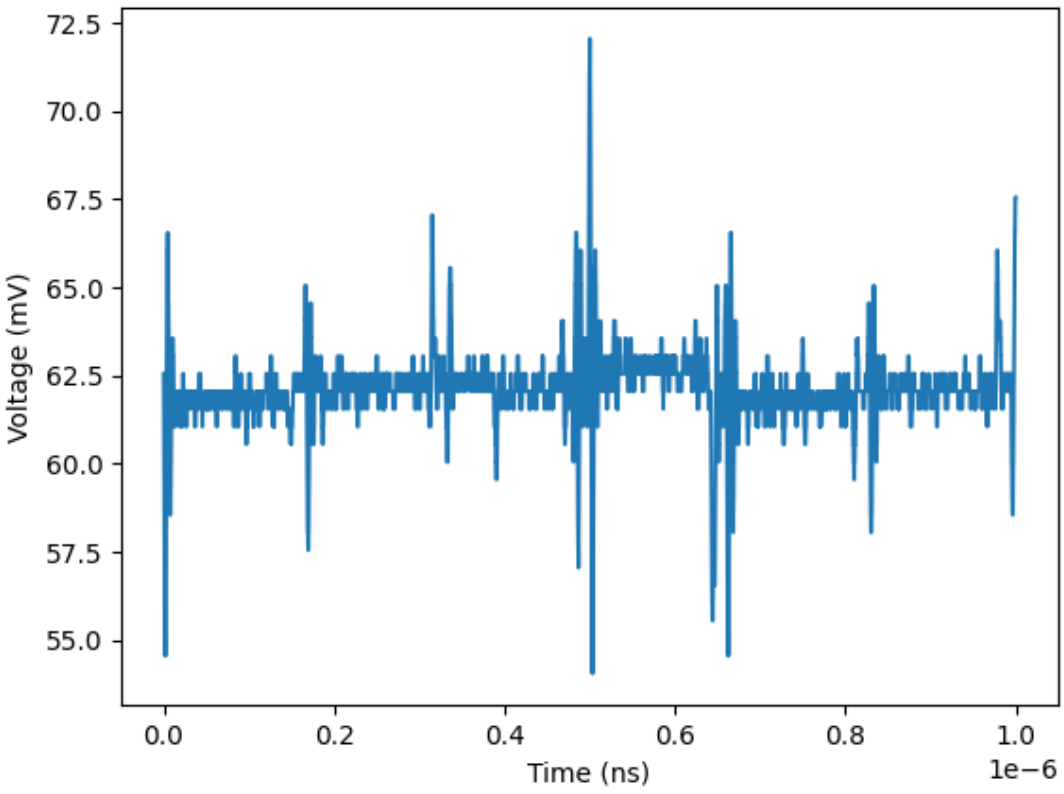

Figura 3: Ejemplo gráfico de la traza capturada del malware de la familia Mirai

estos métodos de aprendizaje no escalan bien conforme aumenta el número de características de los datos de entrada.

6.3. Entrenamiento y validación de la red de machine learning

Durante esta etapa se procedió con el entrenamiento y validación de los dos algoritmos de clasificación de *machine learning*, *Naive Bayes* (NB) y *Support Vector Machine* (SVM).

La mayoría de los pasos se completan con un solo script de shell ".sh", que ha tenido que ser levemente modificado junto con otro script auxiliar debido a un error que se producía al introducir una de las direcciones de las carpetas de los acumuladores en el momento de lanzarlo. Este error se producía debido al carácter '\' que se encontraba al final de dicha dirección y que causaba un conflicto en el momento de filtrar todos los archivos dentro del directorio proporcionado con la librería de python **Glob**, empleada en el script de python auxiliar llamado

“nicv.py” en un bucle “for” que comienza en la línea 68 y que se encontraba así antes:

```
for i in range (len (unique_names)):
        # get the two corresponding accumulators {acc_x, acc_xx
   } and
        # the size of each acc.

        reg_exp = re.compile (path_acc + re.escape (
   unique_names [i])  + r'_\d+_'+ 'acc_x.npy')
        tmp_acc_x  = list (filter (reg_exp.match, glob.glob (
   path_acc + '/*'))) [0]

        reg_exp = re.compile (path_acc + re.escape (
   unique_names [i])  + r'_\d+_'+ 'acc_xx.npy')
        tmp_acc_xx = list (filter (reg_exp.match, glob.glob (
   path_acc + '/*'))) [0]

        acc_names.append ([tmp_acc_x, tmp_acc_xx])
        counts_acc.append (int (tmp_acc_x.split ('_')[-3]))
```

Y así después (se encuentra marcado con un comentario el lugar donde se ha realizado los cambios):

```
for i in range (len (unique_names)):
        # get the two corresponding accumulators {acc_x, acc_xx
   } and
        # the size of each acc.

        reg_exp = re.compile (path_acc + r'\\' + re.escape (
   unique_names [i])  + r'_\d+_'+ 'acc_x.npy') # Aquí se ha
   añadido --> + r'\\'
        tmp_acc_x  = list (filter (reg_exp.match, glob.glob (
   path_acc + '/*'))) [0]
```

```
        reg_exp = re.compile (path_acc + r'\\' + re.escape (
    unique_names [i])  + r'_\d+_'+ 'acc_xx.npy') # Aquí se ha
    añadido --> + r'\\'
        tmp_acc_xx = list (filter (reg_exp.match, glob.glob (
    path_acc + '/*'))) [0]

        acc_names.append ([tmp_acc_x, tmp_acc_xx])
        counts_acc.append (int (tmp_acc_x.split ('_')[-3]))
```

En dicho script de shell se encuentran los comandos para ejecutar el procesamiento de las muestras, la generación de los acumuladores, el entrenamiento y la validación de ambos modelos de clasificación. Los dos primeros comandos solo deberán ejecutarse en esta etapa ya que los archivos generados se reutilizarán también en la siguiente (entrenamiento y validación de la red neuronal).

Para el sistema de entrenamiento y validación se generan una serie de listas, marcadas por el modelo de clasificacion (por ejemplo NB o SVM) y por la carácterística por la que se clasifican las muestras (por ejemplo el tipo o la familia del malware). Parte de estas listas serán comunes a ambas redes (machine learning y neuronal) y contienen la dirección a las muestras, dividiéndolas además en los grupos de entrenamiento, prueba y validación.

Los detalles específicos de los pasos que se llevaron a cabo se encuentran en el manual del anexo titulado “Manual de entrenamiento y validacion de la red usando machine learning”

6.4. Entrenamiento y validación de la red neuronal

Este último procedimiento consistió en el entrenamiento y validación de la red neuronal siguiendo dos arquitecturas distintas, *Convolutional neural network* (CNN) y *Multilayer perceptron* (MLP).

Como se ha mencionado anteriormente, en los scripts utilizados para realizar este paso no se han incluido los comandos de procesamiento y generación de los acumuladores ya que se emplean los archivos generados en el paso anterior. Los scripts empleados son principalmente dos, el primero es el encargado de entrenar la red neuronal (para ambas arquitecturas)

y generar los archivos de modelo ".h5" y el segundo de validarla y obtener los resultados de la precisión de las distintas arquitecturas para las varias clasificaciones y bandas anchas. El segundo script en específico tuvo que ser creado a partir de otro existente ya que los scripts otorgados por la anterior investigación no permitían la posibilidad de validar la red neuronal con bandas anchas diferentes para cada clasificación.

Para todo ese procedimiento se emplea de manera vital la librería **tensorflow** de python que a su vez funciona con un rendimiento especialmente elevado si se dispone de una tarjeta gráfica (GPU) compatible con las herramientas *Nvidia CUDA®*. En este experimento en específico se ha empleado una GPU *Nvidia Geforce 1080*, la cual ha ayudado de manera exponencial a reducir el tiempo de duración del experimento.

Los detalles específicos de los pasos que se llevaron a cabo se encuentran en el manual del anexo titulado "Manual de entrenamiento y validacion de la red usando deep learning"

6.5. Problemas encontrados

En el trabajo de investigación en el que se basa este [22] se utiliza un tipo de osciloscopio específico, un *PicoScope® 6000 Series* que generaba un formato de muestras ".dat" para el cuál el resto de scripts de procesamiento y extracción de la banda ancha estaban diseñados.

Para comprobar en qué consistía ese formato se revisaron los scripts originales que lo generaban, principalmente el llamado "picoscope.py" y también se diseñó un pequeño script de python (disponible en la sección D.1) para observar el contenido de una de las trazas que se aportaban.

Observando esos contenidos, se intentó diseñar un script de python que, mediante instrucciones al osciloscopio del que disponíamos, consiguiera no solo capturar las señales si no además que el archivo de trazas generado fuera del mismo formato que el generado por un *PicoScope®*. Se probaron varios de estos scripts, intentando emplear los mismos parámetros utilizados en los scripts originales diseñados para el *PicoScope®* como por ejemplo la tasa de muestreo, el valor del voltaje en donde se lanzaría el trigger y el tiempo de medición de la señal, además de tener en cuenta que el tamaño de las trazas era siempre de 10MB y que estos se escribían en formato "int16" (enteros de 16 bits con signo) empleando la librería de python **Struct**. Existe un ejemplo de uno de los últimos scripts que se probaron (y no dió los resultados esperados) en el anexo D.2.

Tras diversos intentos fallidos no se consiguió replicar ese tipo de archivo de traza, por lo que se decidió no capturar las nuevas muestras de malware que se tenían preparadas para cumplir con los objetivos de esta investigación (comprobar si la red neuronal podía detectar nuevas variantes de malware) y continuar únicamente con los 30GB de muestras que se proporcionaban en el trabajo anterior.

En caso de haber dispuesto de un *PicoScope® 6000 Series*, se hubieran capturado las nuevas muestras siguiendo un proceso de captura similar y el resto de etapas de la investigación hubieran sido exactamente iguales.

7

Conclusiones y Líneas Futuras

7.1. Conclusiones

A partir de todo el trabajo de investigación y de los experimentos realizados se pueden llegar a diversas conclusiones:

- **Ofuscación**. Aunque díficiles de clasificar comparadas con el resto, las muestras con ofuscación han podido también ser detectadas hasta en un 70 % con algunos métodos, lo que sugiere que con más muestras de entrenamiento de dicho tipo se pueden conseguir aún mejores resultados de detección y adquirir así una nueva técnica de descubrimiento de estos elusivos tipos de malware sin tener que emplear programas o sistemas dentro del propio dispositivo que se desea proteger.

- **Osciloscopio**. Tras los problemas experimentados durante el proceso de recogida de muestras, es obvio que la necesidad de utilizar un osciloscopio específico (*PicoScope® 6000 Series*) ha sido el factor determinante en el fracaso del cumplimiento del objetivo de este trabajo de investigación.

- **Detección del malware**. Los resultados generales de los métodos de IA empleados para detectar el malware confirman la viabilidad del uso de este tipo de sistema para la protección de dispositivos IoT (Internet of Things) frente a ataques reales. La capacidad de algunos métodos de detectar el tipo, familia e incluso la variante exacta del malware que se está ejecutando en un dispositivo en específico sin necesidad de tomar parte de los recursos internos del mismo ejecutando programas es un indício claro de que nos encontramos ante un nuevo método de defensa anti-malware.

7.2. Líneas Futuras

En cuanto a posibles líneas que se podrían explorar en futuros proyectos o pasos que se podrían tomar en un futuro para mejorar y continuar la investigación es indispensable mencionar los siguientes:

- **Mejorar captura de trazas**. El principal obstáculo de la investigación actual fue la necesidad de utilizar un osciloscopio específico y extramadamente caro. Se debería seguir trabajando en una manera de poder generar archivos de trazas del mismo formato pero con osciloscopios comúnmente encontrados en la mayoría de instituciones o sectores.

- **Mayor número de muestras**. Como se ha podido observar en los resultados de las redes, hay aún ciertas categorías en las que no se ha conseguido un porcentaje de detección que se pueda considerar del bastante fiable (superior al 80 %). Es por esto que es primordial obtener un número de muestras de entrenamiento de las redes mayor que el utilizado en la investigación actual, aplicando además especial atención a muestras de malware ofuscado y al empleo de nuevas variantes, familias e incluso tipos de malware.

- **Nuevos sistemas operativos**. En los experimentos realizados solo se ha trabajado con un dispositivo en donde lanzar el malware de sistema operativo linux por lo que si se desea obtener un sistema de detección que permita proteger a ordenadores y dispositivos sin importar el sistema operativo que tengan es crucial ampliar los tipos de malware de las muestras (esta acción va ligada también al punto anterior) a aquellos que sean específicos de otros sistemas operativos, como por ejemplo Windows (en todas o ciertas versiones) o Mac OS.

- **Nuevos métodos de *deep* y *machine learning***. Siguiendo con la temática de los puntos anteriores, también es una buena idea buscar arquitecturas (o combinaciones de arquitecturas) y clasificadores nuevos con los que poder comparar la exactitud de los resultados y posiblemente obtener porcentajes de aciertos aún mayores de los conseguidos actualmente.

Referencias

[1] Maxat Akbanov, Vassilios G Vassilakis y Michael D Logothetis. "WannaCry ransomware: Analysis of infection, persistence, recovery prevention and propagation mechanisms". En: *Journal of Telecommunications and Information Technology* 1 (2019), págs. 113-124.

[2] Manos Antonakakis et al. "Understanding the mirai botnet". En: *26th {USENIX} security symposium ({USENIX} Security 17)*. 2017, págs. 1093-1110.

[3] Arini Balakrishnan y Chloe Schulze. "Code obfuscation literature survey". En: *CS701 Construction of compilers* 19 (2005), pág. 31.

[4] Shrenik Bhansali et al. "A first look at code obfuscation for webassembly". En: *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 2022, págs. 140-145.

[5] Shivam Bhasin et al. "NICV: normalized inter-class variance for detection of side-channel leakage". En: *2014 International Symposium on Electromagnetic Compatibility, Tokyo*. IEEE. 2014, págs. 310-313.

[6] Andreas Bunten. "Unix and linux based rootkits techniques and countermeasures". En: *16th Annual First Conference on Computer Security Incident Handling, Budapest*. 2004.

[7] Huazhou Chen et al. "A deep learning CNN architecture applied in smart near-infrared analysis of water pollution for agricultural irrigation resources". En: *Agricultural Water Management* 240 (2020), pág. 106303.

[8] Shane S Clark et al. "{WattsUpDoc}: Power side channels to nonintrusively discover untargeted malware on embedded medical devices". En: *2013 USENIX Workshop on Health Information Technologies (HealthTech 13)*. 2013.

[9] Fei Ding et al. "DeepPower: Non-intrusive and deep learning-based detection of IoT malware using power side channels". En: *Proceedings of the 15th ACM Asia conference on computer and communications security*. 2020, págs. 33-46.

[10] Matt W Gardner y SR Dorling. "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences". En: *Atmospheric environment* 32.14-15 (1998), págs. 2627-2636.

[11] David Harley y Andrew Lee. *The root of all evil?-rootkits revealed*. 2007.

[12] Christian Janiesch, Patrick Zschech y Kai Heinrich. "Machine learning and deep learning". En: *Electronic Markets* 31.3 (2021), págs. 685-695.

[13] Juho Junnila. *Effectiveness of Linux Rootkit Detection Tools*. University of Oulu, 2020.

[14] Pascal Junod et al. "Obfuscator-LLVM–software protection for the masses". En: *2015 ieee/acm 1st international workshop on software protection*. IEEE. 2015, págs. 3-9.

[15] Haider Adnan Khan et al. "IDEA: Intrusion detection through electromagnetic-signal analysis for critical embedded and cyber-physical systems". En: *IEEE Transactions on Dependable and Secure Computing* 18.3 (2019), págs. 1150-1163.

[16] Tımea László y Ákos Kiss. "Obfuscating C++ programs via control flow flattening". En: *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica* 30.1 (2009), págs. 3-19.

[17] Batta Mahesh. "Machine learning algorithms-a review". En: *International Journal of Science and Research (IJSR).[Internet]* 9.1 (2020), págs. 381-386.

[18] David Meyer y FT Wien. "Support vector machines". En: *The Interface to libsvm in package e1071* 28.20 (2015), pág. 597.

[19] *Number of connected IoT devices growing 16 % to 16.7 billion globally*. URL: https://iot-analytics.com/number-connected-iot-devices/.

[20] Philip O'Kane, Sakir Sezer y Domhnall Carlin. "Evolution of ransomware". En: *Iet Networks* 7.5 (2018), págs. 321-327.

[21] Cheong Hee Park y Haesun Park. "A comparison of generalized linear discriminant analysis algorithms". En: *Pattern Recognition* 41.3 (2008), págs. 1083-1097.

[22] Duy-Phuc Pham et al. "Obfuscation Revealed: Leveraging Electromagnetic Signals for Obfuscated Malware Classification". En: *ACSAC 2021 - Annual Computer Security Applications Conference*. Virtual Event, France: ACM, dic. de 2021, págs. 1-14. DOI: 10.1145/3485832.3485894. URL: https://hal.science/hal-03374399.

[23] Kevin A Roundy y Barton P Miller. "Binary-code obfuscations in prevalent packer tools". En: *ACM Computing Surveys (CSUR)* 46.1 (2013), págs. 1-32.

[24] *Seven Iranians Working for Islamic Revolutionary Guard Corps-Affiliated Entities Charged for Conducting Coordinated Campaign of Cyber Attacks Against U.S. Financial Sector | OPA | Department of Justice*. URL: https://www.justice.gov/opa/pr/seven-iranians-working-islamic-revolutionary-guard-corps-affiliated-entities-charged.

[25] Jagsir Singh y Jaswinder Singh. "Challenge of malware analysis: malware obfuscation techniques". En: *International Journal of Information Security Science* 7.3 (2018), págs. 100-110.

[26] P Vinod et al. "Survey on malware detection methods". En: *Proceedings of the 3rd Hackers' Workshop on computer and internet security (IITKHACK'09)*. 2009, págs. 74-79.

[27] Xiao Wang et al. "Deep learning-based classification and anomaly detection of side-channel signals". En: *Cyber Sensing 2018*. Vol. 10630. SPIE. 2018, págs. 37-44.

[28] Geoffrey I Webb, Eamonn Keogh y Risto Miikkulainen. "Naïve Bayes." En: *Encyclopedia of machine learning* 15.1 (2010), págs. 713-714.

[29] Hossein Rouhani Zeidanloo et al. "A taxonomy of Botnet detection techniques". En: *Proceedings - 2010 3rd IEEE International Conference on Computer Science and Information Technology, ICCSIT 2010* 2 (2010), págs. 158-162. DOI: 10.1109/ICCSIT.2010.5563555.

Apéndice A
Manual de Instalación

Manual de instalación del sistema de análisis

Adrián Meis Asensi

Grado de Ingeniería Informática, Universidad de Málaga

Julio, 2023

El manual de instalación del sistema de análisis consta de los siguientes pasos:

1. Descargar desde el repositorio de GitHub.
2. Instalar todos los plugins estipulados en el archivo requirements.txt.
3. Crear una carpeta donde guardar las muestras.
4. Descargar los modelos pre-entrenados (Si se desea solo validar las redes).
5. Ejecutar el script update-lists

A.1. Descargar desde el repositorio de GitHub

El primer paso para instalar el sistema de análisis es descargar el código fuente desde el repositorio de GitHub. Para ello, siga los siguientes pasos:

1. Visite la página del repositorio de GitHub: https://https://github.com/adrianMeisA/TFG-SideChannel-Analysis.
2. Clone o descargue el repositorio y guárdelo donde desee.

A.2. Instalar todos los plugins estipulados en el archivo requirements

El sistema de análisis utiliza varios plugins que deben instalarse antes de poder ejecutarlo. Para instalar estos plugins se necesitará la versión de python 3.6 ya que versiones superiores pueden dar errores de compatibilidad, si se desea, puede utilizar un entorno virtual aunque la generación del mismo no se explicará en este documento. Python 3.6 se puede descargar desde la página oficial en https://www.python.org/downloads/release/python-368/

1. Abra una terminal o línea de comandos en la carpeta donde extrajo el código fuente del sistema de análisis.
2. Ejecute el siguiente comando para instalar los plugins requeridos:

```
pip install -r requirements.txt
```

A.3. Crear una carpeta donde guardar las muestras

Si desea emplear las muestras recogidas personalmente para el entrenamiento de las redes, es necesario crear una carpeta donde guardarlas dentro del directorio principal de análisis. Para ello, siga los siguientes pasos:

1. Abra una terminal o línea de comandos en la carpeta donde extrajo el código fuente del sistema de análisis.
2. Cree una nueva carpeta con el siguiente comando:

```
mkdir muestras
```

3. La carpeta "muestras" se creará en la ubicación donde se ejecutó el comando. Puede asignarle el nombre que desee.
4. Si desea utilizar el paquete de 12000 muestras que se han empleado en el experimento puede hacerlo desde https://zenodo.org/record/5414107 bajo el nombre "raw_data _reduced_dataset.zip"
5. Si desea emplear las muestras ya procesadas solo para el proceso de validación de las redes las puede obtener desde https://zenodo.org/record/5414107 bajo el nombre "traces_selected_bandwidth"

A.4. Descargar los modelos pre-entrenados (Si se desea solo validar las redes)

Los modelos pre-entrenados son necesarios para la validación de las redes si estas no se han entrenado anteriormente o si no se desean entrenar. Puede descargar los modelos pre-entrenados desde el repositorio de GitHub ahma-hub. Para ello, siga los siguientes pasos:

1. Visite la página del repositorio de GitHub: https://github.com/ahma-hub/pretrained_models.
2. Descargue o clone todo el repositorio en una carpeta dentro de la carpeta principal de análisis.
3. Entre en las carpetas CNN y MLP.
4. Ejecute el script de descompresión con el siguiente comando:

```
./run_decompression.sh
```

Tenga en cuenta que debe tener 7zip instalado en su sistema.

A.5. Ejecutar el script update-lists

Una vez tenga las muestras y haya extraído los modelos pre-entrenados, es necesario ejecutar el script "update-lists" para actualizar las listas de muestras.

Si se desea solo validar las redes entonces siga estos pasos:

1. Abra una terminal o línea de comandos en la carpeta donde extrajo el código fuente del repositorio de análisis.
2. Ejecute el siguiente comando:

```
./update_lists.sh lists_selected_bandwidth/ traces_40bd/
```

Si va a proceder con el proceso de entrenamiento y validación de las redes desde cero entonces siga estos pasos:

1. Abra una terminal o línea de comandos en la carpeta donde extrajo el código fuente del repositorio de análisis.

2. Ejecute el siguiente comando:

```
./update_lists.sh lists_reduced_dataset/ muestras/
```

3. Asegurese de cambiar el nombre de la carpeta “muestras” por el nombre de la carpeta donde haya guardado las muestras “.dat” sin procesar.

¡Listo! Ahora ha ya tiene preparado el directorio para comenzar el entrenamiento y/o validación de las redes a partir de las muestras.

Apéndice B
Manual de entrenamiento de la red usando machine learning

Manual de entrenamiento y validación de la red usando machine learning

Adrián Meis Asensi

Grado de Ingeniería Informática, Universidad de Málaga

Julio, 2023

Este manual describe el proceso de entrenamiento y validación de la red de machine learning desde cero, incluyendo los clasificadores SVM y NB. Se consta de los siguientes pasos:

1. Comprobar los pasos del manual de instalación.
2. Ejecutar el script "run_ml_on_reduced_dataset.sh".
3. Comprobar creación de carpetas.
4. Comprobar resultados.

B.1. Comprobar los pasos del manual de instalación.

Para poder realizar satisfactoriamente todo el proceso detallado en los pasos siguientes de este manual es necesario haber completado todos los del manual de instalación.

- Compruebe que tiene el repositorio de análisis y todos los requerimientos necesarios para continuar.
- Si le falta algún puede volver al manual de instalación y completarlo.

B.2. Ejecutar el script “run_ml_on_reduced_dataset.sh”

En este script se realizan todos los pasos necesarios, se procesan las trazas, se obtienen los acumuladores, se entrenan los clasificadores y se validan los resultados. Para ejecutarlo siga los siguientes pasos:

1. Abra una terminal o línea de comandos en la carpeta donde extrajo el código fuente del sistema de análisis.

2. Ejecute el siguiente comando:

```
./run_ml_on_reduced_dataset.sh
```

B.3. Comprobar creación de carpetas

Si todos los procesos dentro del script ejecutado han finalizado correctamente se deben haber creado varios directorios nuevos dentro del directorio principal de análisis. Para comprobarlo, siga los siguientes pasos:

1. Compruebe la existencia de una carpeta llamada “acc_raw_reduced_dataset” y que en su interior se encuentran diversos archivos cuyos nombres finalizan en “acc_x” y “acc_xx”

2. Compruebe la existencia de una carpeta llamada “traces_40bd_reduced_dataset” y que en su interior se encuentran las trazas procesadas en formato “.npy”

3. Compruebe la existencia de una carpeta llamada “acc_stft_reduced_dataset” y que en su interior se encuentran diversos archivos cuyos nombres finalizan en “acc_x” y “acc_xx”

4. Compruebe que dentro de la carpeta “lists_reduced_dataset” se encuentran diversos ficheros cuyos nombres comienzan por “NB”, “SVM”, “LDA”, “extracted_bd_files” y “transformed_traces”

Si se han podido completar estos pasos entonces la ejecución del script ha sido exitosa.

B.4. Comprobar resultados.

Por la terminal donde se ha ejecutado el script se ha debido mostrar como última línea de texto una tabla con los resultados finales de la evaluación de los clasificadores LDA+NB y LDA+SVM. También se pueden comprobar los resultados de dos formas distintas:

1. Abra una terminal o línea de comandos en el directorio principal de análisis.
2. Ejecute el script de lectura de logs con el siguiente comando:

```
python ml_analysis/read_logs.py --path ml_analysis/log-
   evaluation_reduced_dataset.txt
```

O como otra opción, puede comprobar los resultados de manera manual mirando directamente los logs:

1. Diríjase a la carpeta llamada "ml_analysis" que se encuentra en el directorio principal de análisis.
2. Dentro de esta carpeta encontrará un archivo de texto llamado "log-evaluation_reduced_dataset" en donde podrá observar todos los resultados obtenidos por la evaluación.

Ese es el final de este manual, cabe recalcar que para poder realizar los pasos del manual "Manual de entrenamiento y validacion de la red usando deep learning" se debe haber completado este.

Apéndice C
Manual de entrenamiento de la red usando deep learning

Manual de entrenamiento y validación de la red usando deep learning

Adrián Meis Asensi

Grado de Ingeniería Informática, Universidad de Málaga

Julio, 2023

Este manual consta de los siguientes pasos:

1. Comprobar los pasos del manual de de entrenamiento y validación de la red usando machine learning
2. Ejecutar “run_dl_on_reduced_dataset.sh”.
3. Comprobar creación de archivos de arquitectura y logs.
4. Ejecutar “run_dl_on_reduced_dataset_validate.sh”.
5. Comprobar resultados.

C.1. Comprobar los pasos del manual de de entrenamiento y validación de la red usando machine learning

Para poder realizar satisfactoriamente todo el proceso detallado en los pasos siguientes de este manual es necesario haber completado todos los del manual de de entrenamiento y validación de la red usando machine learning ya que en dicho manual se realiza el procesamiento de las trazas y la generación de los acumuladores, cosa que se da por hecho en este.

- Compruebe que tiene el repositorio de análisis y las carpetas de “acc_stft_reduced_dataset”, las listas actualizadas dentro de la carpeta de “lists_reduced_dataset” y la carpeta de “traces_40bd_reduced_dataset”
- Si le falta alguna de estas puede volver al manual de entrenamiento y validación de machine learning y completarlo.

C.2. Ejecutar “run_dl_on_reduced_dataset.sh”

En este script se realizará el entrenamiento de las dos arquitecturas para cada banda ancha y categoría (*tagmap*). Es recomendable instalar las herramientas *Nvidia CUDA®* si se dispone de una GPU compatbible ya que esto acelerará el proceso de entrenamiento de manera exponencial.

1. Abra una terminal o línea de comandos en el directorio principal de análisis.
2. Ejecute el siguiente comando, en donde ARCH será “cnn” o “mlp” dependiendo de la arquitectura que se desee entrenar, EPOCHS será el número de veces que el algoritmo de aprendizaje trabajará sobre todo el dataset de entrenamiento (por ejemplo 100) y BATCH es el número de muestras sobre las que se itera antes de actualizar los parámetros internos del modelo (por ejemplo 100):

```
./run_dl_on_reduced_dataset.sh lists_reduced_dataset/
  acc_stft_reduced_dataset ARCH EPOCHS BATCH
```

Cabe recalcar que en la ejecución del comando es de vital importancia el hecho de que el directorio “acc_stft_reduced_dataset” no se escriba terminado en ‘\’ ya que esto puede llevar a errores.

C.3. Comprobar creación de archivos de arquitectura y logs

Si los entrenamientos se han completado con éxito se habrán creado una serie de archivos del formato “.h5” en el directorio principal de análisis. Para comprobarlo siga los siguientes pasos:

1. Diríjase al directorio principal de análisis

2. Compruebe que existen una serie de archivos cuyos nombres comienzan por “cnn” y “mlp” y terminan en “band_(número del 8 al 28)”
3. Cree una carpeta llamada “trained_models” o con el nombre que se desee en el directorio principal
4. Dentro de dicha carpeta que se ha creado se deben crear dos carpetas llamadas “cnn” y “mlp”
5. Mueva los archivos “.h5” a esas dos carpetas dependiendo del nombre con el que comiencen. Si comienzan por “cnn” entonces se deben mover a “trained_models/cnn” y si comienzan por “mlp” se deben mover a “trained_models/mlp” (El nombre de “trained_models” será el que se la haya otorgado a ese directorio en el paso 3)
6. Diríjase a la carpeta “dl_analysis” dentro del directorio principal
7. En ella encontrará dos ficheros cuyos nombres comienzan por “training”, estos son los logs del entrenamiento de las dos arquitecturas

C.4. Ejecutar “run_dl_on_reduced_dataset_validate.sh”

Este script se encargará de la validación de las arquitecturas y la obtención de los resultados finales de las mismas. También es recomendable disponer de las herramientas *Nvidia CUDA®* en este caso.

1. Abra una terminal o línea de comandos en el directorio principal de análisis.
2. Ejecute el siguiente comando, en donde “trained_models” será el nombre de la carpeta en la que se encuentras las carpetas “cnn” y “mlp” con los archivos en formato “.h5” de los modelos.

```
./run_dl_on_reduced_dataset_validate.sh
  lists_reduced_dataset/ trained_models/
  acc_stft_reduced_dataset
```

Cabe recalcar que en la ejecución del comando es de vital importancia el hecho de que el directorio “acc_stft_reduced_dataset” no se escriba terminado en ‘\’ ya que esto puede llevar a errores.

C.5. Comprobar resultados

Si el proceso de validación se ha realizado con éxito se generarán logs con los resultados obtenidos. Para comprobarlos siga los siguientes pasos:

1. Diríjase a la carpeta “dl_analysis”
2. En ella encontrará un fichero de texto llamado “evaluation_log_DL” donde podrá revisar los resultados obtenidos en el proceso de validación para cada arquitectura, categoría y banda ancha.

Apéndice D
Código generado

Todo el código mencionado en esta memoria y que no esté presente en esta sección se encuentra disponible en la siguiente dirección web:

https://github.com/adrianMeisA/TFG-SideChannel-Analysis

D.1. Script para observar el contenido de las trazas

```
import numpy as np

dataFile = 'nombre-de-traza.dat' #Nombre de la traza que se
   desea abrir
MAX_VALUE = 32512. #Límite máximo positivo del ancho de banda
resultado = np.fromfile (dataFile, np.dtype ('int16'))/
   MAX_VALUE #Se abre la muestra en formato Int 16 bits (con
   signo)
print(resultado[:150]) #El archivo es demasiado grande para ser
    mostrado por completo por pantalla por lo que se limita su
   tamaño
```

D.2. Script para obtener las trazas en el formato correcto

```
import pyvisa #Libreria para conectarse al osciloscopio
import numpy as np
import struct
import time

# Configuración de parámetros
```

```
tasa_muestreo = 2e6  # Tasa de muestreo: 2 Mega muestras por
   segundo
tiempo_medicion = 2.43  # Duración de la señal: 2.43 segundos
trigger_voltaje = 0.5  # Trigger en subida: 500 mV

# Inicialización de conexión con el osciloscopio
rm = pyvisa.ResourceManager()
scope = rm.open_resource("USB0::0x1AB1::0x0515::MS5A234005436::
   INSTR")

# Configuración de parámetros del osciloscopio
scope.write(":STOP")  # Detener la adquisición de datos
scope.write(":TIMebase:MODE MAIN")  # Usar el modo de tiempo
   principal
scope.write(":TIMebase:MAIN:SCALe {}".format(1 / tasa_muestreo)
   )  # Configurar la escala de tiempo
scope.write(":ACQuire:TYPE HRESolution")  # Usar el modo de
   adquisición de alta resolución
scope.write(":ACQuire:HRESolution:TIME {}".format(
   tiempo_medicion))  # Configurar el tiempo de adquisición
scope.write(":TRIGger:MAIn:MODe EDGE")  # Configurar el modo de
    disparo en flanco
scope.write(":TRIGger:MAIn:SOURce CHANnel1")  # Configurar el
   canal de disparo
scope.write(":TRIGger:MAIn:EDGE:SLOPe POSitive")  # Configurar
   la pendiente de disparo en positivo
scope.write(":TRIGger:MAIn:EDGE:LEVel {}".format(
   trigger_voltaje))  # Configurar el nivel de disparo

# Adquisición de los datos
scope.write(":RUN")  # Iniciar adquisición de datos
```

```
while int(scope.query(":WAVeform:POINts:TRACed?")) != int(
    tasa_muestreo * tiempo_medicion) + 2:
     time.sleep(0.1)

# Obtención de los datos
scope.write(":WAVeform:SOURce CHANnel1")  # Seleccionar canal
    de obtención de datos
scope.write(":WAVeform:FORMat BYTE")  # Formato de datos en
    bytes
scope.write(":WAVeform:BYTeorder LSBFirst")  # Orden de bytes
    en modo LSB (menos significativo primero)
waveform_data = np.array(scope.query_binary_values(":WAVeform:
    DATA?", datatype='h', container=np.array))

# Guardar los datos en un archivo .dat en formato binario
with open("datos_recogidos.dat", "wb") as f:
     f.write(struct.pack('h' * len(waveform_data), *
    waveform_data))

# Cierre de la conexión con el osciloscopio
scope.close()
rm.close()
```

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA