
**Detección de Botnets en Redes IoT Utilizando Técnicas
de Aprendizaje Automático**
**Botnet Detection in IoT Networks Using Machine
Learning Techniques**



Trabajo de Fin de Máster
Curso 2024–2025

Autor

Joel Gómez Santos

Director

Pablo Cerro Cañizares

Máster en Internet de las Cosas

Facultad de Informática

Universidad Complutense de Madrid

Detección de Botnets en Redes IoT
Utilizando Técnicas de Aprendizaje
Automático

Botnet Detection in IoT Networks Using
Machine Learning Techniques

Trabajo de Fin de Máster en Internet de las Cosas
Departamento de Sistemas Informáticos y Computación

Autor
Joel Gómez Santos

Director
Pablo Cerro Cañizares

Convocatoria: *Septiembre* 2025

Calificación: *Nota*

Máster en Internet de las Cosas
Facultad de Informática
Universidad Complutense de Madrid

August 25, 2025

Dedicatoria

*A Pedro Pablo y Marco Antonio, por crear TeXiS
e iluminar nuestro camino*

Agradecimientos

A Guillermo, por el tiempo empleado en hacer estas plantillas. A Adrián, Enrique y Nacho, por sus comentarios para mejorar lo que hicimos. Y a Narciso, a quien no le ha hecho falta el Anillo Único para coordinarnos a todos.

Resumen

Detección de Botnets en Redes IoT Utilizando Técnicas de Aprendizaje Automático

El cada vez más popular fenómeno del Internet de las Cosas y el aumento notable en el número de dispositivos IoT viene vinculada a serios problemas de ciberseguridad de consecuencias muy graves, al pertenecer los dispositivos muchas veces a redes críticas o personales y estar, por sus limitaciones, poco protegidos.

Palabras clave

Máximo 10 palabras clave separadas por comas

Abstract

Botnet Detection in IoT Networks Using Machine Learning Techniques

An abstract in English, half a page long, including the title in English. Below, a list with no more than 10 keywords.

Keywords

10 keywords max., separated by commas.

Contents

1	Introducción	1
1.1	Motivación	1
1.2	Objetivos	2
1.3	Plan de trabajo	3
1.4	Estructura del Documento	4
2	Introduction	5
2.1	Motivation	5
2.2	Objectives	6
2.3	Work Plan	7
2.4	Document Organization	8
3	Marco Teórico	9
3.1	El Internet de las Cosas	9
3.2	Botnets: Una amenaza en auge	11
3.3	Fog, Edge y Cloud Computing	13
3.4	Arquitectura en Microservicios y Docker	14
3.5	Aprendizaje Automático: Detectando lo Indetectable	16
3.5.1	Desbalanceo de Clases	19
4	Estado de la Cuestión	21
4.1	Revisión de soluciones para detección de botnets en IoT	21
4.1.1	Datasets	21
4.1.2	Preprocesamiento, EDA y Selección de Características	23
4.1.3	Técnicas de Detección	24
4.1.4	Conversation analysis	30
4.1.5	Resumen y próximos pasos	32
5	Diseño e Implementación del Sistema	35

5.1	Requisitos	35
5.2	Comparativa de Arquitecturas	37
5.2.1	Distribución o Monolitos	37
5.2.2	Los Microservicios	38
5.3	Arquitectura General del Sistema	40
5.4	Despliegue con Docker Compose	43
5.4.1	Dockerfiles	44
5.5	Microservicios	45
5.5.1	Microservicio de Captura de Tráfico	45
5.5.2	Microservicio de Análisis del Tráfico	50
5.5.3	Microservicio de visualización de Resultados	54
5.5.4	Prometheus	57
5.5.5	Grafana	58
6	Validación Experimental	61
6.1	Selección del Modelo	61
6.1.1	Selección de Datasets	61
6.1.2	Metodología de la Validación	63
6.1.3	Entrenamiento y Validación de los Modelos	64
6.2	Sistema Completo de Detección	67
7	Conclusiones y Trabajo Futuro	69
8	Conclusions and Future Work	71
9	Bibliografía	73

List of figures

3.1	Topología de Botnet en estrella	11
3.2	Paradigma de Computación en la Niebla	13
3.3	Diagrama de Arquitectura en Microservicios	15
5.1	Implementación de la Arquitectura del servicio	37
5.2	Flujo de la Aplicación	42
5.3	Documentación generada automáticamente del endpoint de captura	46
5.4	Documentación Generada Automáticamente para el Servicio de Análisis de Tráfico.	51
5.5	Ejemplo Dashboard de Visualización	56
5.6	Ejemplo Dashboard de Visualización 2	56
5.7	Ejemplo métricas básicas expuestas por el microservicio de análisis	57
5.8	Ejemplo recogida métricas básicas por Prometheus	58
5.9	Añadir Prometheus como Data Source en Grafana	59
5.10	Dashboard simple de métricas creadas ad-hoc	59
6.1	Los contenedores se encuentran operativos	67
6.2	Captura correcta de tráfico	67

List of tables

3.1	Comparativa entre arquitecturas de computación	14
4.1	Resumen crítico de metodologías de selección de características	24
4.2	Resumen de estudios con algoritmos basados en árboles y estrategias ante desbalanceo	26
4.3	Resumen de estudios con algoritmos no basados en árboles y estrategias ante desbalanceo	27
4.4	Resumen de estudios que emplean Deep Packet Inspection para detección de anomalías o botnets	28
4.5	Resumen de estudios que emplean Deep Learning para detección de Botnets en IoT	30
5.1	Resumen de limitaciones de arquitecturas descartadas frente al enfoque basado en microservicios	40
5.2	Resumen del microservicio de Captura de Tráfico	50
5.3	Resumen del microservicio de Análisis de Tráfico	54
6.1	Resumen comparativo de los datasets empleados en el estudio	63

Introducción

“The infant brain only weighs a single pound, but somehow it solves problems that even our biggest, most powerful supercomputers find impossible”

— Nikhil Buduma and Nicholas Locascio en Buduma and Locascio (2017)

En los últimos años, los avances en el ámbito del Internet de las Cosas y la rápida proliferación de dispositivos IoT han traído consigo importantes mejoras en la conectividad, permitiendo la automatización y el monitoreo en contextos que van desde el hogar hasta la industria.

Sin embargo, la propia idiosincrasia del Internet de las Cosas, en su heterogeneidad, lo hace un objetivo ideal para los ataques con redes de bots (*Botnets*). En este contexto, resulta fundamental la implementación de sistemas de detección de este tipo de amenazas.

1.1 Motivación

El sector del Internet de las Cosas ha pasado de pocos nodos prácticamente aislados a redes hiperconectadas que abarcan áreas e infraestructuras críticas como la sanidad. Como se menciona en Snehi et al. (2024) existen estudios que indican que en 2027 habrá más de 30 mil millones de dispositivos IoT en uso, es decir, el Internet de las Cosas formará parte del día a día de todas las personas.

La importancia de la ciberseguridad no ha hecho si no crecer a medida que el IoT se abría paso en nuestras vidas y lo hacía empleando datos personales y dispositivos con acceso a la intimidad de los usuarios. Desde la aparición de este fenómeno no son pocas las Botnets que han sido noticia por sus ataques de consecuencias graves; *Mirai*, *Perisai* o *Matrix* son solo algunas de las *botnets* que han protagonizado recientemente infiltraciones y ataques en redes IoT.

A finales de 2016, Mirai comenzó a lanzar ataques de Denegación de Servicio contra diversas empresas entre las que se incluía la proveedora de nombres de dominio DYN,

viéndose afectado el acceso a internet de innumerables usuarios. Este gran ataque puso de manifiesto la importancia de sólidos sistemas de detección y prevención de estos ataques.

Los avances tecnológicos de la última década no solo mejoran las vidas de los ciudadanos, también permiten que estos ataques sean cada vez más complejos. Los métodos tradicionales para hacerles frente quedan obsoletos y se presenta la necesidad de desarrollar sistemas más inteligentes para poder hacer frente a esta amenaza, el aprendizaje automático.

Es en este contexto donde aparecen nuevas técnicas para combatir estas amenazas, entre las que destaca en los últimos años el aprendizaje automático y, en particular, el aprendizaje profundo.

Los métodos de aprendizaje automático son modelos capaces de detectar patrones y aprender sin que un ser humano le vaya indicando cómo hacerlo a cada paso. Este tipo de algoritmos suponen un avance importante pues pueden entender relaciones complejas que otros métodos más clásicos, como los basados únicamente en tests estadísticos del tráfico, no son capaces de detectar.

En particular, los modelos de aprendizaje profundo, buscan imitar el aprendizaje del cerebro humano y sus neuronas. Estas Redes Neuronales son más potentes que los métodos de aprendizaje automático clásicos pero presentan un mayor consumo de recursos y, muchas veces, corren el riesgo de sufrir un gran sobreajuste. Por ello, no puede decirse con rotundidad que sean unos mejores que otros, siendo muy situacionalmente dependientes (ligereza versus potencia).

Además, en la literatura existente sobre el tema, existen abundantes sistemas probados en entornos simulados en cloud, no correspondiéndose muchas veces con la situación IoT real y presentando mayor latencia y cuellos de botella. Estos ejemplos no exploran las oportunidades de nuevas arquitecturas y despliegues que suponen la capa de niebla (reduciendo la latencia por acercar los cálculos a la fuente) y los microservicios (escalabilidad del modelo, portabilidad...).

Por último, es habitual encontrar que los datasets simulados empleados para detectar redes de Bots, al realizarse en laboratorio, presentan muchas veces una proporción irreal de tráfico malicioso; realizando las inferencias en conjuntos de datos donde los envíos de ataques son muy abundantes y llamativos. Esto puede resultar muy alejado de la realidad, resultando conveniente habitualmente detectar las Botnets antes de que el propio ataque esté ocurriendo como, por ejemplo, en la fase latente (intercomunicativa entre nodos) que presentan las redes P2P.

1.2 Objetivos

El presente trabajo tiene como objetivo principal diseñar un sistema de detección de Botnets en una red IoT desplegable mediante microservicios en la capa Fog de la red. Este sistema deberá emplear el mejor modelo de aprendizaje profundo encontrado mediante una

exhaustiva validación y que debe garantizar el equilibrio entre capacidad de detección y ligereza tanto en consumo de recursos como en tiempo requerido para la inferencia.

Para lograr este propósito existen otros objetivos secundarios que se irán completando como motor del proceso.

Por un lado, estudiar y analizar la idoneidad de diferentes datasets de detección de Botnets en el IoT. Es conveniente tener claro el uso que puede dársele a cada uno de ellos, así como las limitaciones y desventajas que presentan y las técnicas que requieren para solventarlas, por ejemplo, el desbalanceo de clases.

Es muy importante también poder discriminar empleando diversos métodos de selección de características cuáles son aquellas que realmente aportan información relevante. El uso de métodos diferentes, así como la realización de ablaciones para comparar selecciones diversas o, incluso, ninguna, proporciona solidez al sistema y puede garantizar su ligereza manteniendo buenos resultados en su inferencia.

La comparación de diversos modelos de aprendizaje automático clásico y de aprendizaje profundo empleando métricas adecuadas que evalúen no solo su capacidad predictiva si no también su consumo de recursos computacionales y de tiempo es otro objetivo crítico para poder asegurar la correcta elección del modelo final empleado.

También como objetivo fundamental es el diseño con una arquitectura que garantice la escalabilidad y portabilidad. Debe poder ser desplegable en entornos fog manteniendo la robustez tanto de seguridad como de monitorización exigible a cualquier sistema serio.

Por último, debe realizarse una discusión crítica sobre el sistema, haciendo notar no solo sus fortalezas si no también sus puntos de mejora, planteando posibles cambios a futuro que lo hagan más sólido y eficiente.

1.3 Plan de trabajo

Para conseguir los objetivos anteriormente mencionados se empleará el siguiente proceso de trabajo.

Primero, se realizará un estudio de revisión del estado del arte, analizando los modelos y soluciones propuestas por otros autores, así como identificando los vacíos u obstáculos en dichas investigaciones con el fin de evitarlos.

A continuación, se realiza una comparativa de datasets entre aquellos más frecuentemente empleados en el ámbito de la detección de Botnets. Debe determinarse el tipo de contenido que presentan así como posibles inconvenientes existentes en los datos como un fuerte desbalanceo o patrones artificiales triviales. Debe procederse entonces a realizar el preprocesado de todos los datos (escala, limpieza...).

Una vez los datos se encuentran limpios se procederá a realizar la selección de características empleando diversos métodos para ver coincidencias y resultados de cada uno de ellos, buscando determinar cuál es el mejor conjunto de ellas que proporcione una sólida capacidad predictiva sin sobrecargar de información al modelo.

Después, se procederá a probar diversos modelos con diferentes configuraciones de hiperparámetros, validando los mismos con un conjunto de métricas cuidadosamente seleccionadas. Esto permitirá seleccionar el modelo óptimo tanto por potencia como por eficiencia.

A continuación debe diseñarse la solución completa, dividida en 3 microservicios con finalidades claramente diferenciadas primando la independencia y modularidad de los mismos. En esta solución se integrará seguridad y monitorización para garantizar la robustez del sistema.

Por último, se probará el sistema al completo en una simulación y se discutirán los resultados obtenidos, planteando posibles mejoras futuras que lo hagan más preciso y eficiente.

1.4 Estructura del Documento

El documento presenta la siguiente organización:

- **Capítulo 3: Marco Teórico:** En este capítulo se abordan varios fundamentos teóricos que el lector debe comprender para poder lograr un entendimiento completo de la solución aquí planteada.
- **Capítulo 4: Estado del Arte:** Discusión y revisión crítica de la literatura del ámbito abordado. No solo un análisis descriptivo si no que se hace hincapié en debilidades de cada uno de los artículos revisados.
- **Capítulo 5: Diseño e Implementación del Sistema:** Este capítulo presenta el sistema de detección desde un punto de vista de arquitectura y despliegue. Se aborda en el mismo cómo el sistema es desplegado y utilizado por un usuario garantizando la portabilidad e independencia de todas sus componentes.
- **Capítulo 6: Validación Experimental:** En este capítulo se hace un recorrido por todo el proceso de validación del modelo seleccionado, presentando la comparativa de resultados entre modelos que justifican la elección final. Además, se muestra también un ejemplo de uso del sistema en su completitud.
- **Capítulo 7: Conclusiones y Trabajo Futuro:** En el capítulo final se realiza un resumen del trabajo realizado, extrayendo conclusiones y aprendizajes del mismo y presentando posibles líneas de trabajo futuro.

Chapter 2

Introduction

“The infant brain only weighs a single pound, but somehow it solves problems that even our biggest, most powerful supercomputers find impossible”

— Nikhil Buduma and Nicholas Locascio en Buduma and Locascio (2017)

During the last few years, the advancements made in the field of the Internet of Things and the fast proliferation of IoT devices has brought important upgrades in connectivity, enabling the automatization and monitoring in areas as different as industry and personal homes.

However, the idiosyncrasy of the Internet of Things, with its heterogeneity, makes it an ideal objective for Botnet attacks. In this context, it is fundamental to implement systems that aim to detect this kind of threats.

2.1 Motivation

The Internet of Things sector has developed from a few practically isolated nodes to hyperconnected networks that work in critical areas and infrastructures such as public health. As mentioned in Snehi et al. (2024) certain studies affirm that by 2027 as many as 30 billion IoT devices will be operative worldwide, making IoT part of every person's day to day lives.

The importance of cybersecurity has done nothing but increase as the Internet of Things became more popular in our lives and started using personal data and devices that could access user's intimacy such as security cameras. Since the beginning there have been many Botnets that have been responsible for attacks with critical consequences; Mirai, Perisai or Matrix are just a few of them.

In late 2016 Mirai started launching Denegation of Service attacks against different companies. Amongst those companies there was the domain names provider DYN, making it difficult or even impossible for many users to access the Internet. This attack brought

to light the necessity of solid detection and prevention systems to deal with these threats.

Technological advancements in the last decade not only make everyday lives easier but also enable Botnets attacks to gain complexity. Traditional detection and prevention methods became obsolete and the necessity of more intelligent systems arose, machine learning systems.

In this context, new techniques to tackle these threats appear, amongst which Machine Learning and, in particular, Deep Learning, stand out.

Machine Learning algorithms are able to detect patterns and learn without any human being manually imputing their learning process. These kind of algorithms can understand complex relationships that other more traditional methods, such as those based solely on statistical tests, fail to detect.

In particular, deep learning models, aim to simulate the learning process of a human brain and its neurons. These Neural Networks are more powerful than classic machine learning methods but have a bigger resource consumption and have a high risk of overfitting. Therefore, it is impossible to say that either classical methods or deep learning ones are better, being their suitability entirely situational (lightweight vs power).

In addition, in existing literature, there are many detection systems that have been tried in simulated cloud environments, making them quite different from real IoT environments and having a higher latency with, sometimes, bottlenecks. These examples fail to explore the opportunities provided by new architecture and deployment styles such as fog layer deployment (better latency by bringing computing closer to the source of the data) and microservices (scalability, portability...).

Lastly, it is common to find that the datasets that have been used for testing, as they have been done in controlled laboratory environments, often present an unreal proportion of malicious traffic with artificial patterns. This can be really different from real IoT cases where it is convenient to detect the attack before it is happening, such as detection in the dormant stage of Peer to Peer Botnets.

2.2 Objectives

The main objective of this paper is the design of a Botnet detection system in an IoT network deployable in the Fog layer via microservices. This system must use the best machine learning model found amongst a variety of models that have been thoroughly validated and must guarantee the balance between detection capacity and lightness both in resource consumption as well as prediction time.

In order to achieve this goal there are some other secondary objectives that will be achieved in the process.

Firstly, different IoT Botnet detection datasets must be studied. It is important to know how some of them can be used as well as the limitations and disadvantages they present (being also necessary to know which techniques are needed to overcome these

obstacles such as class imbalance).

It is also really important to understand which features provide actual relevant information. This can be done by testing different feature selection methods as well as by making ablations and testing feature selections vs no feature selection. Making these tests provides strength and lightness to the system while maintaining good prediction results.

The comparative between different classical machine learning models and deep learning ones by using suitable metrics that evaluate not only their predictive power but also their resource consumption is a critical objective that ensures a correct selection of the final model.

It is also fundamental the design with an architecture that ensures the scalability and portability of the system. It must be deployable in the fog layer maintaining the robustness both in system security and monitoring that should be demanded in any serious solution.

Lastly, a critical discussion about the system must be made. This discussion should highlight not only the system's strengths but also its growth opportunities that, in the future, could be made.

2.3 Work Plan

In order to achieve the objectives previously mentioned the following work plan will be followed.

Firstly, a state of the art revision will be made. It is important to review the models and solutions presented by other authors while identifying the faulty points or obstacles these other papers have.

Then, a comparative between Botnet detection datasets is made using some of the most frequently used ones. The kind of contents they have as well as possible obstacles have to be determined, such as class imbalance or artificial data patterns. Once that is clear, data preprocessing must be made (cleaning, scalation...).

Once the data is clean, several different feature selection methods are used. By comparing them a suitable feature set can be established that maintains the model's predictive capacity without overloading the model with information that is not needed.

Then, different models with different hyperparameter configurations are tested, validating them with a suitable set of metrics that have been carefully selected. This enables the selection of the best model not only by prediction power but also by efficiency.

Lastly, the complete system must be designed. This solution will be divided into 3 different microservices that handle completely independent parts of the process, ensuring their modularity and portability into different systems. In this solution security and monitoring must be integrated.

Lastly, the complete system has to be tested in a simulation and the resulting predictions have to be discussed, laying out possible future improvements that make the model more precise and efficient.

2.4 Document Organization

This document presents the following chapter organization.

- **Chapter 3: Theoretical Framework:** In this chapter several theoretical concepts are briefly explained. These ideas are fundamental in order to achieve a complete understanding of the proposed solution.
- **Chapter 4: State of the Art:** Discussion and critical review of the literature. It is not only a descriptive analysis but also the weaknesses of the analysed papers are highlighted.
- **Chapter 5: System Design and Implementation:** In this chapter the botnet detection system is presented from an architecture and deployment point of view. It is explained how the system is deployed and used by a user ensuring portability and independence of each individual component.
- **Chapter 6: Experimental Validation:** This chapter narrates the complete model selection process, showing the results obtained by each model and thus justifying the final model selection. It also shows how the complete system would work.
- **Chapter 7: Conclusions and Future Work:** In the final chapter a summary of the work is done, extracting conclusions and lessons and laying out possible future work lines.

Marco Teórico

Es fundamental establecer un marco sólido que garantice la comprensión del contexto técnico del problema afrontado, además de resultar la base teórica de las decisiones tomadas en la realización del presente trabajo.

Entre los conceptos abordados en esta sección se encuentran los dos elementos principales de esta tarea, el Internet de las Cosas y las Botnets. Se expondrá la creciente expansión del IoT en las aplicaciones tanto industriales como domésticas y como este, por su propia idiosincrasia, presenta vulnerabilidades y desafíos de gran interés en el ámbito de la informática y, en particular, de la ciberseguridad. Se procederá a dar una descripción de algunos tipos de ataques de Botnets para entender en profundidad cómo actúan estas y cómo podrían enfocarse las contramedidas ante estas amenazas.

A continuación se darán detalles técnicos relativos a la computación en la niebla (*Fog Computing*) y la motivación detrás de la decisión de esta como la mejor opción para el despliegue del servicio. Ahondando en la arquitectura interna de este sistema se sentarán las bases necesarias para comprender una arquitectura en microservicios y las ventajas detrás de sistemas de containerización como Docker.

Por último, se expondrán los diferentes modelos de Machine Learning empleados en las pruebas del servicio de análisis de tráfico. Estas especificaciones permiten entender las ventajas y limitaciones de cada uno de ellos y cómo contribuyen a la clasificación del tráfico como malicioso o benigno.

Estos fundamentos permitirán, por tanto, entender completamente la solución propuesta en los siguientes capítulos.

3.1 El Internet de las Cosas

Se ha vivido recientemente un aumento en la adopción del Internet de las Cosas por diferentes organizaciones y particulares. El IoT trae consigo enormes ventajas y ofrece un mundo de posibilidades para sus usuarios, sin embargo, conviene prestar especial atención a sus vulnerabilidades e implementar soluciones de forma acorde.

El Internet de las Cosas es la posibilidad de conectar a Internet elementos comunes inicialmente no concebidos para ello (como pueden ser puertas, medios de transporte o electrodomésticos). Esto permite generar entornos totalmente conectados que automatizan múltiples procesos frecuentes, facilitando tareas cotidianas para sus usuarios.

La arquitectura del IoT posee 4 capas. La primera de ellas la conforman los sensores y actuadores encargados de recopilar información del entorno. Suelen ser dispositivos de tamaño y recursos reducidos, haciendo de ellos un objetivo prioritario para los atacantes en estos entornos pues deben equilibrar la implementación de seguridad con su funcionalidad principal dentro de la limitación de recursos que sufren.

La siguiente capa es la capa de red, la cual provee transporte de los datos entre los elementos que conforman la red.

La capa de procesamiento de datos emplea técnicas como el aprendizaje automático para limpiar, ordenar y formatear los datos para su análisis posterior o para facilitar la toma de decisiones en base a los mismos.

Por último, se encuentra la capa de aplicación. Esta capa supone la capa de interacción con el usuario, permitiendo, mediante el uso de interfaces sencillas y explicativas, que la persona usuaria pueda visualizar los datos extraídos y controlar los dispositivos de la capa de recopilación de datos.

Desde un punto de vista de seguridad la cantidad limitada de recursos previamente mencionada no es el único desafío que enfrentan las redes IoT. El entorno del Internet de las Cosas es extremadamente heterogéneo, tanto a nivel de software (protocolos, sistemas operativos...) como a nivel de hardware. Esta diversidad dificulta la adopción de técnicas comunes de seguridad y facilita que un atacante pueda descubrir una vulnerabilidad en alguno de los elementos que conforman la red. Además, como se verá en el apartado siguiente, las contraseñas por defecto con las que se comercializan muchos dispositivos IoT no son modificadas por el usuario, permitiendo su corrupción mediante ataques de Fuerza Bruta.

Estas vulnerabilidades representan una amenaza importante pues pueden dar acceso a la red a personas ajenas a la misma. Esta infiltración tendría consecuencias de enorme importancia si se realizara contra infraestructuras proveedoras de servicios en, por ejemplo, un sistema de riego automático de una ciudad o una red de sensores en Smart Farming. Sin embargo, no conviene infravalorar el impacto que esta pueda tener también para un ciudadano común, pues un ataque contra un sistema de Smart Home podría, por ejemplo, otorgar acceso a un tercero a imágenes privadas del hogar o permitir que este controle elementos del edificio como sistemas de aire acondicionado, pestillos o luces.

Como se discute en Amoo et al. (2024) resulta fundamental comprender los retos en ciberseguridad a los que se enfrenta el IoT y cómo afrontarlos.

3.2 Botnets: Una amenaza en auge

Considérese una red de dispositivos IoT. Se denomina *botnet* a un subconjunto de dichos dispositivos que se encuentran controlados remotamente por un usuario externo denominado *Botmaster*.

Estos aparatos, al pertenecer a la red IoT original, son empleados por su nuevo controlador para realizar ataques o acciones maliciosas en perjuicio de la red y/o en beneficio del *Botmaster*.

Dentro de la estructura de una *botnet* centralizada destaca también el Servidor de Comando y Control (C&C). Este es un dispositivo externo a la propia red (es decir, no es un dispositivo infectado si no uno en posesión del atacante) que se encarga de organizar la red de bots y de enviar las órdenes de ataque para su realización. Este tipo de servidores dan lugar a topologías como la Botnet en estrella (donde un Servidor C&C envía comandos a todos los bots) o la Botnet Multi-Server (en la que un conjunto de servidores C&C intercomunicados se encargan del envío de instrucciones).

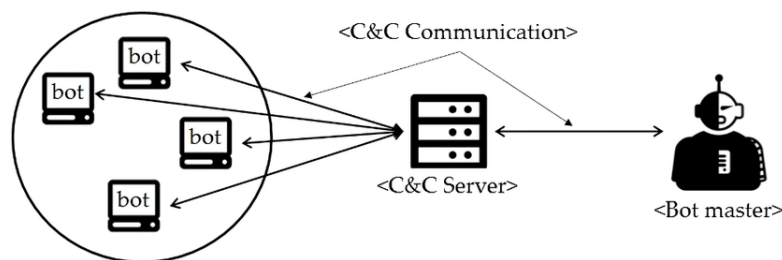


Figure 3.1: Topología de Botnet en estrella

Sin embargo, existe una clara desventaja ante este modelo de redes de bots que ha derivado en un uso creciente de botnets descentralizadas en los últimos años (Botnets Peer-to-Peer) y es el *punto de fallo*. Es decir, al existir un (o un conjunto reducido de) servidores de control la trazabilidad del ataque puede remontarse a dicho servidor y neutralizar la amenaza de una forma que, pese a ser compleja, es más sencilla que en una topología descentralizada.

Es entonces cuando surgen las Botnets Peer-to-Peer o Botnets descentralizadas. Esta concepción distribuida de la red permite al Botmaster enviar la instrucción a un único dispositivo infectado y son los propios bots quienes, mediante comunicación entre ellos, distribuyen los comandos a lo largo de toda la red. La inexistencia de un servidor de control y la capacidad para insertar las órdenes de ataque entre comunicación legítima de los dispositivos hacen de estas redes mucho más difíciles de detectar y, por tanto, mucho más resistentes a su desmantelamiento que aquellas con un Servidor C&C.

Se presentarán a continuación algunos de los ataques más habituales de las Botnets en el Iot, especificando la fase de vida de la Botnet en la que se producen.

- **Reconocimiento:** En esta etapa la Botnet se encuentra en proceso de escaneo de la red con el objetivo de detectar dispositivos vulnerables para su posterior infección. Como

parte de este proceso destaca el **Ataque de Escaneo de Puertos/Port Scanning Attack**; en un escaneo de este tipo el atacante identifica puertos abiertos como posible indicio de vulnerabilidades explotables y son, por tanto, candidatos a facilitar el acceso y, en última instancia, el control del dispositivo. Esta técnica puede tomar distintas formas, entre ellas destacando el envío de flags de tipo SYN y, al recibir el SYN-ACK se descubre que el puerto está abierto, o el envío de pings (una técnica mucho más sencilla).

- **Infección:** Una vez detectadas las vulnerabilidades se procede a la infección del dispositivo. Es habitual el empleo de **técnicas de fuerza bruta** para la obtención de credenciales (siendo la no modificación de las credenciales predeterminadas de fábrica una vulnerabilidad muy habitual en el IoT) o el **File Downloading Attack** en el que se busca lograr que el dispositivo descargue un fichero malicioso.
- **Explotación:** Una vez la red de bots ya se encuentra establecida surgen otros ataques. El **ataque de denegación de servicio (DOS)** busca imposibilitar a la víctima su uso del servicio (por ejemplo, su conexión a Internet). Para ello se emplean diferentes técnicas (como se verá más adelante con el ejemplo de la Red Mirai) siendo una de las más habituales consiste en sobrecargar el servidor de forma que no pueda gestionar la cantidad tan elevada de peticiones que recibe y no pueda dar servicio a las peticiones legítimas. En el contexto del Internet de las Cosas también se observa el **Cryptojacking o Minado de Criptomonedas**; este tipo de ataque, de gran interés desde el auge de Bitcoin en la última década, consiste en emplear los dispositivos de la víctima para minar criptomonedas sin su conocimiento, resultando en un aumento no autorizado de los costes computacionales y de recursos, viéndose perjudicado el rendimiento del dispositivo y generando, además, posibles perjuicios económicos.

Estos ataques pueden comprometer gravemente la seguridad de una red IoT resultando en exposición de información sensible o inutilización de servicios. Un ejemplo sería un sistema domótico de seguridad en un hogar víctima de una Botnet; si un atacante encontrara una vulnerabilidad en una red de cámaras de una vivienda tendría acceso sin conocimiento de la víctima a imágenes en directo del hogar, además de otros posibles dispositivos de la red como un sensor que permitiera bloquear o desbloquear puertas en la casa.

Para una mayor comprensión del problema que suponen las botnets en el Internet de las Cosas se presenta a continuación el ejemplo más emblemático de este tipo de ataques y su impacto. La botnet **Mirai**, creada en sus inicios por un grupo de 3 jóvenes estadounidense, fue utilizada por diversos hackers después de que su código fuente fuera publicado en varios foros de Internet. Esta Botnet buscaba realizar **Ataques de Denegación de Servicio** y, para ello, se valía de **Ataques de Fuerza Bruta** para adivinar claves de dispositivos IoT que, muchas veces, eran aquellas que el fabricante implementó por defecto con poco grado de complejidad. Una vez se había logrado apoderar del dispositivo usaba su nueva red de bots para saturar los servidores mediante la generación masiva de tráfico. A finales de 2016 Mirai fue la causante de diversos ataques que inutilizaron algunas grandes compañías tecnológicas como la francesa OVH o la estadounidense DynDNS. El ataque de Mirai puso de manifiesto la vulnerabilidad de las redes IoT y los peligros que este nuevo paradigma

traía consigo si se descuidaba la seguridad.

3.3 Fog, Edge y Cloud Computing

Si el lector ahonda en artículos relativos a la detección de botnets en Redes IoT, como Snehi et al. (2024), podrá observar que el concepto de Fog Computing cobra cada vez más fuerza en el ámbito del IoT y, en especial, en tareas de computación en redes IoT. Por ello, conviene abordar este nuevo paradigma y entender las ventajas y las desventajas que este ofrece como posible entorno de despliegue de la solución de análisis de tráfico.

La Computación en la Niebla (*Fog Computing*) busca ofrecer capacidad de cómputo y almacenamiento en un nivel intermedio entre los dispositivos frontera y la computación en la nube tradicional. Como se discute en Bonomi et al. (2012) esta nueva capa ofrece diversas ventajas entre las que destacan la disminución en la latencia, la descentralización y la cercanía al origen del dato. Esta concepción local requiere el despliegue de nodos como routers o microservidores a poca distancia de los elementos de la capa de recopilación, siendo muy claro como esto contribuye a reducir la latencia respecto a la computación en la nube. Además, desde un punto de vista de seguridad y robustez, este paradigma computacional descentralizado destaca en dos aspectos. Por un lado, la no necesidad de enviar datos a través de la nube y la capacidad para procesar datos sin salir del entorno local, pudiendo mantener en el mismo ciertos datos sensibles, evita posibles vulnerabilidades o transporte de datos que requerirían especial esfuerzo de seguridad para evitar su filtración ante ataques externos. Por otro lado, se garantiza la robustez del sistema al ser resiliente ante caídas de red y conexión con el servicio en la nube. Si la red pierde su capacidad de interactuar con el servicio centralizado todo el sistema estaría comprometido, mientras que la existencia de varios microservidores locales con capacidad de almacenamiento y cómputo permitiría que este siguiera en funcionamiento mientras se reestablece la conexión.

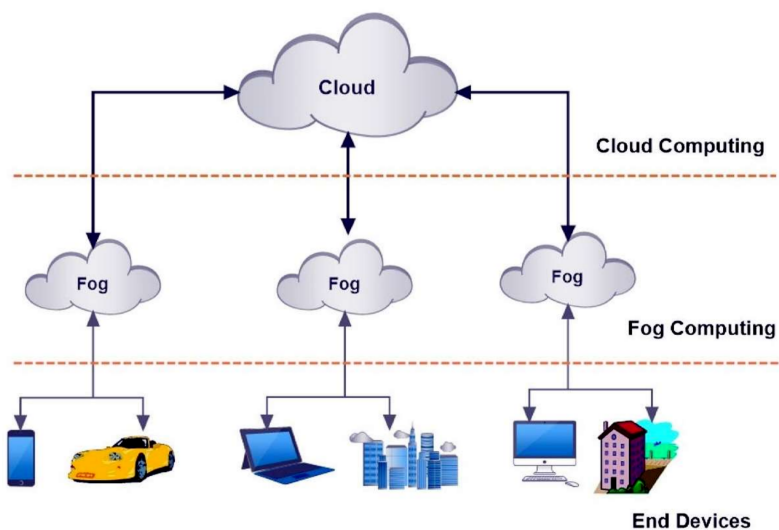


Figure 3.2: Paradigma de Computación en la Niebla

Conviene no confundir la Computación en la Niebla con otro modelo de también reciente popularidad, la Computación en la Frontera. En este caso, la computación se realiza en algunos de los propios dispositivos IoT, por ejemplo en ciertos sensores de la red. Este modelo lleva los cómputos lo más cerca posible de la capa de más bajo nivel de la red reduciendo la latencia lo máximo posible, como se presenta en Sarkar et al. (2025). No obstante, y aunque esta eficiencia en tiempo es fundamental en ciertos entornos e infraestructuras críticas del IoT como ciudades inteligentes o Health IoT, el análisis de tráfico malicioso en la red no requiere de dicha inmediatez. Además, realizar los propios cálculos en los dispositivos demanda una inversión económica y de recursos en hacer de estos nodos, muchas veces muy simples y con poca capacidad de procesamiento, capaces de soportar las computaciones requeridas. En el caso de modelos posiblemente complejos como una red neuronal seguramente no merece la pena la inversión por el beneficio.

Paradigma	Latencia	Consumo de Ancho de Banda	Escalabilidad	Control
Cloud Computing	Alta	Alta	Alta	Centralizado
Fog Computing	Baja	Baja	Media	Local
Edge Computing	Muy baja	Muy baja	Baja	Local

Table 3.1: Comparativa entre arquitecturas de computación

En el problema abordado en el presente trabajo la latencia y el consumo de ancho de banda no son elementos críticos a tener en cuenta, por tanto, y como se observa en el resumen presentado en 3.1, la Computación en la Niebla ofrece un equilibrio entre tiempo, escalabilidad y control que hace de ella una solución de especial interés para el despliegue del servicio de análisis de datos.

3.4 Arquitectura en Microservicios y Docker

Debido a la decisión de adoptar un modelo de arquitectura en microservicios, en lugar de una arquitectura monolítica, resulta importante comprender las características de este modelo de despliegue.

El principio fundamental sobre el que opera este tipo de arquitectura es el desacople. Un enfoque en microservicios parte de la premisa de que una aplicación o servicio puede dividirse en componentes funcionales más pequeños, cada uno encargado de una responsabilidad específica y delimitada sin que sus funciones interfieran entre sí. En este esquema la aplicación es la unión de los microservicios, que trabajan de forma independiente pero comunicada para lograr un objetivo común.

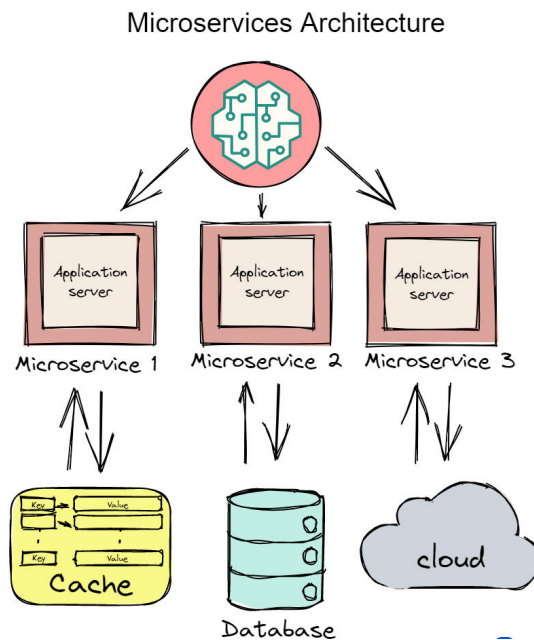


Figure 3.3: Diagrama de Arquitectura en Microservicios

A diferencia de la clásica arquitectura monolítica, donde la aplicación se concibe como una única unidad o servicio indivisible, los microservicios proporcionan modularidad, escalabilidad y facilidades en el despliegue y el mantenimiento como se discute en Richards and Ford (2020).

Cada microservicio tiene una funcionalidad clara, esto permite diferenciar claramente en qué parte del proceso se encuentran los cuellos de botella o las dificultades o deficiencias. Esta independencia permite, además, realizar el redespliegue únicamente de la tarea problemática y no de todo el servicio en su conjunto. Además, pueden escalarse los recursos de los microservicios que lo requieran.

Estas ventajas y su adecuación al sistema implementado en el presente trabajo, como se verá más adelante, justifican su adopción en la solución final.

Como herramienta para el despliegue de la arquitectura en microservicios se empleará Docker. Docker es un software de código abierto que permite desplegar aplicaciones de forma aislada en *contenedores*. Cada uno de estos contenedores agrupan todos los elementos que comprenden el dominio de dicha aplicación.

Para lograr esto se definen imágenes, plantillas que contienen el código y sus dependencias, y posteriormente se instancian creando los contenedores. Estas plantillas hacen de la aplicación un servicio portable y reproducible, pues cualquier sistema que posea la plantilla podrá instanciar exactamente el mismo contenedor. Docker permite también el empleo de volúmenes, almacenamientos persistentes de datos compartibles entre contenedores, que facilitan enormemente la integración de los microservicios tanto entre sí como con el host. Además, docker permite el despliegue ágil de múltiples contenedores en un solo sistema anfitrión, lo cual unido a sus características de manejo de recursos y escalabilidad lo hacen

una opción sólida y eficiente como herramienta de despliegue.

Se empleará la herramienta de orquestación Docker Compose para gestionar el despliegue y el ciclo de vida de múltiples microservicios. Este software permite agrupar dicho despliegue en un solo fichero yaml y gestionar de forma sencilla tanto los volúmenes compartidos como la red.

Estas herramientas y conceptos permiten entender el diseño del sistema implementado y el por qué se ha decidido adoptar una topología distribuida en detrimento de la arquitectura monolítica.

3.5 Aprendizaje Automático: Detectando lo Indetectable

De la mano del desarrollo tecnológico han aparecido máquinas cada vez más inteligentes. Los avances en los últimos años del *Machine Learning* permiten a los dispositivos aprender de los datos para tomar decisiones sobre datos que nunca antes han visto. Estas técnicas permiten automatizar tareas como la que se aborda en el presente trabajo, la detección de tráfico de bots en una red de IoT.

En esta sección se explicarán brevemente algunas de las técnicas más populares para lograr este objetivo, las cuales se mencionarán en secciones posteriores cuando se aborde el estado del arte, centrándose en la conexión de estas con el problema particular de la clasificación de tráfico.

La primera parte del proceso consistirá en determinar las características (*features*) más apropiadas para entrenar a los modelos. La elección de características es un paso fundamental pues se reciben múltiples métricas en la captura de tráfico y una sobrealimentación de estas a los modelos de aprendizaje automático puede hacer crecer drásticamente el tiempo de entrenamiento y predicción de los mismos sin aportar información nueva relevante. Se emplearán técnicas de selección de características como ANOVA o las importancias de modelos como el Random Forest. Además, un procedimiento recomendado es normalizar las variables numéricas y convertir las categóricas en identificadores numéricos, esto es necesario para el empleo de algunos de los modelos.

Se emplearán modelos de Aprendizaje Automático, siendo algunos de ellos modelos de Aprendizaje Profundo. En el Aprendizaje automático se diseñan sistemas capaces de aprender patrones y relaciones y tomar decisiones en base a los mismos sin haberles sido explícitamente definido cómo hacerlo. En el caso particular del Deep Learning el modelo trata de emular el razonamiento de un cerebro humano, implementando capas de neuronas que le permiten identificar, entender y emplear relaciones y procesos de mayor complejidad cuyo razonamiento no sabríamos o nos resultaría muy difícil programar como se argumenta en Buduma and Locascio (2017). Los modelos de Machine Learning, en definitiva, ofrecen una solución efectiva para la detección de tráfico Botnet en una red IoT al ser capaces de detectar los patrones subyacentes del tráfico malicioso.

Random Forest y Extra Trees

Entre los modelos cuyos resultados se analizarán el primero es **Random Forest**. Este modelo, un popular modelo de ensemble, consiste en la combinación de muchos *Árboles de Decisión* y selección por mayoría. En este tipo de árboles cada nodo plantea una posible división del dataset en base a una característica (por ejemplo, en el caso aquí abordado, flujos conversacionales con más de x bytes por paquete o con menos) mientras que los nodos hoja del árbol determinan la clasificación final de dicho elemento del conjunto. El bosque de árboles proporciona estabilidad y robustez al modelo, pues basa su clasificación en la clasificación que hayan realizado la mayoría de sus miembros.

Además del **Random Forest** se hará uso de otro modelo de ensemble similar, **Extra Trees**. La diferencia entre ellos, siendo ambos combinaciones de árboles de decisión, radica en el sistema empleado para el entrenamiento de dichos árboles. **Random Forest** emplea *bagging* para seleccionar subconjuntos aleatorios para entrenar los árboles que lo componen, asegurando diversidad entre los mismos. Sin embargo, **Extra Trees** entrena los árboles de decisión con todo el dataset de entrenamiento que se le provee. Por tanto, para mantener la diversidad en las predicciones de sus miembros, realiza de forma aleatoria las divisiones de sus nodos. El empleo de la totalidad del dataset aumenta su sesgo pero reduce su varianza, es decir, disminuye el sobreajuste del entrenamiento, mientras que la aleatoriedad agiliza el proceso de entrenamiento. En el problema abordado en el presente trabajo, como se verá en secciones subsecuentes, existe un alto problema debido al desbalanceo de las clases en el dataset, haciendo del sobreajuste un riesgo a tener en cuenta.

Gradient Boosting, XGBoost y LightGBM

Además de los algoritmos de ensemble recién mencionados se realizarán pruebas con modelos que emplean potenciación del gradiente (**Gradient Boosting**). El procedimiento base sobre el que se fundamentan este tipo de algoritmos es el *boosting*, es decir, el entrenamiento secuencial de modelos débiles, como árboles de decisión, tratando de mejorar en cada uno el rendimiento de su predecesor. En este caso esto se realiza mediante la disminución de la función de pérdida empleando el descenso del gradiente, es decir, tomar la dirección opuesta al gradiente como método para avanzar de un paso del boosting al siguiente.

Como caso particular de este tipo de modelos se implementarán **XGBoost** y **LightGBM**. Por tanto, se procederá a aclarar las particularidades de estos y qué ventajas ofrecen para el problema de detección de tráfico malicioso. **Extreme Gradient Boosting** incorpora ciertas ventajas computacionales y estadísticas entre las que destacan la regularización L1 y L2 para evitar overfitting, post-pruning (eliminar ramas que no aportaban información relevante) y paralelización permitiéndole procesar de forma mucho más sencilla datasets de gran tamaño. Además, este algoritmo maneja de forma nativa valores faltantes obteniendo similares resultados a los casos en los que se imputan los valores faltantes empleando diferentes técnicas, como se discute en Ergul Aydin and Kamisli Ozturk

(2021), lo cual es habitual en datos relativos a tráfico IoT. **LightGBM** es un método desarrollado por Microsoft que ofrece soporte a variables categóricas sin codificar (como podría ser el protocolo empleado), usa leaf-wise growth en lugar de level-wise (es decir, divide el nodo con mayor ganancia independientemente de su profundidad en lugar de dividir todos los nodos del mismo nivel antes de proceder con el siguiente, esto aumenta el riesgo de sobreajuste) y optimiza el tiempo de entrenamiento mediante el empleo de histogramas. Por tanto, ambas opciones presentan ventajas muy relevantes en el desafío que se aborda, algunas a nivel de optimización de recursos y tiempo y otras enfocadas en el sobreajuste.

Dando por concluido el conjunto de modelos de aprendizaje automático no profundo relevantes se procede a presentar de forma sencilla y breve las redes neuronales empleadas.

Perceptrón Multicapa

En primer lugar, se aborda el **perceptrón multicapa (MLP)**. El MLP es una red neuronal cuyas capas de neuronas están completamente conectadas y poseen una función de activación no lineal, haciendo de ella un modelo especialmente fuerte en casos donde los datos no son linealmente separables. Este modelo supone un punto de partida óptimo antes del empleo de redes más complejas, su capacidad para procesar datos tabulares (como los csv) hace que se adapte muy bien a la situación contemplada.

Autoencoders

Se ha hecho uso también de **Autoencoders**. Estos modelos, de aprendizaje no-supervisado, se emplearán para tratar de reconstruir tráfico normal y detectar anomalías. Es decir, se entrena empleando tráfico únicamente no malicioso y luego, al recibir el tráfico real completo, la red neuronal es capaz de detectar desvíos en su intento de reconstrucción, siendo estos desvíos síntoma de posibles botnets.

GRU y Transformadores

Además, debido a que los datos de tráfico son datos ordenados temporalmente, se ha considerado relevante el empleo de redes neuronales secuenciales, en particular redes de tipo **GRU**. Este tipo de modelo es capaz de capturar dependencias temporales relevantes, descartando aquellas que no lo son, lo cual resulta muy efectivo si se conciben los envíos de paquetes como flujos temporales de conversaciones entre dispositivos en lugar de elementos por separado.

Por último, se han empleado **transformadores**. Estos modelos detectan relaciones globales en secuencias de flujos conversacionales empleando atención (es decir, asignación de importancias diferentes a características diferentes de los datos). Su coste computacional es, sin embargo, elevado, lo cual resulta un factor determinante en la adopción de dicho modelo como el más óptimo.

Los modelos introducidos en esta sección son la base de las pruebas realizadas y sus

resultados motivan la decisión de implementar únicamente uno de ellos en la solución final.

3.5.1 Desbalanceo de Clases

Este apartado final pretende abordar el desbalanceo, un problema que aparece en ciertos problemas de clasificación. Se presentarán varias formas de abordarlo, incluyendo SMOTE que ha sido la finalmente seleccionada.

Primero, se busca entender cuál es la dificultad que se presenta. El desbalanceo en una tarea de clasificación surge cuando una o varias de las clases de los datos son muy mayoritarias. En el caso de un problema de análisis de tráfico de una botnet esto se da porque el envío de paquetes de la botnet es mucho más reducido, por lo general, que el de las aplicaciones legítimas, que suelen presentar un flujo mucho más continuo con mucho tráfico de fondo. El desbalanceo afecta a muchos modelos y requiere un trabajo muy riguroso de elección de métricas de evaluación y de hiperparámetros para evitar, por ejemplo, que el modelo decida minimizar la pérdida prediciendo siempre la clase mayoritaria, esto traería una alta accuracy pero no significaría que el modelo esté separando de forma correcta las clases.

Las técnicas para resolver esto pueden agruparse en 2 categorías: oversampling y undersampling. La primera de estas consiste en aumentar de forma artificial los registros de la o las clases minoritarias para igualar su número, mientras que la segunda busca eliminar mediante algún criterio registros de las clases mayoritarias.

Se abordará primero el undersampling. La forma más sencilla de realizarlo es eliminando datos aleatorios pertenecientes a la clase que se busca disminuir. Una técnica diferente son los Tomen Links, que buscan pares de datos muy próximos entre sí pero que pertenezcan a clases diferentes. Eliminando estos registros se facilita la labor de los modelos sin borrar información relevante.

En el caso del oversampling este puede hacerse también de forma aleatoria mediante la duplicación de registros de la clase minoritaria escogidos al azar. Este modo, aunque efectivo para resolver el objetivo principal, realmente tiene otros problemas subyacentes. La duplicación de registros existentes no aporta información nueva y, aunque el modelo puede entrenarse con más datos, no le han ayudado a aprender información nueva. El otro método es SMOTE (Synthetic Minority Oversampling Technique), que aporta varias ventajas respecto al anterior. En este caso, no se duplican registros existentes si no que se seleccionan puntos de la clase minoritaria y se escogen sus k vecinos, creando registros artificiales a medio camino entre ambos. Esta opción mejora la anterior al aportar cierta información nueva aunque es la única de todas las presentadas que crea datos ficticios.

Estado de la Cuestión

Resulta fundamental analizar el estado actual de las soluciones existentes frente a la creciente amenaza que suponen las botnets en los entornos del Internet de las Cosas. Esta revisión permite identificar las técnicas actualmente empleadas, así como los logros alcanzados y las posibles limitaciones aun existentes, especialmente en lo referente al despliegue y al uso de modelos más avanzados de aprendizaje automático.

El objetivo de esta sección es aportar contexto al trabajo desarrollado dentro del panorama de investigaciones recientes en el ámbito, identificando las aproximaciones relevantes y las posibilidades aún no abordadas. Esta base comparativa servirá como punto de partida para argumentar la necesidad de la solución planteada posteriormente. En particular, el artículo *Foggier skies, clearer clouds*, constituye el referente principal sobre el que se articula esta propuesta, siendo complementado y ampliado con nuevos modelos, arquitectura modular y una comparativa en detalle de los resultados de diferentes modelos.

4.1 Revisión de soluciones para detección de botnets en IoT

Se abordará el estado del arte siguiendo el proceso natural de una solución de detección de botnets en redes IoT desde la selección de datasets hasta la comparativa de resultados de evaluación de modelos, haciendo hincapié tanto en las decisiones tomadas por otros autores como en las áreas abiertas que se dejan sin investigar.

Por ejemplo, el recientemente mencionado artículo, Sneh et al. (2024), hace uso de algoritmos XGBoost y DRF (Distributed Random Forest) para crear el metamodelo final. Estos modelos ofrecen un resultado robusto, como se verá más adelante, pero existen técnicas de aprendizaje automático más potentes, especialmente las relacionadas con aprendizaje profundo, cuya actuación en estas situaciones es mejor.

4.1.1 Datasets

Existe cierto consenso en el campo de la detección de botnets mediante aprendizaje automático respecto a la importancia de emplear datasets directamente tomados de entornos

IoT reales o lo más parecido posible a ellos pues de esta forma se consiguen datos abundantes y de gran calidad, impactando directamente en los resultados del modelo, como se menciona en Nazir et al. (2023). La diversidad, considerada en el ya citado artículo, también tiene una importancia crítica en la consecución de objetivos. Existen diversos tipos de ataque que una botnet puede realizar, habiendo presentado ya algunos de ellos en el capítulo previo. Por tanto, si quiere realizarse un modelo lo más completo posible (entendiendo completitud como la capacidad para detectar una botnet presente en la red independientemente del tipo de ataque que esta realice) es importante entrenar el modelo con datasets que incluyan ataques diferentes. No obstante, puede seleccionarse un solo tipo de ataque y especializar el servicio en su detección, como es el caso de Snehi et al. (2024) y los ataques de denegación de servicio.

En la mayoría de los casos se realiza una selección de 4 o 5 datasets relevantes y ampliamente aceptados en el ámbito de la detección de botnets en redes IoT. Entre estos conjuntos de datos destacan los siguientes:

- **Aposemat IoT-23**, un extenso dataset obtenido de dispositivos IoT reales en la Universidad Técnica Checa que incluye diversos tipos de ataque en ficheros log obtenidos mediante Bro (actualmente Zeek).
- El dataset **Bot-Iot** de la Universidad de Nueva Gales del Sur (UNSW). Similar al anterior en la diversidad de registros de ataques que proporciona pero su formato es CSV y no logs.
- **N-BaIoT**, creado por investigadores de la Ben-Gurion University, incluye tráfico de red de dispositivos IoT infectados por malware como Mirai y BASHLITE. Está disponible como ficheros CSV.
- **TON_IoT** es un dataset creado también por la Universidad de Nueva Gales del Sur con tráfico de diversos dispositivos IoT como puertas y sistemas de iluminación automatizados.

Pese a la existencia y el uso de muchos otros datasets se ha decidido resaltar estos 4 al ser aquellos sobre los que se trabajará en el presente trabajo. Para realizar evaluaciones completas y robustas de los modelos es conveniente realizar, por ejemplo, mezcla de datasets para entrenar y predecir sobre ellos o realizar predicciones en un dataset con un modelo entrenado con otro conjunto de datos. Esto busca abordar una de las principales dificultades en el IoT que es su heterogeneidad, cuanto mayor sea la variedad de situaciones de prueba en el modelo más alta será su fidelidad.

Muchos de los estudios en el ámbito realizan pruebas sobre un único dataset, como es el caso de Javeed et al. (2023), que, generalmente, son simulaciones de entornos reales de IoT. Sin embargo, el uso de un único conjunto de datos hace de los métodos de detección unas herramientas cuya robustez no se encuentra del todo demostrada. En Snehi et al. (2024) se busca ampliar la solidez de las predicciones mediante la combinación de datasets en un solo conjunto más grande de datos, esto, aunque efectivo para el aprendizaje, puede resultar en una distorsión de la realidad del IoT pues modifica artificialmente el conjunto

de entrenamiento y test, pudiendo hacer que se eviten problemas como el desbalanceo de clases que sí ocurre en un contexto real.

4.1.2 Preprocesamiento, EDA y Selección de Características

El siguiente paso natural sería el preprocesamiento de los datos y la realización de un análisis exploratorio. Sin embargo, este aspecto es muchas veces obviado por completo o infravalorado excepto la extracción de la importancia de las características (como ocurre en Snehi et al. (2024)). En este trabajo se ha decidido dar una mayor relevancia y profundidad a este apartado, realizando, además de un análisis de correlaciones y relevancias, otro tipo de estudios como detección y visualización de outliers o análisis de series temporales.

En *Foggier skies, clearer clouds* se realiza ingeniería de características en base a la correlación de Pearson entre las mismas, eliminando así aquellas que no aportan información nueva relevante, y se seleccionan las de mayor significancia empleando ANOVA. Se ha decidido hacer uso de esta técnica también, añadiendo la información obtenida en un Feature Importances de un Random Forest para proporcionar mayor respaldo a la decisión final. Además, se observarán las características que presentan mayores anomalías empleando medidas como el rango intercuartílico y la distribución de frecuencias.

El uso del Coeficiente de Pearson + ANOVA presenta ciertas limitaciones. Por ejemplo, ANOVA es una técnica que asume normalidad en los datos, no obstante, muchas métricas encontradas en los paquetes (como el número de bytes) pueden seguir distribuciones de colas pesadas (es decir, colas no acotadas exponencialmente y, por tanto, más proclives a valores atípicos). Esta premisa lo hace menos apropiado en ciertos casos que otros tests similares que no parten de la misma suposición como Kruskal-Wallis.

Otro inconveniente es que ANOVA presenta su mayor potencia cuando la pérdida es cuadrática. Esto no se adapta del todo al contexto de ciberseguridad en el que se sitúa el presente trabajo pues, en un contexto real, la detección errónea de una Botnet suele ser más cercana a "no acertar es un error absoluto", considerando cualquier Botnet que pase indetectada como una vulnerabilidad lo suficientemente grave como para cuestionar todo el modelo.

Por último, es importante mencionar que el coeficiente de Pearson solo captura correlaciones lineales, haciéndolo poco efectivo frente a dependencias que no lo son.

La evaluación realizada en Snehi et al. (2024) compara exclusivamente los resultados arrojados por el modelo con la colección completa de características frente a los resultados con la selección. Sin embargo, no se evalúan de forma crítica estos resultados frente a los que surgen cuando se seleccionan características diferentes empleando otras técnicas.

Es por esto que el uso de selección de características puede resultar subóptimo y es uno de los ámbitos que se busca potenciar.

Método	Ventajas	Limitaciones
Coefficiente de Pearson (PCC)	<ul style="list-style-type: none"> • Cálculo sencillo y fácilmente interpretable. • Eliminar características con alta correlación lineal. 	<ul style="list-style-type: none"> • Ignora relaciones no lineales. • Puede eliminar variables útiles si hay colas pesadas, outliers o ruido.
ANOVA	<ul style="list-style-type: none"> • Seleccionar características que presentan diferencias altas entre clases. • Es un método rápido. 	<ul style="list-style-type: none"> • Asume normalidad y homocedasticidad, haciéndolo poco realista. • Máxima potencia sólo si la función de pérdida es cuadrática.
Propuesta (PCC + ANOVA/Kruskal-Wallis + Importances)	<ul style="list-style-type: none"> • Se combina simplicidad (Pearson), robustez (tests paramétricos y no paramétricos) y contexto multivariable (Importances). • Mitiga limitaciones univariadas y lineales. • Más adaptable a datos no normales de contextos IoT. 	<ul style="list-style-type: none"> • Mayor complejidad y coste computacional. • Requiere validación cruzada para evitar sobreajuste.

Table 4.1: Resumen crítico de metodologías de selección de características

4.1.3 Técnicas de Detección

El elemento más importante en el sistema de detección de Botnets es el conjunto de técnicas que se seleccionan para analizar el tráfico de red y concluir la existencia o no de este tipo de redes maliciosas.

Aunque el presente trabajo se centra en el empleo de Algoritmos de Aprendizaje Profundo es conveniente analizar otros estudios que, haciendo uso de algoritmo de aprendizaje automático u otras técnicas, han abordado el problema desde un enfoque diferente.

Métodos de Aprendizaje Automático

Entre los métodos de Machine Learning más habituales destacan los árboles de decisión y los algoritmos de *ensemble* como los Random Forest.

En Ferrag et al. (2020), al igual que en muchos otros trabajos, se combinan varios de estos métodos. En concreto, se hace uso de dos clasificadores (un Robust Error-Pruned Tree, abreviado en adelante como REP Tree, y un JRip) que reciben el mismo conjunto de características y realizan sus propias clasificaciones de forma independiente. Este resultado es posteriormente proporcionado a un Random Forest junto al conjunto original, de este modo el bosque puede hacer uso de los patrones ya aprendidos por los dos algoritmos más simples para reforzar su propia predicción.

Es importante observar que en este estudio se menciona uno de los principales problemas de la detección de Botnets mediante Aprendizaje Automático, el desbalanceo de clases, pues la clase mayoritaria tiende a abrumar al modelo con su presencia y complicar la detección del resto de ellas. En este sentido, su decisión de emplear una combinación de modelos así como algoritmos de *ensemble* resulta acertada pues REP Tree produce una clasificación binaria entre tráfico malicioso y benigno que, combinada con la predicción multiclase del método basado en reglas JRip y el Random Forest (que hace uso de un conjunto amplio de árboles individuales), reducen significativamente el impacto del desbalanceo en la decisión final, minimizando el sesgo y la varianza de la predicción.

También en Moubayed et al. (2020) se parte de un Random Forest como algoritmo principal para la detección. En este estudio, al igual que en el previamente mencionado, se menciona el problema del desbalanceo de clases; el dataset empleado por los autores, TI-2016 DNS, presenta un fuerte desequilibrio en sus clases (benigna vs botnet). Los árboles de decisión pueden verse afectados seriamente por esta problemática pues su división habitual en los nodos (basada generalmente en entropía o en Gini) tiende a favorecer las clases más habituales. Esto, además, se aprecia en los resultados finales que ofrece el propio artículo, donde se observa que el algoritmo sin modificaciones arroja una muy alta accuracy pero muy baja recall, es decir, el bosque prioriza la predicción de la clase mayoritaria. Los autores resaltan la importancia y la necesidad de emplear técnicas de balanceo como SMOTEs que, aunque disminuye ligeramente la accuracy del modelo, permite que este tenga suficientes ejemplos de la clase menos representada para poder aprender de forma sólida sus patrones y reforzar así sus predicciones de la misma.

En el caso de Snehi et al. (2024) se usan asimismo bosques de árboles de decisión pero, a diferencia del caso de Ferrag et al. (2020), se combinan con algoritmos basados en el descenso del gradiente, en particular XGBoost. El desbalanceo importante de los datasets IoT se menciona también como un problema notable a tener en cuenta. No obstante, los autores del artículo deciden no generar datos sintéticos empleando técnicas como SMOTEs si no realizar validación cruzada estratificada para generar conjuntos de entrenamiento en los que se mantiene la proporción original de las clases. Afrontando, por tanto, el desbalanceo mediante la combinación de modelos.

Los dos modelos base empleados son un Distributed Random Forest y un XGBoost. El Random Forest, que como se ha mencionado previamente al ser un algoritmo de *ensemble* resulta muy conveniente para conjuntos desequilibrados, hace uso de bagging, es decir, entrena cada árbol con un subconjunto de entrenamiento diferente cuyas características han sido también seleccionadas aleatoriamente. Este método reduce significativamente la varianza y el sobreajuste del bosque, un riesgo alto en datasets desbalanceados. Sin embargo, se combinan estas predicciones con las realizadas por XGBoost, que también emplea árboles de decisión pero de forma secuencial, optimizando los mismos con el descenso del gradiente.

Estos dos modelos básicos se combinan en un metamodelo final que, al igual que en

Ferrag et al. (2020), empleará ambas predicciones para, dando mayor importancia a unas que a otras en función del caso, realizar sus propias predicciones finales.

En resumen, los algoritmos de aprendizaje automático basados en árboles de decisión son una opción robusta para detectar Botnets en redes IOT (alcanzando accuracies superiores al 95% con recall por encima de 90%) cuando se emplean en conjunto. Los Random Forest, por su condición de *ensemble*, permiten mitigar notablemente el efecto del desbalanceo aunque no son lo suficientemente potentes como para poder solventarlo por su cuenta, requiriendo o bien técnicas de resampling o bien la combinación de varios modelos.

Referencia	Algoritmos	Estrategia ante desbalanceo	Ventajas	Limitaciones
Ferrag et al. (2020)	REP Tree + JRip + Random Forest	Clasificación jerárquica (binaria + multiclase) combinada en RF final	Reduce sesgo y varianza, combina patrones simples y complejos	Requiere 3 modelos, mayor coste computacional
Moubayed et al. (2020)	Random Forest optimizado	SMOTE sobre el train para equilibrar clases	Mejora recall y F1 en clase minoritaria	Ligera caída en accuracy global
Snehi et al. (2024)	Distributed Random Forest + XGBoost	Validación cruzada estratificada (sin oversampling)	Complementariedad bagging (RF) + boosting (XGB)	No genera ejemplos nuevos para clase minoritaria

Table 4.2: Resumen de estudios con algoritmos basados en árboles y estrategias ante desbalanceo

Aunque los árboles de decisión y sus derivados sean los métodos seguramente más populares también existen análisis realizados con algoritmos más simples de aprendizaje automático.

En Doshi et al. (2018) se emplean otros métodos no basados en árboles como K Near-est Neighbours, donde se evalúan los registros mediante distancias optimizando el entrenamiento (aunque sufre en la inferencia en datasets de gran volumen), o Support Vector Machine. Los resultados arrojados por estos modelos se encuentran en general por encima de 0.91 en todas las métricas, siendo el SVM quien muestra en ocasiones peores resultados.

No obstante, es importante hacer notar que, tal y como se menciona en el artículo, el dataset escogido muestra un desbalanceo muy notable a la inversa que el que uno podría esperar, habiendo 15 veces más paquetes maliciosos que benignos. Los propios autores mencionan que, incluso prediciendo todos ellos como malignos la accuracy se mantiene en 0.93. Este experimento resulta, por tanto, muy concreto para el caso estudiado, un ataque de denegación de servicio por inundación. También conviene observar que, en el caso de un ataque Flood, el sistema de detección mediante ML resulta efectivo pero excesivo, pues una inundación es notablemente fácil de detectar mediante la observación de métricas de tráfico como el número de paquetes por segundo.

Por lo tanto, el estudio, aunque de resultados prometedores, se realiza en un escenario muy sesgado que puede no representar de forma fidedigna las condiciones reales en un entorno IoT con patrones menos evidentes y ataques más sutiles. Además, en el caso de

un ataque de inundación, como se ha mencionado, resulta más ligero y más rápido emplear métodos estadísticos para la detección.

En Watanabe et al. (2024) se emplea también KNN, así como Support Vector Classification. Se observa como estos métodos obtienen resultados en torno a 0.95-0.97 en F1 Score. Estos números son peores que aquellos logrados por los métodos basados en árboles con los que se compara en el mismo artículo. En este caso, el problema del desbalanceo de clases en el conjunto de entrenamiento se ve mitigado mediante el uso de honeypots, que atraen ataques de Botnets logrando así una mayor representación de tráfico malicioso. Esto, aunque efectivo, tampoco simula un entorno IoT real en el que no existiría una tan alta cantidad de registros de Botnets.

En resumen, los métodos no basados en árboles parecen ofrecer una alternativa extremadamente situacional que ofrece buenos resultados en casos muy particulares de ataques IoT donde, además, existen alternativas o bien menos complejas computacionalmente (métodos estadísticos) o bien más potentes en su inferencia (Random Forests).

Referencia	Algoritmos	Estrategia ante desbalanceo	Ventajas	Limitaciones
Doshi et al. (2018)	K-Nearest Neighbours, SVM lineal	Ninguna; dataset con 15 veces más malicioso que benigno	Métricas >0.91 en todos los casos, muy alta detección de ataques Flood	Escenario muy sesgado, poco representativo; en Flood se puede detectar con métodos estadísticos más simples
Watanabe et al. (2024)	KNN, Support Vector Classification	Captura de tráfico malicioso adicional mediante honeypots	F1-score \approx 0.95–0.97, adaptable a cambios de patrón	Representación irrealmente alta de tráfico malicioso; peor rendimiento que métodos basados en árboles

Table 4.3: Resumen de estudios con algoritmos no basados en árboles y estrategias ante desbalanceo

Deep Packet Inspection (DPI)

Existe un tipo particular de análisis de tráfico que, aunque pueda emplear aprendizaje automático, conviene analizar por separado, el Deep Packet Inspection.

En un DPI se analiza no solo la cabecera del paquete si no también su contenido (payload), recopilando así toda la información y no solo elementos como el puerto o el protocolo del envío.

En McDermott et al. (2018), por ejemplo, se recopila mediante Wireshark las cabeceras y el campo de información (que contiene, entre otras cosas, el payload). Esta información extraída es luego proporcionada a modelos de aprendizaje automático de la misma forma que en estudios previamente analizados.

Primero, convierte dichos datos en vectores numéricos teniendo en cuenta su frecuencia en el texto y su rareza en el conjunto de textos (TF-IDF). Una vez tiene los datos en un

formato aceptable puede emplearlos para entrenar una Red neuronal recurrente de memoria a corto y largo plazo bidireccional (Bidirectional Long Short-Term Memory – Recurrent Neural Network, BLSTM-RNN). Este tipo de redes neuronales son muy efectivas para capturar secuencias en ambas direcciones (pudiendo detectar relaciones tanto posteriores como previas al dato de análisis).

El estudio muestra como la red neuronal presenta una accuracy por encima de 0.92 en todos los casos llegando incluso al 0.99 cuando realiza la inferencia sobre Mirai.

En Song et al. (2020) también se emplea el payload del paquete para hacer uso de expresiones regulares y reglas con el objetivo de detectar anomalías en la red. El elemento fundamental del estudio es el parámetro de Hurst, un valor estadístico que mide el grado de autocorrelación temporal de una serie, permitiendo, como es el caso, detectar un cambio en el tráfico que puede indicar la existencia de un ataque en un momento temporal dado. El artículo demuestra que el valor de Hurst aplicado sobre DPI es muy sensible a cambios en el tráfico y por tanto altamente eficiente para la detección de anomalías.

En resumen, el Deep Packet Inspection resulta muy útil para proporcionar conjuntos de datos de entrenamiento a modelos tanto de ML como de DL permitiendo que estos realicen una inferencia con resultados muy sólidos.

Sin embargo, conviene resaltar que DPI puede ser muy costoso computacionalmente. En un entorno IoT real, donde sería conveniente que el sistema pudiera detectar el ataque en tiempo real, la necesidad de leer todos los paquetes aumenta el coste computacional, el empleo de recursos y la latencia del sistema. Es un método poco escalable, que crece de forma notable a medida que aumentan los dispositivos de la red.

Además, existe un riesgo legal que conviene no pasar por alto. En el IoT pueden existir situaciones en las que el dueño o encargado de la red no es el único participante de la misma, por tanto, leer el contenido de todos los paquetes que son enviados en la misma puede vulnerar normativas de privacidad y protección de datos de los usuarios de la red, como el RGPD.

Referencia	Método de DPI	Modelos / Técnicas	Ventajas	Limitaciones
McDermott et al. (2018)	Extracción de cabeceras y payload con Wireshark, conversión TF-IDF	BLSTM-RNN	Muy alta accuracy (hasta 0.99 en Mirai), captura patrones complejos en secuencias	Alto coste computacional; requiere acceso completo a paquetes
Song et al. (2020)	DPI con expresiones regulares y reglas, análisis estadístico de payload	Parámetro de Hurst + reglas heurísticas	Muy sensible a cambios en tráfico, detección de anomalías en tiempo real	No compara contra ML/DL moderno; escalabilidad limitada en redes grandes

Table 4.4: Resumen de estudios que emplean Deep Packet Inspection para detección de anomalías o botnets

Deep Learning

A la vista de los buenos resultados ofrecidos por los métodos de aprendizaje automático basados en árboles y de la mano de los notables avances en el campo del ML es inevitable pensar que los modelos de Aprendizaje Profundo pueden ser mucho más eficientes en la detección de Botnets en redes de dispositivos.

En Ouhssini et al. (2024) se emplea una red neuronal sobre el dataset CIDDS-001. El estudio analiza diferentes modelos y determina que la red neuronal que mejores resultados puede ofrecer es una red CNN-LSTM-Transformer.

En DeepDefend toman la entropía del tráfico como serie temporal que proveer a las redes neuronales, buscando predecir la entropía futura y, si esta resulta anormal, vincularlo a un posible ataque de denegación de servicio. La combinación de diferentes tipos de redes neuronales resulta extremadamente efectiva para lograr este objetivo.

Por un lado, la red convolucional (CNN) resulta muy eficaz para detectar patrones locales como posibles subidas o bajadas muy rápidas de la entropía. La memoria larga a corto plazo (LSTM) complementa a la CNN buscando detectar patrones más largos en la serie temporal. Por último, el Transformador aporta atención, pudiendo detectar qué momentos pasados son los más relevantes para la predicción futura.

El uso de la entropía reduce enormemente la cantidad de datos que recibe el modelo y aumenta tanto la posibilidad de verse afectada por ruido como el tiempo de procesamiento de la información. Además, permite la activación del modelo solo cuando se detecta un cambio de entropía, sin necesidad de que este esté en análisis constante.

El uso de un único dataset así como la alta complejidad y requerimientos de software que implica la implementación de una red neuronal CNN-LSTM-Transformer son las principales desventajas que se presentan en este sistema. Tampoco debe pasarse por alto que el uso de la entropía, aunque con sus ya mencionadas ventajas, puede ignorar determinados ataques y patrones más sutiles.

El problema de la complejidad del modelo es evitado en Doriguzzi-Corin et al. (2020) mediante el uso de una red CNN más ligera. El estudio ofrece una validación más robusta que en el caso anterior, empleando 3 datasets de la Universidad de New Brunswick, concretamente ISCX2012, CIC2017 y CSECIC2018. Los flujos extraídos son recibidos por una red neuronal convolucional cuyos hiperparámetros han sido seleccionados buscando maximizar la métrica de evaluación F1 Score sin disparar el coste de tiempo y de recursos que requiere el modelo.

Los resultados del análisis muestran como un modelo tan ligero como el aquí presentado es capaz de arrojar una accuracy que ronda el 0.99, comparable e incluso mejor que otros modelos más complejos como el DeepDefense 3LSTM.

También en Elsayed et al. (2023) se opta por emplear modelos ligeros de aprendizaje profundo. En este estudio se opta por el dataset UNSW 2018 IoT Botnet, que presenta una variedad de registros obtenidos de dispositivos IoT reales con diversos tipos de ataque.

El modelo elegido en este caso es una red neuronal que resulta de una combinación de

dos redes más simples. Por un lado se emplea la ya habitual CNN, pero, en este caso, se combina con una red neuronal de tipo GRU. La elección de una unidad recurrente cerrada por encima de una red de memoria larga a corto plazo viene motivada especialmente por su mayor ligereza y simplicidad, al poseer únicamente dos compuertas, haciéndola una opción mucho más eficiente y mucho más apropiada para contextos de pocos recursos.

El estudio muestra como una combinación de estas redes presenta unos resultados equiparables e incluso superiores a los de otros sistemas del estado del arte, con una accuracy en torno al 0.99, mientras mantiene un coste computacional no muy elevado frente a otras opciones de aprendizaje profundo.

No obstante, la validación empleando un único dataset supone un punto crítico a la hora de garantizar su robustez y eficacia.

Referencia	Arquitectura	Datasets	Ventajas	Limitaciones
Ouhssini et al. (2024) (DeepDefend)	CNN + LSTM + Transformer sobre series de entropía	CIDDS-001	Predicción anticipada de anomalías; activa el modelo solo en cambios de entropía	Alto coste computacional; ignora ataques sutiles; un único dataset
Doriguzzi-Corin et al. (2020)	CNN ligera	ISCX2012, CIC2017, CSE-CIC-2018	Bajo coste; accuracy aprox. 0.99; validación robusta con varios datasets	Menor expresividad que arquitecturas más profundas
Elsayed et al. (2023)	CNN + GRU (modelo ligero)	UNSW 2018 IoT Botnet	Accuracy aprox. 0.99; muy eficiente en recursos; idóneo para IoT/fog	No detecta bien clases minoritarias; validación en un único dataset

Table 4.5: Resumen de estudios que emplean Deep Learning para detección de Botnets en IoT

4.1.4 Conversation analysis

Como se ha mencionado anteriormente el análisis de los paquetes, ya sea en profundidad o no, puede suponer un coste en recursos computacionales y en tiempo no asumibles por el sistema de detección. Es por ello, que muchas de las propuestas más recientes optan por el análisis conversacional.

En el análisis conversacional se busca evitar en la medida de lo posible el uso de características de los paquetes, ignorando el payload y tomando, quizás, únicamente los protocolos y puertos empleados. Entre los sistemas de análisis conversacional resulta de especial interés PeerShark.

El fundamento de PeerShark es la agrupación del tráfico en conversaciones entre dos nodos tomando únicamente una dupla de IP de origen e IP de destino. Esto lo diferencia de otros análisis similares donde se agrupa por IPs, protocolo y puerto haciendo de ellos, en cierto modo, una mezcla entre análisis conversacionales e inspección de paquetes. Esta condición hace de PeerShark un método protocol-oblivious, port-oblivious y, por supuesto payload-oblivious.

El estudio Narang et al. (2014) y este tipo de algoritmo de análisis cobra una importancia capital si se comprende el riesgo que entrañan las Botnets P2P que se han explicado previamente. Este tipo de redes maliciosas, como Waledac, se camuflaban en protocolos peer-to-peer legítimos haciéndolas altamente difíciles de detectar mediante un análisis convencional. Una evaluación de este tipo permite detectar cambios sutiles en el comportamiento agregado de las conversaciones.

Los autores emplean un dataset estrictamente balanceado, donde se encuentran 50.000 conversaciones P2P benignas y otras tantas maliciosas. Esto evita los problemas de desbalanceo previamente mencionados y hace que un modelo pueda entrenarse con datos suficientes como para detectar patrones en las conversaciones de los bots. Aunque un análisis conversacional en redes P2P puede justificar en mayor medida el equilibrio de clases en su dataset, pues busca detectar a la botnet en su estado latente, es decir, no en el momento del ataque si no cuando los dispositivos infectados se comunican entre sí. El ataque (salvo un ataque DDOS) será seguramente breve, con pocos registros de tráfico, sin embargo, la conversación entre los nodos de la botnet puede presentar un mayor número de envíos.

Al no analizar paquetes, las características empleadas por los modelos son también diferentes. En este caso, se emplean las siguientes métricas:

- La duración de la conversación
- El número de paquetes intercambiados
- El volumen de información intercambiada
- La mediana del inter-arrival time

Se evita analizar los contenidos de los paquetes y se busca detectar patrones en la forma en la que los nodos hablan entre ellos empleando métricas muy simples pero muy representativas del estilo de una conversación.

Esta información extraída es, al igual que en el resto de casos, empleada para entrenar uno o varios modelos de aprendizaje automático con los que realizar predicciones. El estudio emplea:

- Árboles de Decisión
- Boosting REP Trees
- Redes Bayesianas

Los árboles de decisión son capaces de detectar ciertos patrones más simples que los que pueden llegar a aprender el resto de modelos empleados. Si se observan los resultados se aprecia que este tipo de modelos han obtenido una accuracy muy elevada, llegando al 0.98. Además, se refuerza la robustez del modelo en el estudio empleando otras métricas de evaluación como TPR, FPR y el área bajo la curva ROC, donde también alcanza resultados que rondan, por lo general, el 0.98.

Los Boosting REP Trees que emplean Adaboost son capaces de alcanzar una AU-ROC casi perfecta, >0.99 , pero esto viene a costa de un ligeramente peor rendimiento en TPR/FPR. Es decir, la separación entre clases es muy buena pero el umbral seleccionado incentiva la clase botnet derivando en un mayor número de falsos positivos.

Las redes bayesianas utilizadas, sin embargo, aunque también mejoran AUROC no ven tan perjudicada su tasa de falsos positivos, haciendo de ellas una sólida opción.

Este estudio muestra de forma clara como, empleando una recogida de información que no requiere análisis en profundidad de los paquetes con sus implicaciones en coste de recursos, pueden obtenerse resultados similares aquellos de otros métodos más complejos.

4.1.5 Resumen y próximos pasos

Una vez analizados algunos de los trabajos más relevantes en el área conviene realizar un último repaso general de la situación, haciendo hincapié en los aspectos faltantes que el presente trabajo busca cubrir o expandir.

Ha quedado ampliamente demostrada la eficacia de los métodos de aprendizaje automático basados en árboles, muy extendidos y utilizados. Su simplicidad ofrece indudables ventajas entre las que destacan la facilidad de despliegue y el bajo consumo de recursos. Son sólidos y aportan un nivel de accuracy muy alto. No obstante, esa misma simplicidad es la que los limita. Tienen mayores dificultades para detectar patrones subyacentes complejos, además de verse altamente limitados para hacer frente a conjuntos de datos desbalanceados, tan habituales en el IoT. Es por ello que su uso, aunque adecuados en muchos casos, no son fácilmente aplicables a entornos IoT reales.

El Deep Packet Inspection, aunque de resultados muy muy buenos es un callejón sin salida en el IoT. Proporciona resultados muy buenos porque emplea información interna del contenido de los paquetes pero sus desventajas lo vuelven una opción inviable dentro del contexto del internet de las cosas. Por un lado, la necesidad de leer la totalidad o casi la totalidad de cada uno de los paquetes enviados lo vuelven un método lento e intensivo. Si el sistema de detección buscara ser en tiempo real o, al menos, capaz de mitigar el ataque, cualquier tiempo extra perdido sería inaceptable.

Además, aunque muchas veces pasado por alto, la vulnerabilidad de la privacidad de los usuarios debe tenerse en cuenta y el riesgo legal que esto conlleva deben tenerse en cuenta. En definitiva, los resultados del análisis en profundidad de los paquetes no justifican sus limitaciones y riesgos.

Los dos campos más prometedores son el aprendizaje profundo y el análisis conversacional. Por un lado, los modelos de redes neuronales del Deep Learning (CNN, GRU, LSTM...) han demostrado su eficacia con resultados de detección muy positivos, siendo su alto coste computacional su principal talón de Aquiles. Por otro lado, el análisis conversacional resuelve problemáticas habituales en los sistemas de detección, como la alta cantidad de información y el ataque a la privacidad al analizar paquetes. No obstante, se encuentra aun poco estudiado y probado en entornos IoT que se asemejen a la realidad.

Por todo lo anteriormente mencionado, el valor de este trabajo no radica en proponer un nuevo modelo que busque maximizar aun más la precisión de la detección, pues ha quedado demostrado que ya existen soluciones robustas a este problema. Su objetivo es demostrar la viabilidad de desplegar un sistema de detección de Botnets en Fog Layer bajo

una arquitectura en microservicios, equilibrando la precisión con el coste computacional. Frente a trabajos previos muy enfocados en contextos irreales y modelos casi perfectos en la presente propuesta se busca la robustez, la viabilidad práctica de la solución y la innovación mediante el uso de elementos aun poco explorados en el ámbito en cuestión.

Diseño e Implementación del Sistema

El objetivo final del presente trabajo es el diseño y posterior despliegue de un servicio de análisis de tráfico y detección de Botnets integrado en una red IoT. Es por ello que una de las primeras consideraciones debe ser el enfoque arquitectónico del mismo; el diseño de una topología adecuada es fundamental para lograr un servicio lo más eficiente posible.

Además, conviene conocer la forma en la que ha sido implementado el servicio, tanto sus requisitos técnicos como los códigos que lo componen. Un entendimiento en profundidad de ambas partes, la arquitectónica y la de código, permite comprender las decisiones adoptadas, garantizar su reproducibilidad y evaluar su funcionamiento, mantenimiento y posible futura ampliación.

Como se ha mencionado en el planteamiento inmediatamente previo relativo al estado del arte, este sistema busca expandir, complementar y ampliar el diseño presentado en Snehi et al. (2024). Para ello, se concibe en una arquitectura modular configurada en microservicios independientes, primando así la reproducibilidad, la escalabilidad y la eficiencia. Además, se ofrecen análisis de desempeño de métodos de aprendizaje profundo y se comparan los resultados de diversos modelos, permitiendo esto analizar la optimalidad de unos frente a otros.

Este capítulo busca aclarar los dos objetivos definidos y proporcionar una explicación detallada del sistema y su implementación.

5.1 Requisitos

Se han establecido diversas prioridades en el diseño del sistema de detección, las cuales pueden ser categorizadas como funcionales o no funcionales.

Los requisitos funcionales son 3: captura de tráfico bajo demanda, análisis de dicho tráfico y presentación de resultados.

El servicio puede considerarse un sistema de seguridad integrado en la red que, o bien periódicamente o bien cuando el responsable de la misma detecta alguna anomalía, pueda ser consultado. Por tanto, debe existir un endpoint expuesto en el cual se puedan realizar

peticiones que inicien el análisis. La duración de la captura será configurable por la persona que realiza la solicitud, capturando todo el tráfico de la red durante el periodo de tiempo establecido y generando al terminar un archivo de tipo pcapg y dos CSVs apropiados que podrá emplear el proceso de análisis de tráfico.

El análisis de tráfico, como se describirá más adelante, consiste en un preprocesamiento de los datos capturados en el fichero para hacer posible su ingestión por parte de un modelo de aprendizaje automático. Este modelo habrá sido previamente entrenado y será el mejor de todos aquellos probados en la fase de testing. Este debe ser capaz de producir como resultado un análisis que determine la probabilidad de que en el tráfico que ha analizado haya presente una botnet.

Sin embargo, conviene tener en cuenta otro tipo de cuestiones de diseño, aquellas relativas a la eficiencia y a la interoperabilidad.

Un buen servicio o aplicación pensado para desplegarse en un entorno de Internet de las Cosas, heterogéneo tanto en software como en hardware por su propia naturaleza, debería ser capaz de funcionar en prácticamente cualquier sistema o dispositivo, es decir, *independencia de la plataforma*. Esta interoperabilidad debe garantizarse a todos los niveles, tanto en requerimientos del sistema operativo como en protocolos.

Además, presentando 3 funcionalidades claramente diferenciadas (captura, análisis y visualización) resulta interesante apostar por la modularidad en su arquitectura, haciendo que estas 3 funciones puedan desplegarse y ejecutarse de forma independiente, pudiendo así utilizar solo una o dos de ellas sin que se requieran ajustes.

La modularidad y la interoperabilidad serán los dos principios fundamentales sobre los que se basará la solución aquí presentada.

Por último, resulta fundamental la seguridad y la tolerancia a fallos. Poder actuar ante la caída de parte del servicio y, además, garantizar el acceso al mismo solo a las personas apropiadas (siendo una aplicación con acceso interno al tráfico de la red) son dificultades de primer nivel en el despliegue.

Tras estudiar los requisitos anteriormente mencionados se ha considerado la arquitectura containerizada en microservicios como el diseño más apropiado para este sistema de detección de botnets. Un enfoque basado en microservicios permite el despliegue de aplicaciones independientes y el uso de contenedores facilita su portabilidad y modularidad. Además, garantiza el funcionamiento del sistema si una de las funcionalidades caen, pudiendo, por ejemplo, seguir capturando tráfico y almacenar el fichero resultante incluso si el sistema de análisis está momentáneamente inoperativo y es empleado más tarde. Esta robustez también facilita el trabajo de monitoreo y parcheado del sistema pues, a diferencia de una aplicación monolítica tradicional, permite la modificación y el rediseño de parte en lugar de todo el servicio, agilizando el trabajo.

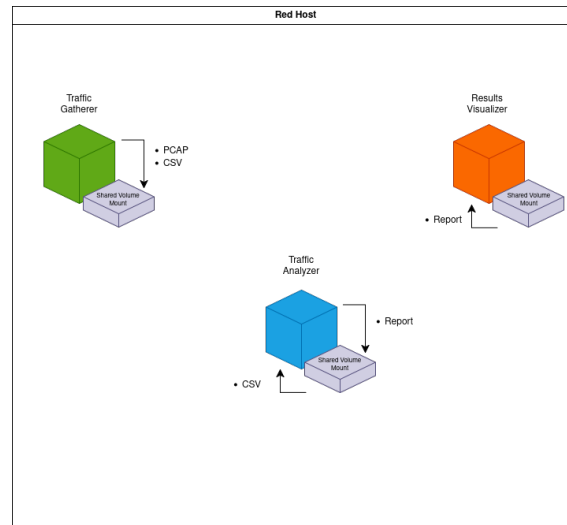


Figure 5.1: Implementación de la Arquitectura del servicio

El requisito de modularidad del sistema choca frontalmente con la comodidad y la facilidad del usuario, ya que, si los microservicios deben ser capaces de operar de forma independiente bajo demanda (pudiendo, por ejemplo, ejecutar solo la captura) no puede darse el caso de que ninguno de ellos invoque a otro. Esto impide la posibilidad de que sea el propio servicio de captura quien invoque al de análisis o este al de visualización de resultados. Por tanto, y como se abordará más adelante, la arquitectura escogida presenta también desventajas e inconvenientes que deberán tratarse.

5.2 Comparativa de Arquitecturas

En esta sección se ahondará en el diseño arquitectónico del servicio, entrando en detalle de por qué una concepción en microservicios es la más apropiada para este caso; cómo esta garantiza los requerimientos anteriormente mencionados y sus ventajas frente a otros modelos descartados.

Primero se estudiarán sus ventajas frente a modelos monolíticos de diseño, para posteriormente comparar la arquitectura de microservicios frente a otras formas de diseño distribuidos, enfocando siempre la comparativa en el problema abordado en el presente trabajo.

5.2.1 Distribución o Monolitos

La arquitectura en microservicios, como arquitectura distribuida, es fundamentalmente opuesta a la aplicación monolítica, como se ha observado en los preliminares, conviene por tanto saber el por qué de la elección de una frente a la otra, siendo esta una de las primeras disyuntivas que se presentaron.

La primera gran ventaja que se observa es la practicidad. La solución aquí presentada, aunque un solo proceso, es una aplicación dividida de forma muy clara en fases

independientes y con un dominio muy marcado; la captura, el análisis y la visualización. Es importante hacer notar también que, si diseñada correctamente, ninguna de estas funcionalidades necesita del resto para funcionar. Por tanto, inmediatamente viene a la mente la visión de tres aplicaciones pequeñas actuando en conjunto para realizar un proceso común más elaborado. Esta modularidad no se permite en una arquitectura monolítica tradicional que, aunque puede admitir diferenciación de funcionalidades como en una arquitectura monolítica por capas, no son módulos que actúen de forma independiente y, por tanto, no se ha considerado apropiada.

Existen, no obstante, estilos monolíticos con gran modularidad como la arquitectura de pipeline en la que se concibe un proceso donde cada módulo recibe unos datos, los procesa y se los envía al siguiente módulo en la secuencia. Aunque este diseño es, aparentemente, idóneo por su clara división del servicio en fases presenta importantes limitaciones que deben tenerse en consideración.

Por un lado, los módulos viven en el mismo sistema, esto supone un problema de escalabilidad al hacer imposible escalar funcionalidades particulares de ser necesario. Esta falta de desacoplamiento implica también la necesidad de reestructurar todo el proceso si se quisiera cambiar de nodo alguna de las etapas. Además, no puede desplegarse únicamente uno o varios módulos sueltos ya que pertenecen al sistema global y no es posible extraerlos del mismo.

Todas estas limitaciones hacen de las arquitecturas monolíticas diseños poco apropiados para el problema que se pretende abordar y fueron inmediatamente descartadas.

5.2.2 Los Microservicios

Una vez se observa como una arquitectura distribuida es la más idónea para la solución que aquí se implementa se plantea la siguiente decisión: qué tipo de arquitectura distribuida escoger.

La Arquitectura Enfocada en Servicios (SOA) también busca dividir la aplicación en funcionalidades basadas en su dominio. Estas suelen ser más grandes y complejas que en el caso de los microservicios y, además, no son completamente independientes. Su objetivo está más enfocado a la reusabilidad de sus componentes que al desacople y la independencia de los mismos. Prueba de esto es que los servicios comparten, por lo general, elementos como la Base de Datos o un Bus. En este caso, y aunque podría ser un diseño apropiado, se ha preferido primar la completa independencia de los procesos debido al contexto particular de su despliegue, el entorno IoT, cuyas condiciones heterogéneas de software y hardware, como se ha mencionado previamente, no pueden garantizar la interoperabilidad de prácticamente ningún servicio o aplicación.

Las siguientes dos arquitecturas que se analizarán a continuación buscan solventar el problema mencionado anteriormente, la automatización del proceso y la facilidad de cara al usuario. Como se ha hecho notar previamente, la interacción más sencilla para la persona que desea utilizar el servicio sería poder lanzar un único proceso que ejecute todos los pasos

del análisis, y esto se podría realizar de dos formas diferentes.

Por un lado, podría concebirse un diseño basado en eventos. En este tipo de diseño existiría un Broker de eventos encargado de gestionar los eventos emitidos al finalizar cada uno de los microservicios y ejecutar el siguiente paso del proceso. Esta arquitectura, sin embargo, añade un elemento, el Broker, de complejidad innecesaria a la hora de controlar el flujo. Además, surge un acoplamiento indirecto entre los servicios, y la ejecución de uno de ellos sin el despliegue del resto se complica pues requiere reconfigurar suscripciones a eventos o modificar el Broker.

Otra opción, de cara a resolver el mismo problema, sería un sistema de colas mediante, por ejemplo, RabbitMQ. El usuario lanzaría el servicio inicial de captura y este, al terminar se publicaría un mensaje que consumiría el servicio de análisis para iniciar su funcionalidad. Los problemas presentados por este tipo de diseño son los mismos que en el caso anterior, la complejidad y la falta de un desacoplamiento real.

No existiendo realmente la necesidad de incluir infraestructura y requerimientos externos, por ejemplo forzando la instalación de RabbitMQ en el sistema anfitrión del servicio, es conveniente evitarlo. No hacerlo así generaría una dependencia en las mismas que no sería inmediatamente portable a otros entornos si no quesaría responsabilidad del usuario gestionar esa pieza faltante del proceso. Además, la complejidad y el manejo de los eventos y las colas complican el sistema aumentando su probabilidad de fallo.

Por todo esto, se ha optado por el diseño en microservicios, sacrificando la automatización total del proceso en pos de la independencia.

Arquitectura	Principales desventajas
Monolítica tradicional	Fuerte acoplamiento entre componentes, ninguna posibilidad de desplegar módulos de forma independiente, difícil mantenimiento y escalado parcial.
Monolítica tipo pipeline	Modularidad interna pero no desacoplamiento real, todos los módulos deben convivir en el mismo sistema, sin posibilidad de escalar o reubicar funcionalidades de forma parcial.
SOA (Arquitectura Orientada a Servicios)	Servicios no completamente independientes; tienden a compartir elementos como bases de datos o buses, lo que reduce la portabilidad y complica despliegues en entornos heterogéneos como IoT.
Arquitectura orientada a eventos (EDA)	Requiere broker de eventos, añade complejidad innecesaria al flujo de control, acoplamiento indirecto entre servicios, difícil trazabilidad y testeo parcial.
Arquitectura basada en colas (RabbitMQ)	Introduce dependencia fuerte de una infraestructura externa, mismo acoplamiento indirecto que EDA y mayor dificultad para despliegues parciales.

Table 5.1: Resumen de limitaciones de arquitecturas descartadas frente al enfoque basado en microservicios

5.3 Arquitectura General del Sistema

Una vez se ha establecido que se optará por un diseño en microservicios, se realiza una primera presentación de los mismos a alto nivel, específicamente conviene determinar cuántos y cuáles serán los microservicios implementados, cómo se conectarán entre ellos y qué tipo de red emplearán.

En cuanto a la primera cuestión, la división de funcionalidades es muy clara; la captura del tráfico de red, el análisis del mismo empleando aprendizaje automático y la visualización de resultados. Existiendo así 3 microservicios de dominios claramente limitados y sin solape entre los mismos.

La conexión entre los mismos es, como se he mencionado en la sección previa, uno de los mayores inconvenientes arquitectónicos. Es por ello, que se ha optado por evitar que los 3 microservicios se comuniquen entre sí de forma directa, garantizando y manteniendo su independencia, y permitiendo que estos compartan los datos necesarios en una unidad de volumen compartida. Esta falta de conexión permite que, si se desea implementar, por ejemplo, un sustituto al servicio de análisis, el de captura no trate de realizar un tipo particular de conexión con este para el que el nuevo microservicio no estaría preparado.

En cuanto a la red, el microservicio de captura de tráfico podría compartir red con el servidor anfitrión, esto facilitaría el acceso del mismo a la red local de la que debe

tomar los datos de tráfico. Sin embargo, las vulnerabilidades de seguridad que esto podría traer consigo hacen que se haya optado por que el mismo se ejecute en la red propia "botnetdetection" otorgándole permisos de lectura de tráfico. Los otros microservicios, sin embargo, no tienen este requerimiento y puede, sin ningún inconveniente ni requerimiento adicional, emplear esa misma red. Esto se debe a que el análisis y la captura de tráfico no necesitan acceder a la red local (solo al volumen compartido) y permitirlo expondría innecesariamente ambos contenedores al sistema.

La arquitectura en microservicios escogida se fundamenta en los siguientes principios, considerados críticos a lo largo de toda la implementación. El primero y uno de los más importantes es el desacoplamiento. Se busca que los 3 microservicios sean absolutamente independientes entre sí, para garantizar que cualquiera de ellos puede extraerse del conjunto del sistema y desplegarse en cualquier otro contexto con mínimas o ninguna modificación.

Muy vinculado a este principio se encuentra la división funcional/modularidad. Para garantizar la independencia de los 3 microservicios es importante que cada uno de ellos cumpla una función claramente diferenciada, como engranajes de un sistema. Esto, además, asegura evitar la repetición de código pues no hay partes del proceso realizadas por más de un microservicio.

Los siguientes dos principios guardan asimismo una estrecha relación. Por un lado, la portabilidad. Los dos principios anteriores facilitan y posibilitan este. Los microservicios buscan ser portables, que puedan tomarse y llevarse a otro sistema sin dificultad. Además, se quiere garantizar la reproducibilidad en el despliegue. Estas dos propiedades se garantizan mediante la containerización.

La elección de Docker y el empleo del despliegue en contenedores asegura que cualquiera de los microservicios, con sus requerimientos de software pueden reproducirse exactamente igual en cualquier sistema que posea Docker. Esto evita problemas clásicos como el "funciona en mi máquina" e, incluso a nivel comercial, asegura que el servicio pueda ser empleado por cualquier cliente independientemente de su dispositivo.

El resultado de aplicar estos principios al esquema de diseño planteado anteriormente genera el siguiente flujo de trabajo.

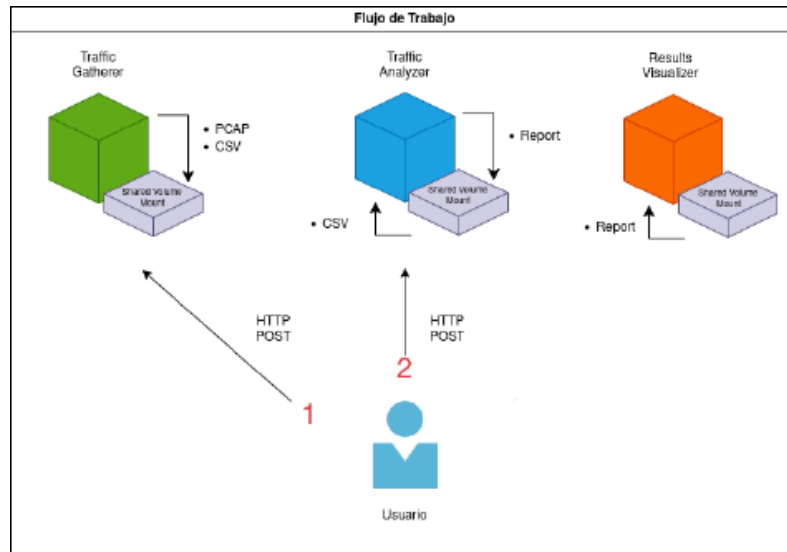


Figure 5.2: Flujo de la Aplicación

Como puede observarse en ?? es el usuario quien debe invocar manualmente cada microservicio o, si desea facilitarse el proceso, generarse su propio programa encargado de realizar el proceso completo de forma secuencial.

En su defecto, el flujo sería el que sigue. El usuario comienza haciendo uso del microservicio de recogida de datos de tráfico, esto puede venir motivado por una revisión periódica de la red o porque este ya sospeche de la existencia de una botnet en la misma. Para ello, realiza una petición HTTP POST al endpoint del microservicio especificando durante cuánto tiempo debe realizarse la recogida. El proceso se ejecuta y los datos se almacenan en un PCAP y en dos CSV (tras limpiar y agrupar flujos) en el almacenamiento compartido.

Una vez el proceso ha finalizado el usuario observa por terminal el mensaje relativo a su completitud y entonces procede a realizar la siguiente petición. Esta es también una solicitud HTTP POST pero dirigida al microservicio de análisis, en la que se le especifica la ruta del fichero a analizar.

Al recibir esta petición, el proceso asume la existencia de un archivo CSV con datos relativos a tráfico, lo carga, preprocesa y emplea el modelo seleccionado y almacenado para realizar predicciones de clasificación. El dato resultante de mayor importancia es la probabilidad de que exista una Botnet dentro del tráfico analizado. Este análisis se almacena también en el volumen compartido, también como archivo CSV (se ha optado por este formato al ser ampliamente soportado por distintas herramientas, de esta forma si se desplegaran los microservicios en solitario tendrían mejor encaje con otros procesos). El usuario observa la notificación de finalización y procede con el siguiente paso.

No sería necesario invocar a continuación el servicio de visualización, pues este se encuentra operativo permanentemente asumiendo la existencia de un fichero de análisis y presenta los resultados, exponiendo un dashboard en un puerto local empleando mapeo de

puertos.

5.4 Despliegue con Docker Compose

Como se ha mencionado previamente, se ha optado por emplear Docker Compose como herramienta de orquestación y despliegue de los diferentes microservicios.

En este caso, se instanciarán 3 contenedores, uno por microservicio. El primero de ellos debe poder acceder al tráfico de la red del anfitrión. Esto se realiza para poder capturar fácilmente mediante *tshark* el tráfico de la red a la que pertenece el sistema que aloja el conjunto de servicios.

En el caso del microservicio de captura de tráfico, es necesario especificar su ejecución en modo privilegiado para poder tener permisos de escucha del tráfico. Esto es potencialmente peligroso y motiva la decisión de emplear medidas de seguridad como *rate limiting* o autenticación para evitar un uso malintencionado del mismo.

Surge además un problema clásico en las arquitecturas modulares que buscan implementar microservicios completamente independientes, poder hacer uso en todos los contenedores de ficheros generados por el resto. Para ello se monta un volumen (un almacenamiento persistente de los datos) común en los tres. Este volumen será quien aloje los ficheros generados por cada uno de los procesos para que el resto pueda emplearlos.

El volumen compartido es crítico para el funcionamiento del servicio, pues garantiza que los datos y resultados generados en cada uno de los contenedores no desaparecerán si este deja de estar operativo (el problema de persistencia de datos es una de las más importantes limitaciones en Docker y los volúmenes suponen una efectiva solución como se indica en Ibrahim et al. (2021)) y serán accesibles entre microservicios. En este caso se monta como volumen el directorio *shared* que debe ser creado previo despliegue de los microservicios.

Este despliegue y orquestación es reproducible empleando el fichero *docker-compose.yml*, que define cada uno de los servicios y su forma de despliegue, independientemente del sistema operativo en el que uno se encuentre. Esta independencia y reproducibilidad no son las únicas ventajas que plantea la elección de Docker Compose como herramienta.

Como parte de la concepción en microservicios se ha considerado que cualquiera de ellos podría ser desplegado en solitario, por tanto, resulta muy conveniente que exista la posibilidad de realizar pruebas de testing o monitorización de los microservicios de forma individual.

De la mano de esta parcialidad viene la posibilidad de realizar el despliegue de forma progresiva, garantizando por ejemplo que los microservicios se encuentren correctamente desplegados antes de proceder con el siguiente. Incluso si uno de los contenedores fallara podría redeplegarse sin necesidad de reinstanciar y redespargar todos ellos.

Estas ventajas se obtienen, además, con muy baja dificultad de implementación. Docker Compose permite su uso mediante sencillos comando por terminal (*docker compose up/-*

down para levantar y bajar servicios o docker compose logs para monitoreo).

No conviene, de todas formas, obviar las limitaciones presentadas por docker compose. Entre las más notables se incluyen la falta de monitoreo sin aplicación externa, pues, aunque pueden observarse los logs de los contenedores, deben realizarse a mano y por terminal. Además, no existe la gestión avanzada de estados de control ni de dependencias. Es por ello que en un entorno de producción real, si se dispusiera de una mayor cantidad de recursos, sería recomendable el empleo de herramientas más avanzadas como Kubernetes.

Por último, Docker Compose permite también el despliegue de dos servicios que resultan claves a la hora de garantizar la robustez del sistema de detección, Prometheus y Grafana, permitiendo el monitoreo constante de los microservicios.

5.4.1 Dockerfiles

Las plantillas de generación de las imágenes se encuentran definidas en los archivos Dockerfiles, estos incluyen los requisitos y los códigos necesarios para el funcionamiento de los microservicios y son construidas e instanciadas mediante el comando docker compose up.

Como parte del estudio arquitectónico del servicio, y antes de profundizar en la implementación de cada uno de ellos, conviene revisar estos ficheros para cada uno de los contenedores.

Se ha optado en las tres imágenes por partir de una imagen base que incluye Python 3.10 (decidiendo no tomar la versión más reciente de Python y de esta forma garantizar la compatibilidad de todas las librerías) y que corre en un sistema Debian Slim.

Debian Slim o Alpine Linux

Se ha optado por escoger la imagen base en Debian Slim garantiza un tamaño extremadamente reducido, consumiendo menos almacenamiento en el dispositivo anfitrión que con un sistema más tradicional como Ubuntu.

Aunque existe una forma de disminuir aún más el tamaño de la imagen, empleando como base Linux Alpine, este sistema es extremadamente minimalista y hace que haya que instalar a mano de formas poco convencionales ciertas librerías (como podría ser el caso de Pyarrow que tiene fuertes dependencias de librerías creadas en C++ diseñadas para glibc, mientras que Alpine emplea musl) y la reducción de tamaño no compensa la complejidad y el tiempo que habría que invertir en lograr la instalación de todas ellas.

Dockerfile de Traffic Gatherer

Una vez se ha definido la base, en el caso del *traffic gatherer*, se define un directorio de trabajo, se actualizan los repositorios y se instala tshark, el software que se empleará para la recolección de datos de tráfico de red.

A continuación, se copian los códigos de la aplicación y se realizan las instalaciones necesarias de librerías de python empleando el fichero de requerimientos.

Por último, se define el comando de punto de entrada, es decir, aquel que se ejecuta por defecto al instanciar el contenedor, siendo, en este caso, la ejecución mediante `uvicorn` de la aplicación `FastAPI` contenida en su interior y expuesta en el puerto 9000.

Dockerfile de Traffic Analyzer

El procedimiento es, en este caso, muy similar.

- Directorio de trabajo
- Copia de ficheros
- Instalación de librerías de Python
- Comando de punto de entrada

Este Dockerfile, a diferencia del anterior, no requiere de la instalación de software fuera de las librerías de Python.

Dockerfile de Traffic Visualizer

5.5 Microservicios

Una vez se tiene clara la estructura de arquitectura del servicio puede entrarse en profundidad al estudio de cada módulo de forma individual. En cada microservicio se pretende analizar el flujo de trabajo del mismo, su modo de uso y sus ficheros generados.

Además, se hará un breve análisis técnico de las herramientas de software empleadas por cada uno de ellos. Este análisis abarca desde las librerías de Python que permiten emplear modelos de aprendizaje automático para realizar las predicciones hasta el uso de Prometheus para exponer métricas de monitorización del microservicio.

5.5.1 Microservicio de Captura de Tráfico

El objetivo principal de este microservicio es recopilar los datos de tráfico de la red que se busca analizar y procesarlos de tal forma que se generen los archivos necesarios para realizar un análisis empleando técnicas de aprendizaje automático.

Este microservicio es el primer paso dentro del proceso de detección de botnets en redes IoT y busca permitir el análisis en tiempo real en aquellos entornos de producción donde no se está probando con datasets predefinidos.

Tecnologías empleadas

Una vez se tiene esto claro, se puede hacer un análisis de las tecnologías empleadas en la implementación del microservicio, entre las que destacan:

- FastAPI
- tshark
- nsenter
- SlowAPI

- Authorization
- Python
- Prometheus

FastAPI permite la construcción de APIs en Python de forma sencilla y rápida, es por ello que se ha optado por la elección de este framework. Su facilidad a la hora de la implementación, acompañada de una gran robustez y alta velocidad de funcionamiento hacen de ella una opción ideal para este tipo de proyectos. Empleando FastAPI se ha definido el endpoint `/gather`, el cual admite solicitudes de tipo POST con un parámetro `duration` que permite modificar la duración de la recogida de tráfico a gusto del usuario.

Una de las grandes ventajas que incorpora FastAPI es su capacidad de generación de documentación automática mediante swagger UI. Esta funcionalidad permite probar fácilmente el endpoint, pudiendo observar los parámetros admitidos y las respuestas esperadas, haciendo más sencillo su mantenimiento.

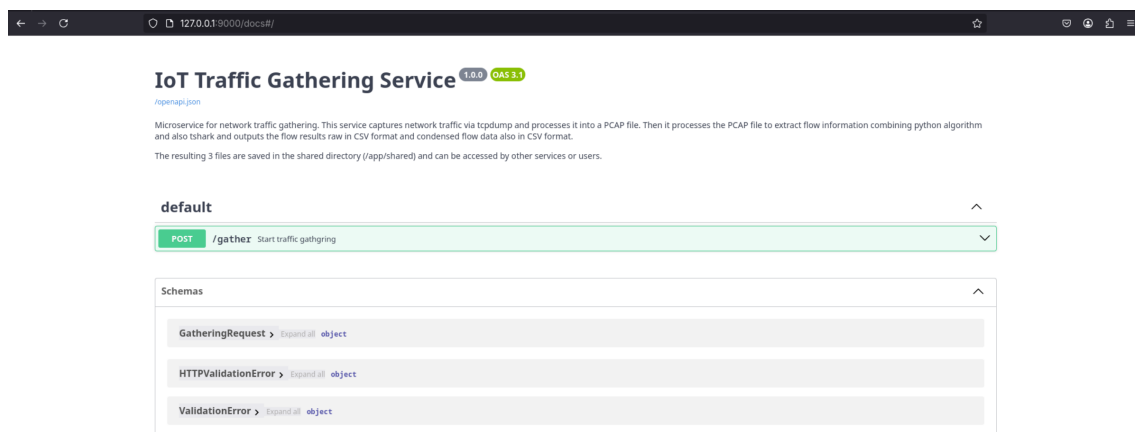


Figure 5.3: Documentación generada automáticamente del endpoint de captura

De la mano de FastAPI, se ha decidido emplear la librería SlowApi. Esta librería permite implementar rate limiting en la API generada usando FastAPI. Como se verá más adelante, esto supone una garantía de seguridad crítica en un microservicio como este.

Se ha decidido emplear autenticación mediante tokens para aumentar aun más la seguridad del servicio.

El elemento fundamental de este microservicio es el software de captura de tráfico, en este caso tshark junto con nsenter. A la hora de hacer uso de él se le especificará la red local como objetivo de captura y la ruta al fichero PCAP donde se deberá escribir el resultado, nsenter permite que el comando de captura se ejecute en el host, y no en el contenedor, pues por seguridad este no puede ver la red local. Este archivo generado será almacenado en el volumen compartido. Se analiza el código de respuesta de tshark para evaluar si la recogida ha sido correcta o no y poder monitorizar el error.

Se ejecuta entonces nuevamente tshark para extraer la información relevante sobre los envíos del PCAP y almacenarla en un fichero CSV para su posterior procesado y análisis, el

cual se realiza a continuación, agrupando los envíos en flujos en base al emisor, el receptor y el protocolo y almacenándolo en otro archivo CSV. Es importante que estos resultados generados se guarden también en el volumen compartido, de esta forma el microservicio de análisis podrá hacer uso de ellos.

Por último, se ha implementado Prometheus como parte de la API. Este software posibilita la creación y recolecta de métricas de monitoreo que permiten al dueño del sistema comprobar y asegurar que este funciona correctamente

Estructura del Microservicio

El microservicio se encuentra ordenado de la siguiente forma.

- `traffic_gatherer/`
 - `Dockerfile`
 - `requirements.txt`
 - `main.py`
 - `README.md`
 - `.env`
 - `src/`
 - * `gather.py`
 - * `verify.py`
 - * `process.py`
- `shared/`

El directorio `shared` es el volumen compartido ya mencionado, es por ello que no se volverá a ahondar en el mismo.

La división presentada busca ser sencilla de entender y fácil de mantener. El código `main.py` es el punto de entrada lanzado por FastAPI, mientras que aquellos que se encuentran en el directorio `src` implementan las diferentes lógicas de la captura de tráfico, desde la propia recogida al procesamiento de flujos pasando por las comprobaciones de seguridad.

El servicio (`main.py`) se lanza empleando Uvicorn, exponiendo así el microservicio vía HTTP en el puerto 9000 del propio sistema anfitrión, al haber sido desplegado este microservicio en la red local.

Archivos Generados

Este proceso genera 3 ficheros diferentes en el volumen compartido *shared*.

El primero de ellos es un archivo PCAP generado inmediatamente tras la captura de tráfico de tshark al comienzo del proceso. Este fichero, aunque fundamental, no puede emplearse directamente por los modelos de aprendizaje automático, por tanto debe procesarse y convertirse en un fichero estructurado.

Es aquí donde vuelve a emplearse tshark, esta vez sobre el archivo PCAP. En este caso la herramienta permite convertir los datos crudos de tráfico en un archivo CSV con los

envíos de paquetes estructurados (datos de timestamp, IPs de origen y destino, protocolos etc). Este archivo ya puede ser consumido por un modelo de ML que busque hacer el análisis por paquetes. No obstante, como se ha comprobado en las pruebas de validación y se menciona asimismo en Narang et al. (2014), los resultados son notablemente mejores si se consideran los flujos conversacionales como objeto del análisis.

Por tanto, se aplica un proceso de agrupación en flujos empleando python al archivo CSV recién generado y se produce otro nuevo fichero CSV con los datos relativos a los flujos.

Estos 3 archivos, almacenados en el volumen compartido, pueden ser empleados por el microservicio de análisis pero, como parte de la implementación independiente y modular, podrían usarse por otros servicios o aplicaciones diferentes que permitan gestionar este formato de ficheros.

Seguridad y Control de Accesos

El microservicio de captura de tráfico posee una vulnerabilidad crítica que conviene mencionar. Este microservicio de captura posee accesos privilegiados para posibilitar la captura del tráfico y esto puede emplearse de forma malintencionada por un usuario no legítimo. Es por ello por lo que resulta fundamental implementar medidas que impidan o dificulten este uso nocivo del microservicio.

Empleando la librería SlowApi previamente mencionada se ha establecido un sistema de rate limiting. Este sistema permite establecer un máximo de solicitudes en un periodo determinado de tiempo, evitando así posibles ataques de denegación de servicio. Es importante notar como, aunque se produjera uno de estos ataques DOS, la independencia y la modularidad de la arquitectura en microservicios permitirían que solo la funcionalidad de captura de tráfico estuviera inoperativa, pudiendo emplear de forma normal el análisis y la visualización mientras el microservicio afectado se redespiega.

Se ha diseñado también un sencillo sistema de autenticación por Token. Al recibir una solicitud el servidor comprueba si esta viene acompañada de una cabecera con el Token válido esperado por la misma (y almacenado en .env), de no ser así se considera la petición como proveniente de un usuario ilegítimo y no se procesa.

La combinación de estos dos sistemas de defensa buscan evitar un uso malintencionado del servicio y garantizar que solo el dueño del mismo podrá hacer uso de él.

Monitoreo

Aunque la seguridad del microservicio es un elemento crítico que no se debe pasar por alto, también resulta importante poder asegurar que el contenedor no se ha caído o ha sufrido algún error sin que el dueño del mismo se haya percatado.

Para esto, se ha decidido implementar Prometheus como parte de la API. Por un lado, se expone un nuevo endpoint en la misma, *metrics*, en el cual se vuelven accesibles ciertas métricas básicas de la API como el número

de solicitudes recibidas, el tiempo de respuesta del endpoint o el número de peticiones en proceso que pueden facilitar la detección de bloqueos en el manejo de las mismas. Sin embargo, conviene implementar algunas métricas concretas que ayudan a este microservicio en particular.

Se ha decidido crear dos métricas para el microservicio de captura de tráfico, por un lado el tamaño en bytes del fichero CSV de flujos generado y por otro su número de filas. Esto se realiza con el objeto Gauge de Prometheus y pueden, posteriormente, observarse ambas empleando Grafana como se verá en la sección correspondiente del presente trabajo.

Conclusión

En resumen, el microservicio de captura de tráfico representa una funcionalidad fundamental en aquellos entornos donde no exista una herramienta de recogida de tráfico y generación de datasets.

Su diseño, en el que priman la seguridad y la robustez, permite un uso sencillo y configurable por el usuario a la vez que garantiza que el acceso al mismo estará limitado a aquellas personas con autorización para su uso. Además, se facilita al usuario el control en tiempo real de que el microservicio permanece operativo y funciona correctamente.

Se ha conseguido una completa modularidad de la captura, sin dependencia del microservicio de análisis posterior pudiendo así realizar su despliegue como complemento de otra aplicación no necesariamente de análisis de botnets. No obstante, los ficheros generados, en formatos ampliamente aceptados en el área, son el punto de entrada necesario del siguiente microservicio.

Aspecto	Descripción
Tecnologías	FastAPI, TShark, SlowAPI, Autenticación por Token, Python
Endpoint principal	POST <code>/gather</code> con parámetro <code>duration</code> y cabecera de Token
Seguridad	Autenticación con token (<code>.env</code>), rate limiting (1 req/min) con SlowAPI
Archivos generados	<ul style="list-style-type: none"> • <code>.pcap</code>: captura cruda del tráfico • <code>.csv</code>: extracción estructurada de paquetes • <code>.csv</code>: agrupación conversacional por IP y protocolo
Estructura del microservicio	<code>traffic_gatherer/</code> con subcarpeta <code>src/</code> (código), <code>main.py</code> , <code>Dockerfile</code> , <code>requirements</code> y volumen compartido <code>shared/</code>
Modo de despliegue	Docker, expuesto en puerto 9000
Modularidad	Independiente del resto de servicios. Puede usarse de forma aislada o integrada.
Ventajas	Captura real en red del host, seguridad reforzada, generación automática de documentación OpenAPI, robustez frente a fallos y monitoreo

Table 5.2: Resumen del microservicio de Captura de Tráfico

5.5.2 Microservicio de Análisis del Tráfico

En el caso de este microservicio, que constituye el segundo paso del proceso completo, se pretende analizar la probabilidad de existencia de una Botnet en una muestra de tráfico de red empleando un modelo de aprendizaje profundo.

Se busca que su diseño no dependa del proceso de captura de tráfico recién presentado y pueda integrarse en cualquier entorno en el que existan uno o más ficheros CSV con los datos que se busca analizar.

Tecnologías empleadas

Conviene observar qué tecnologías y herramientas de software se han empleado en este microservicio, siendo algunas de ellas las mismas que en el caso del recolector de tráfico y que, por tanto, no se explicarán nuevamente en profundidad.

Para el Análisis de Tráfico se ha utilizado:

- FastAPI
- SlowApi
- Authorization
- Pickle
- Python (SKLearn...)

- Prometheus

Debido a la necesidad de que cada servicio pueda ser invocado de forma independiente se ha repetido el planteamiento de generar una API mediante FastAPI. En este caso definiendo el endpoint *analyze* que, admitiendo como parámetro file la ruta al fichero a analizar, comenzará el proceso.

En este caso, igual que en el microservicio anterior, FastAPI proporciona documentación automática.

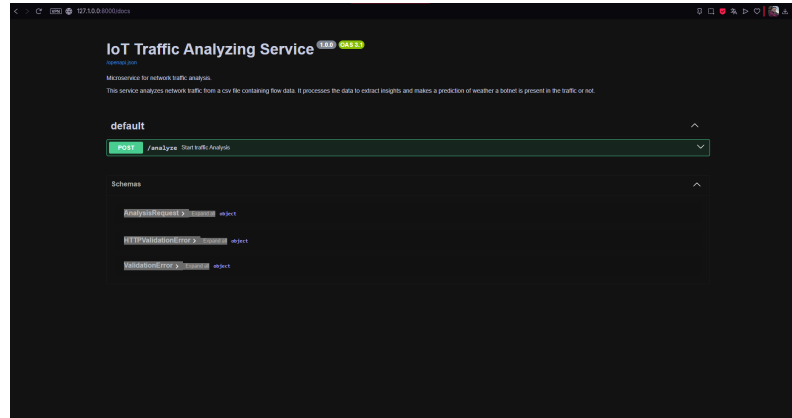


Figure 5.4: Documentación Generada Automáticamente para el Servicio de Análisis de Tráfico.

Se ha decidido repetir el sistema de seguridad empleado en el microservicio de captura; la autenticación mediante Token y el rate limiting para evitar inundaciones de tipo HTTP, las tecnologías empleadas para esto son las mismas que en el caso anterior, autenticación por Token mediante un código de Python y SlowApi para la limitación de peticiones.

El modelo, para su almacenamiento de forma sencilla, se guardará empleando la librería Pickle de Python. Este módulo permite serializar modelos de sklearn, se almacena serializado y se carga para su uso en los códigos de Python.

También en Python se realizan las predicciones del modelo, su evaluación de confianza y su almacenamiento. Esto se realiza empleando librerías populares de aprendizaje automático como la ya mencionada sklearn o pandas.

De la misma forma que en el caso anterior, se hace uso de Prometheus para garantizar que el microservicio de análisis opera correctamente.

Estructura del Microservicio

Para facilitar su entendimiento en un despliegue estándar con el resto de microservicios se ha optado por replicar la arquitectura de directorios y de aquellos ficheros que sean comunes, resultando en:

- `traffic_analyzer/`
 - `Dockerfile`
 - `requirements.txt`

- `main.py`
- `README.md`
- `.env`
- `src/`
 - * `analyze.py`
 - * `verify.py`
- `Models`
 - * `model.sav`
- `shared/`

De la misma forma que en el caso anterior se emplea `main.py` como punto de entrada de la API, el cual invoca la lógica de análisis y se carga el modelo almacenado en `Models`.

Además, se observa como existe también el directorio compartido `shared`, en el que se encontrarán los códigos o bien generados por el microservicio de captura o bien por el sustituto empleado por el usuario.

En este caso, también mediante Uvicorn, se expone el microservicio via HTTP en el puerto 8000 del sistema anfitrión empleando mapeo de puertos.

Archivos Generados

Este microservicio genera únicamente un archivo en el volumen compartido.

Este fichero, `predicciones.csv`, consiste en un registro que representa cada flujo encontrado en el archivo de flujos, su clase predicha (botnet o no) y la probabilidad o confianza asociada.

El formato seleccionado, CSV, es fácilmente empleable posteriormente para analizar las predicciones y valorar el riesgo de ataque. El microservicio de visualización hace uso de este fichero pero cualquier sustituto o un análisis manual del mismo también puede realizarse.

Seguridad y Control de Accesos

No se ahondará especialmente en esta sección pues las dos técnicas empleadas para garantizar la seguridad y la robustez del microservicio son las mismas que en el caso anterior.

Se emplea `SlowApi` para limitar el número de peticiones que se pueden realizar en un minuto y evitar así la saturación del sistema.

Además, se espera que la petición incluya un token de autenticación en la cabecera para su comprobación y validación por parte del microservicio. Evitando también su procesamiento de no serlo.

Estas dos medidas suponen una garantía de seguridad para el usuario que emplea el microservicio.

Monitoreo

En este caso, al igual que en el microservicio de captura de tráfico, se usa Prometheus para la exposición y recogida de métricas de monitoreo del microservicio que, posteriormente, serán visualizables mediante Grafana.

Se exponen las métricas básicas así como dos métricas adicionales específicas del microservicio. De la misma forma que en el caso anterior, la primera de las métricas es el tamaño en bytes del fichero de predicciones generado mientras que la segunda es su número de filas. Este segundo valor, al ser común a ambos microservicios, permite de forma muy sencilla comparar si realmente el análisis se ha realizado sobre el fichero de flujos correcto que ha sido generado por el microservicio de captura.

Conclusión

Como se ha visto, el microservicio de análisis de tráfico tiene un papel crítico en todo este proceso pues es el encargado de aportar información sobre si la red está sufriendo o no un ataque.

Esto implica que, aunque pueda emplearse un método propio de análisis del tráfico, este microservicio es el más fundamental en este proceso.

Aspecto	Descripción
Tecnologías	FastAPI, SlowAPI, Autenticación por Token, Pickle, Python (sklearn, pandas)
Endpoint principal	POST /analyze con parámetro file (ruta al CSV) y cabecera de Token
Seguridad	Autenticación con token (<code>.env</code>), rate limiting (1 req/min) con SlowAPI
Archivos generados	<ul style="list-style-type: none"> • <code>predicciones.csv</code>: clase predicha y probabilidad por flujo
Estructura del microservicio	<code>traffic_analyzer/</code> con subcarpeta <code>src/</code> (lógica), <code>main.py</code> , Dockerfile, carpeta <code>Models/</code> con el modelo serializado y volumen compartido <code>shared/</code>
Modo de despliegue	Docker, expuesto en puerto 8000, mapeo de puertos (<code>-p 8000:8000</code>)
Modularidad	Independiente del microservicio de captura. Puede usarse sobre cualquier CSV compatible.
Ventajas	Carga flexible de modelos, análisis automatizado por API, seguridad integrada, salida estandarizada y visualizable y monitoreo

Table 5.3: Resumen del microservicio de Análisis de Tráfico

5.5.3 Microservicio de visualización de Resultados

Este último microservicio busca, únicamente, ofrecer al usuario la capacidad de visualizar de forma sencilla gráficos que aporten información relativa al análisis del tráfico.

En pos de la independencia y la modularidad se busca que este microservicio no dependa del resto de elementos, por tanto, asume únicamente la existencia de un fichero CSV en el volumen compartido *shared* de nombre *predictions.csv*.

Este fichero debe poseer al menos una columna *prediction* con la predicción del modelo y *probability* con la confianza en dicha predicción.

Tecnologías empleadas

Este microservicio resulta ser diferente al resto de los aquí presentados en la mayoría del software empleado.

- Python
 - Plotly
 - Streamlit
 - Pandas
- Prometheus

En este caso, no se emplea FastAPI para exponer una API. Para el microservicio de visualización se hace uso de Streamlit, una herramienta de software muy ligera y sencilla para la creación de dashboards interactivos. Este despliega un servidor web en una URL y puerto local siendo, por tanto, accesible y observable desde un navegador.

La interactividad de Streamlit permite no tener que relanzar el microservicio si se modificara el archivo de predicciones.

Estructura del Microservicio

Este microservicio posee una estructura notablemente más sencilla que el resto, siendo esta la siguiente:

- `traffic_visualizer/`
 - `Dockerfile`
 - `requirements.txt`
 - `main.py`
 - `README.md`
- `shared/`

De la misma forma que en los casos anteriores `main.py` supone el programa principal que, en este caso, expone el servidor de Streamlit, pero este no requiere códigos auxiliares como sí necesitaban el resto de microservicios.

Vuelve a aparecer también el directorio compartido `shared`, desde el que el microservicio carga el fichero de predicciones sobre el que basar su dashboard.

Archivos Generados

Este microservicio no está diseñado para generar archivos, solo para mostrar visualizaciones de otros ficheros.

Seguridad y Control de Accesos

Tampoco se han implementado medidas de seguridad al no requerir el control de solicitudes HTTP.

Monitoreo

Pese a no desplegar una API sí que resulta conveniente poder garantizar que el microservicio se encuentra operativo y el dashboard es visualizable.

Al ser un Dashboard de Streamlit y no emplear FastAPI esto no resulta en una implementación directa de Prometheus como en los dos casos anteriores, si no que requiere exponer un servidor de métricas concreto del dashboard y que Prometheus pueda capturarlo.

El Dashboard

El Dashboard, en su versión actual más sencilla, presenta los siguientes gráficos.

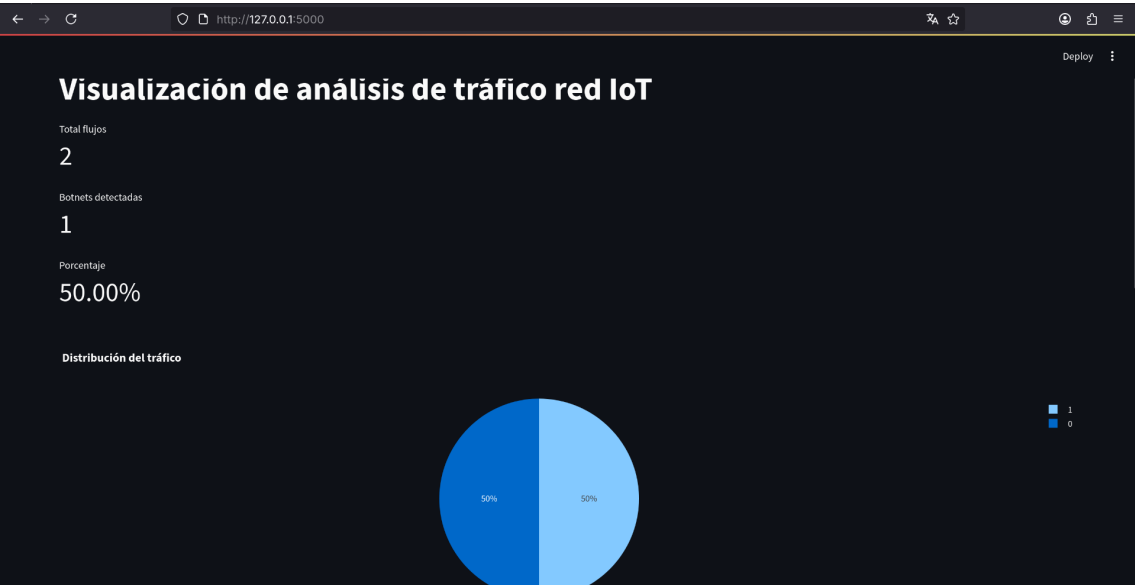


Figure 5.5: Ejemplo Dashboard de Visualización

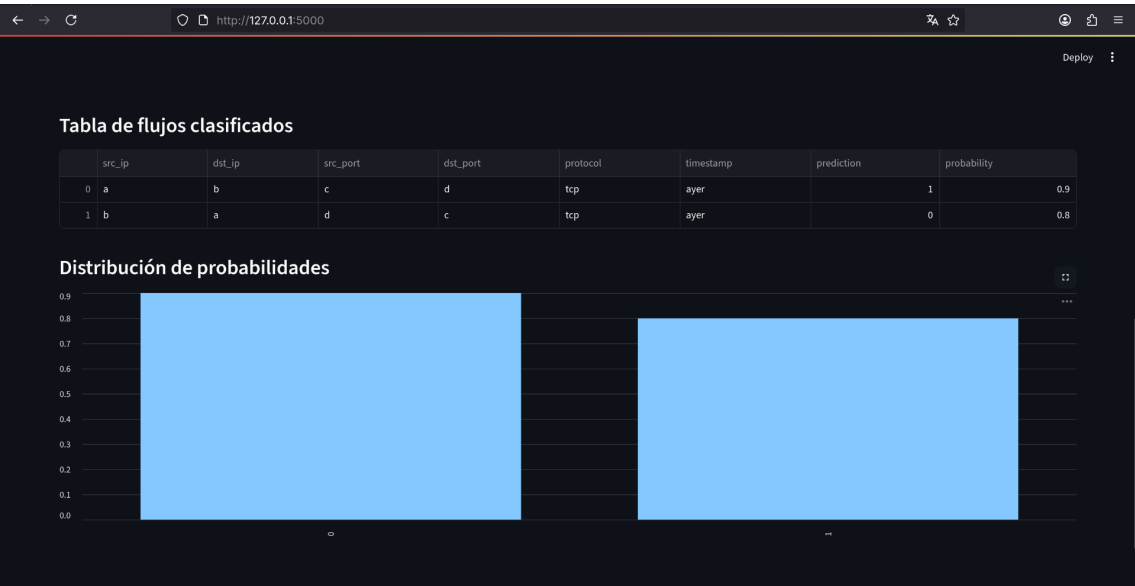


Figure 5.6: Ejemplo Dashboard de Visualización 2

Al principio se presentan 3 sencillos pero fundamentales números, primero el total de flujos analizados, a continuación las botnets detectadas y por último el porcentaje que este número de botnets supone sobre el total.

A continuación, la distribución del tráfico entre las dos clases posibles mediante un gráfico tipo *pie chart*.

Luego, se expone la tabla de flujos que se han clasificado y, por último, la distribución

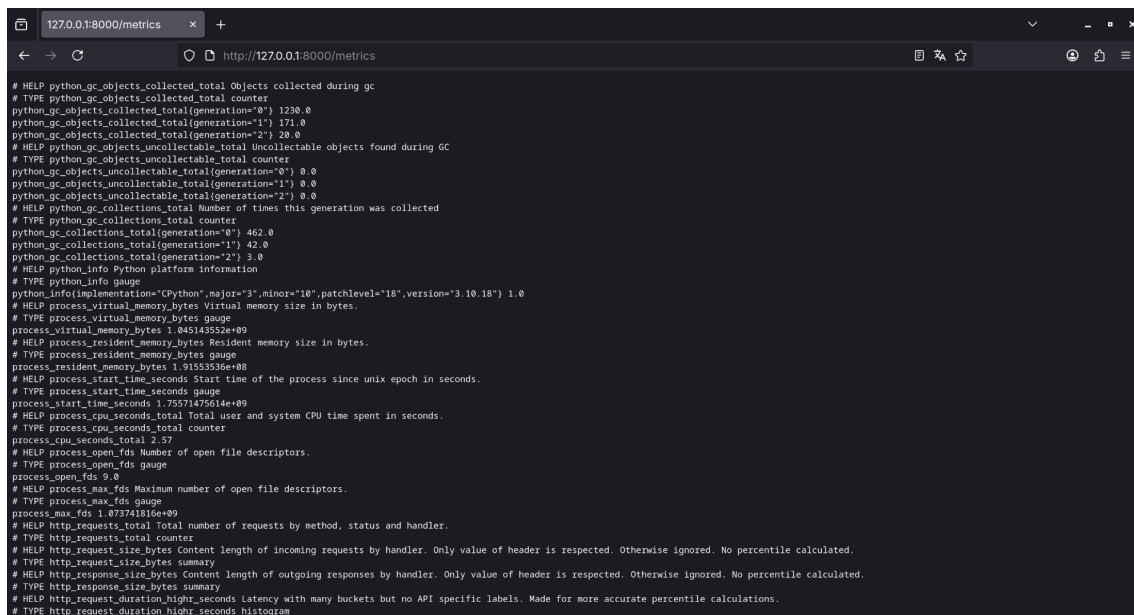
de probabilidades de las predicciones de las dos clases.

5.5.4 Prometheus

Se ha visto a lo largo del análisis de los microservicios anteriores que se hará uso de Prometheus para la recogida de métricas de monitoreo del estado de los mismos. Sin embargo, esto requiere desplegar Prometheus en el Docker Compose como si se tratara de un microservicio más, aunque no esté directamente involucrado en el proceso de detección de Botnets.

Este contenedor se basará en la imagen oficial de Prometheus y correrá en la misma red interna de Docker Compose previamente mencionada "botnetdetection" facilitando así que vea a dos de los 3 microservicios cuyas métricas debe recoger. Sin embargo, al encontrarse el microservicio de captura en la red del host debe añadirse esta como lugar adicional donde Prometheus podrá buscar el endpoint de métricas.

Además, Prometheus requiere un archivo yaml en el que se han detallado los jobs que debe realizar. En concreto, se le especifica el nombre de los 3 microservicios y sus puertos para que sepa donde encontrar los endpoints de métricas en los que debe recolectar.



```
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total(generation='0') 1230.0
python_gc_objects_collected_total(generation='1') 171.0
python_gc_objects_collected_total(generation='2') 28.0
# HELP python_gc_objects_uncollectable_total Uncollectable objects found during GC
# TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total(generation='0') 0.0
python_gc_objects_uncollectable_total(generation='1') 0.0
python_gc_objects_uncollectable_total(generation='2') 0.0
# HELP python_gc_collections_total Number of times this generation was collected
# TYPE python_gc_collections_total counter
python_gc_collections_total(generation='0') 462.0
python_gc_collections_total(generation='1') 42.0
python_gc_collections_total(generation='2') 3.0
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation='CPython',major='3',minor='10',patchlevel='18',version='3.10.18'} 1.0
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 1.045143552e+09
# HELP process_resident_memory_bytes Resident memory size in bytes.
# TYPE process_resident_memory_bytes gauge
process_resident_memory_bytes 1.91553536e+08
# HELP process_start_time_seconds Start time of the process since unix epoch in seconds.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.7557147561e+09
# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 2.57
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 9.0
# HELP process_max_fds Maximum number of open file descriptors.
# TYPE process_max_fds gauge
process_max_fds 1.073741824e+09
# HELP http_requests_total Total number of requests by method, status and handler.
# TYPE http_requests_total counter
# HELP http_request_size_bytes Content length of incoming requests by handler. Only value of header is respected. Otherwise ignored. No percentile calculated.
# TYPE http_request_size_bytes summary
# HELP http_response_size_bytes Content length of outgoing responses by handler. Only value of header is respected. Otherwise ignored. No percentile calculated.
# TYPE http_response_size_bytes summary
# HELP http_request_duration_higher_seconds Latency with many buckets but no API specific labels. Made for more accurate percentile calculations.
# TYPE http_request_duration_higher_seconds histogram
```

Figure 5.7: Ejemplo métricas básicas expuestas por el microservicio de análisis

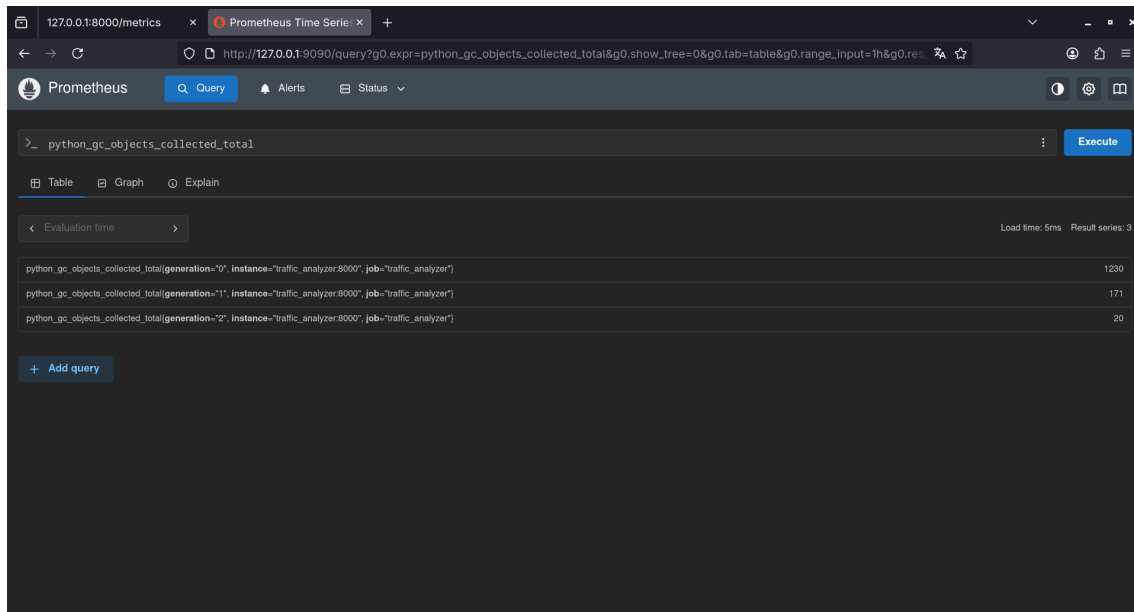


Figure 5.8: Ejemplo recogida métricas básicas por Prometheus

5.5.5 Grafana

Se ha podido comprobar como la recogida de estas métricas funciona correctamente. No obstante, esto tiene poco valor si no puede visualizarse en un momento dado el estado de estas métricas obtenidas por Prometheus. Es Grafana quien cumple ese papel.

De la misma forma que Prometheus se implementa Grafana en el Docker Compose como un microservicio más. Se emplea la imagen oficial de Grafana y se especifica que este correrá en la misma red propia en la que lo hacen el resto de contenedores (exceptuando el da captura).

Una vez se ha desplegado, puede accederse a la url expuesta por Grafana en el puerto 3000 y acceder empleando las credenciales (ya establecidas si se ha entrado previamente o por defecto). Una vez dentro se selecciona Prometheus como fuente de datos y se emplea para crear un Dashboard que incluya las métricas que queremos monitorear; por limpieza es recomendable utilizar un Dashboard por microservicio pero si quiere estudiarse el estado de todo el sistema pueden concentrarse en uno solo.

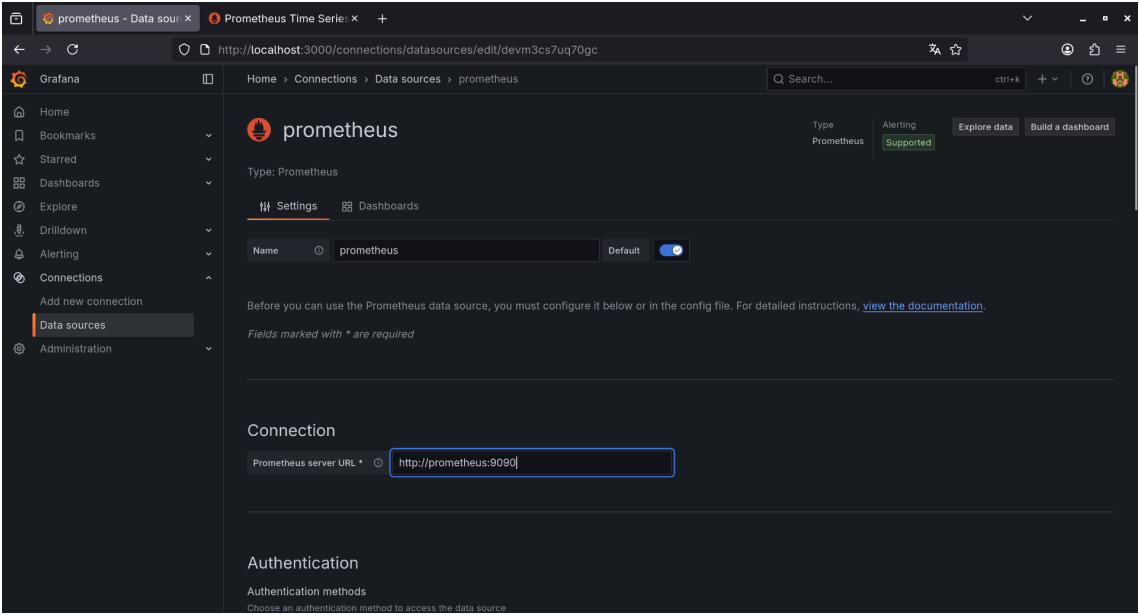


Figure 5.9: Añadir Prometheus como Data Source en Grafana

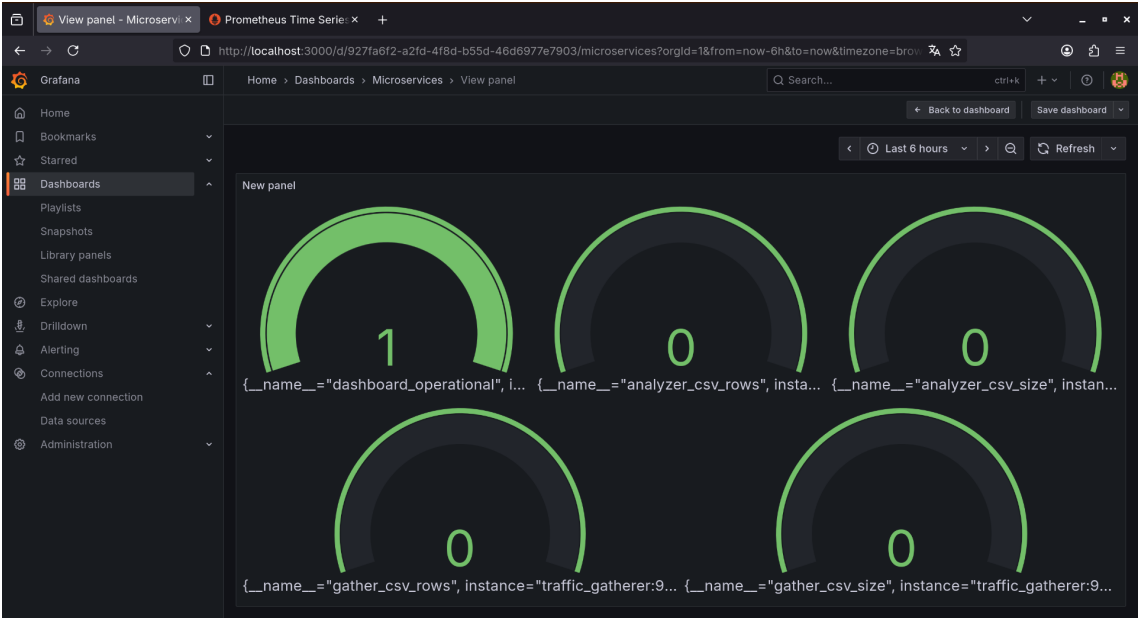


Figure 5.10: Dashboard simple de métricas creadas ad-hoc

Validación Experimental

Se ha explicado en el capítulo previo la implementación técnica de los microservicios, analizando en detalle el proceso de funcionamiento de los mismos. Sin embargo, conviene también observar las pruebas que han llevado a la elección del modelo de aprendizaje profundo finalmente seleccionado así como algunas ejecuciones del sistema de detección.

6.1 Selección del Modelo

Se procede a continuación a narrar el proceso completo de selección del modelo. Para una mejor comprensión se busca seguir un orden claro de todos los pasos que han llevado a la selección del mismo, desde la elección de datasets hasta la validación del modelo mediante métricas relevantes.

6.1.1 Selección de Datasets

Como se ha observado en la visión del estado del arte, muchos de los estudios en el ámbito emplean un único dataset, haciendo de sus resultados menos garantes de la validez del modelo en cualquier escenario. Es por ello que el presente trabajo busca utilizar un conjunto representativo de varios de los datasets más ampliamente utilizados en el área, tratando de aumentar la robustez del modelo seleccionado.

El primer dataset empleado es Aposemat IoT-23. Este dataset, que incluye tráfico malicioso y benigno de dispositivos IoT, ha sido generado por investigadores de la Universidad Técnica Checa y tiene como objetivo proveer de un amplio conjunto de datos de dispositivos reales a proyectos de detección de botnets con técnicas de aprendizaje automático.

Aposemat posee ciertas particularidades que lo distinguen de otros datasets. Por un lado, los registros no se encuentran a nivel de paquete si no de flujo, haciéndolo incomparable de forma directa a otros posibles datasets. Además, al ser tráfico real, presenta un fuerte problema de desbalanceo entre el tráfico malicioso y el benigno que, aunque una dificultad, se asemeja a la realidad del IoT. Por último, destacar que el tráfico se ha obtenido en un entorno controlado, disminuyendo el número de registros "de fondo" y existiendo la

posibilidad de que surjan patrones artificiales.

Por todo esto, los resultados obtenidos empleando este dataset pueden no ser extrapolables a otros conjuntos de datos y se priorizará el aporte del mismo a la hora de extracción de características.

Como segundo dataset se ha seleccionado Bot-IoT de la Universidad de Nueva Gales del Sur (UNSW). La principal aportación de este dataset es un enorme tamaño y su muy amplia variedad de ataques diferentes. Esto permite hacer frente a otra de las grandes problemáticas de los estudios del sector, que se centraban en la detección de un único tipo ataque, generalmente de denegación de servicio.

El problema que presenta Bot-IoT surge de su generación mediante herramientas en un laboratorio, haciendo que los patrones de comportamiento puedan destacar de forma artificial, haciendo de modelos que funcionan muy bien en este dataset poco aplicables a otros conjuntos.

Resulta, por tanto, un dataset muy útil para probar modelos y seleccionar características pero ofrece poca robustez como conjunto de datos final para validación, necesitando ser complementado por otros conjuntos.

N-BaIoT es el tercer dataset seleccionado. Este dataset, muy popular en otros análisis similares, ha sido creado por la universidad israelí de Ben-Gurion e incluye dispositivos IoT infectados por las Botnets Mirai y BASHLITE. Su estilo secuencial lo hace muy apropiado para entrenar Redes Neuronales Recurrentes aunque es importante notar que la división por dispositivo puede afectar notablemente a los resultados si no se tiene en cuenta a la hora de entrenar y validar los modelos.

Por último, se ha decidido utilizar el dataset TON_IoT, también de la Universidad de Nueva Gales del Sur. Este conjunto de datos, muy realista y con diversos formatos de tráfico, resulta una opción muy sólida como prueba final del modelo. Su heterogeneidad y diversidad en formato recién mencionada lo hacen más difícil de preprocesar que otros datasets vistos, por tanto se opta por emplearlo únicamente como validación final tras el empleo de los otros datasets.

Aspecto	IoT-23 (Aposemat)	Bot-IoT (UNSW)	N-BaIoT (Ben-Gurion)	TON_IoT (UNSW)
Ventajas	Tráfico real de dispositivos IoT infectados. Incluye varias familias de malware. Útil para extracción de características.	Gran tamaño y amplia variedad de ataques (DoS, DDoS, probing, exfiltración). Facilita comparativas entre modelos.	Muy usado en estudios previos. Dispositivos infectados con Mirai y BASHLITE. Formato secuencial, adecuado para RNN.	Heterogéneo y realista (red, logs, sensores). Más cercano a escenarios reales de IoT. Útil como validación final.
Limitaciones	Entorno controlado con poco ruido de fondo. Fuerte desbalanceo de clases.	Generado en laboratorio con tráfico sintético. Riesgo de patrones artificiales. Resultados poco extrapolables a otros entornos.	Altamente desbalanceado. El particionado por dispositivo sesga resultados. Features ya preprocesadas, poco margen para innovación.	Complejo de preprocesar (formatos diversos). Poco apto como dataset inicial de entrenamiento.

Table 6.1: Resumen comparativo de los datasets empleados en el estudio

6.1.2 Metodología de la Validación

Antes de ahondar en los contenidos de los datasets conviene determinar el proceso que se seguirá para realizar las pruebas, así como las métricas que se emplearán para validar la actuación de los modelos.

El objetivo final es lograr un modelo de aprendizaje profundo que logre una alta detección de las Botnets manteniendo una latencia baja. Por tanto, se parte de varias hipótesis que se busca demostrar.

Primero, los modelos de aprendizaje profundo capturarán mejor las relaciones no lineales y los patrones secuenciales que modelos simples de aprendizaje automático. Por otro lado, aunque útil en entrenamiento y validación, debe existir una pipeline capaz de automatizar feature extraction de forma que no deba realizarse de forma manual. Además, la tasa final de falsos negativos del modelo debe ser baja.

Se emplearán todos los datasets previamente mencionados, siendo los dos primeros la base que servirá para comparar entre sí la mayoría de modelos y poder ofrecer comparativas sólidas entre ellos. N-BaIoT será crítico a la hora de poder validar la eficacia de modelos secuenciales y, por último, TON_IoT será la prueba final a la que se enfrentarán los

modelos seleccionados.

Se concibe todo este proceso como una pipeline, en la que el primer paso tras cargar los datos es el preprocesado. Aquí, se realizará el escalado y la selección de características basada en diversos tests para añadir robustez a dicha selección. Además, es en este momento donde pueden realizarse ciertas visualizaciones interesantes como la detección de outliers (IPR, frecuencias...) y la visualización de correlaciones.

A continuación, debe hacerse frente al problema del desbalanceo, pudiéndose alterar el conjunto de datos de entrenamiento previo paso a la división en train/test.

El siguiente paso es entrenar y realizar inferencias con los modelos, siendo necesaria una seria y óptima búsqueda de hiperparámetros en todos los casos.

Luego, se comparan los resultados de los modelos empleando las métricas seleccionadas. En este caso, se han seleccionado 3 métricas principales (AUPRC, MCC y la tasa de falsos negativos), 3 secundarias (ROC-AUC, F1 y Brier) y la latencia y el tamaño del modelo. Se ha optado por FNR en lugar de los datos sobre falsos positivos pues se considera peor evaluar como benigno el tráfico malicioso que lo contrario.

Una vez se tienen las métricas pueden emplearse tests como McNemar para seleccionar el mejor modelo, que se exporta como binario para su uso en el sistema.

Además, como complemento al estudio, pueden realizarse análisis de ablaciones (quitando partes del proceso para estudiar su afectación) y Inteligencias Artificiales Explicativas.

6.1.3 Entrenamiento y Validación de los Modelos

Debido a problemas de viabilidad computacional y de recursos se ha optado por no procesar el contenido completo de todos los datasets, por resultar los mismos de un tamaño enorme, por ello, de cada dataset, se tomará una conjunto de 2 o 3 ficheros representativos y variados.

Toda la lógica de carga y limpieza básica de datasets se ha agrupado en una única clase, `DataLoader`, cuyos detalles pueden obtenerse en el apéndice correspondiente. De la misma forma, el preprocesado se agrupa en `DataProcessor`. `DataLoader` debe ser capaz de cargar diferentes formatos, debido a la heterogeneidad de los mismos en los datasets seleccionados. Es notable mencionar el caso de `Aposemat IoT-23` donde los datos se presentan en ficheros Zeek con la particularidad de que las etiquetas fueron añadidas a mano, y no respetan el separador (tabulador) del resto del archivo, haciendo su carga algo más compleja.

Es muy importante, con el fin de evitar la fuga de datos (data leakage) que no se realice la normalización, el encoding no determinista ni la selección de características antes de la división en conjuntos de entrenamiento y test. Es por ello que `DataProcessor` será compatible con los transformadores de `sklearn`, haciendo del preprocesamiento un paso integrable en una pipeline que podrá ejecutarse dentro de cada división de entrenamiento.

La pipeline tratará un dataset ya cargado y con los duplicados eliminados y realizará un preprocesamiento, posteriormente seleccionará las mejores características empleando

diversas técnicas y, por último, probará un modelo.

Se ha decidido que se emplearán ANOVA y Kruskal-Wallis (paramétrico y no paramétrico respectivamente) para la selección de características, pudiendo comprobar así si ambos métodos coinciden en su valoración.

El método de testing será Nested Stratified K-Fold con GridSearch, es decir, se seleccionan los modelos a probar junto a un conjunto de sus posibles hiperparámetros. Se realizan dos bucles:

- El bucle interior: El conjunto de entrenamiento se divide en folds con una representación adecuada de las clases y se prueban los hiperparámetros para encontrar aquellos que mejor funcionan.
- El bucle exterior: El resto del conjunto también se divide en folds que serán conjuntos de test.

Con la mejor configuración de hiperparámetros se entrena el modelo en el conjunto de entrenamiento y se prueba con cada conjunto de test, de esta forma se puede demostrar su validez en sets nunca vistos.

En cuanto a las métricas de evaluación, y como se ha establecido anteriormente, las 3 principales serán AUPRC, MCC y la tasa de falsos negativos, las 3 secundarias ROC-AUC, F1 y Brier y por último el tamaño del modelo y métricas relativas a los tiempos.

Respecto a las 3 métricas principales, el área bajo la curva de precisión-recall ofrece resultados más fiables que ROC-AUC cuando existe desbalanceo de clases. El coeficiente de correlación de Matthews aúna TP, TN, FP y FN en una sola cifra y es más robusta que accuracy cuando las clases están desbalanceadas. Por último, la tasa de falsos negativos viene derivada del concepto "better safe than sorry" que en ciberseguridad indica que es mejor preocuparse en exceso por tráfico benigno que ignorar tráfico malicioso.

Por último, por comodidad, se han dividido los modelos que se probarán en 3 grupos que no coinciden con la división ML/DL. Por un lado, los considerados como modelos clásicos que serán XGBOOST, RandomForest y la red neuronal MLP. Por otro lado, las redes neuronales recurrentes LSTM y GRU. Y por último, otros modelos de aprendizaje profundo como autoencoders y transformadores.

Aposemat IoT-23

La carga del primero de los ficheros de Aposemat IoT-23 seleccionados resulta en un dataframe de 1.008.748 registros. El primer paso es una validación de la integridad del conjunto empleando `.info()`. Esta función ha sido crucial para detectar el error en el formato del separador de las columnas de etiquetas. A continuación, se separan las features del objetivo, en este caso, se busca distinguir un ataque de una conversación benigna, por lo que se convierte label a 0 si es benigno y a 1 si es malicioso. Además, al ser este dataset uno de flujos, deben evitarse todas las características (como IPs o puertos) que puedan condicionar al modelo enseñándole patrones generados artificialmente.

Es imprescindible evaluar el balanceo, como se ha mencionado previamente, pues no

hacerlo resulta en la necesidad de tener que aplicar técnicas que ayuden al entrenamiento de los modelos. En el caso del primer dataset se encuentra que el tráfico benigno (47%) y el malicioso (53%) están bastante equilibrados.

Cuando se realizó el testing con el primer fichero se observó que los 3 modelos clásicos presentaban unos resultados muy altos y prácticamente idénticos. El caso del primer fichero es menos notable, pues, aunque no se clasifica erróneamente ninguna Botnet, existe un número considerable de falsos positivos que mantienen la AUPRC y MCC en valores en torno a 0.91 o 0.92 pese a tener una F1 Score de 0.96. Esto indica que, aunque la precisión es bastante buena, no es perfecta, además, el modelo no se encuentra perfectamente calibrado y podría fallar para otros umbrales. Este fenómeno ocurre también con el segundo fichero de Aposemat de forma más notable, donde se obtienen valores de AUPRC perfectos, un correlación de Matthews superior a 0.97 y ningún falso negativo.

Es importante también observar como ambos sistemas de selección de características optaron por `proto_udp` (primer fichero) y por `conn_state_OTH`, `conn_state_S0` y `proto_icpm` en el segundo fichero. Las dos relativas a protocolos requieren una inspección ligera de cabeceras IP/TCP/UDP mientras que las relativas al estado de conexión se obtienen revisando los flags TCP realizando una stateful inspection. Como se ha mencionado en secciones previas del presente trabajo, es preferible evitar características del tráfico no relativas al contenido de los propios paquetes. Es por ello que, aunque en este caso las características de estado de conexión han ofrecido resultados prácticamente perfectos, se prefiere evitar su uso.

Por todo esto, la complejidad de los ficheros seleccionados del dataset no parece muy elevada y se realiza, para validar esta hipótesis, una prueba similar con un modelo sencillo como es el caso de una regresión logística simple. Este modelo muestra unos resultados perfectos en el segundo fichero y mantiene F1 scores de aproximadamente 0.95 en el primero.

Si se observa el tipo de ataque malicioso presente en ambos ficheros puede notarse como los registros de Botnet son parte de un ataque de escaneo de puertos. Es importante este dato, pues estos ataques son especialmente detectables mirando las features anteriormente mencionadas ya que se generan muchos estados particulares, como SYN sin respuesta (`conn_state_S0`), ping sweeps y respuestas del tipo ICMP port unreachable. Es decir, son features que simplifican mucho la detección, haciéndola un problema casi trivial.

Se repite por tanto el proceso empleando el mismo dataset y los mismos modelos pero con 2 modificaciones. Por un lado, se toma la configuración de hiperparámetros de cada modelo ya seleccionada por el GridSearch y, por otro lado, se le permite al selector de características tomar solo aquellas que no implican ningún tipo de inspección de paquetes, es decir, las relativas al número de bytes, de paquetes y duración de cada flujo.

Al realizar este cambio los resultados varían notablemente y se observan diferencias entre los modelos, pudiendo así realizar una comparativa.

N-BaIoT

6.2 Sistema Completo de Detección

Primero, debe levantarse el sistema completo mediante docker compose up, de tal forma que los servicios se encuentran desplegados y operativos.

```
joel@fedora-joel:~/MasterIoT/deteccion-botnets-tfm$ sudo docker ps
```

CONTAINER ID	IMAGE	NAMES	COMMAND	CREATED	STATUS	PORTS
2acb452b8bb6	deteccion-botnets-tfm-traffic_analyzer	traffic_analyzer	"uvicorn main:app --..."	About a minute ago	Up About a minute	0.0.0.0:8000->8000/tcp, [::]:8000->8000/tcp
4d4ccb151574	deteccion-botnets-tfm-traffic_visualizer	traffic_visualizer	"streamlit run main..."	About a minute ago	Up About a minute	0.0.0.0:5000->5000/tcp, [::]:5000->5000/tcp,
0.0.0.0:9100->9100/tcp, [::]:9100->9100/tcp						
65010c9fc3a5	grafana/grafana	grafana	"/run.sh"	About a minute ago	Up About a minute	0.0.0.0:3000->3000/tcp, [::]:3000->3000/tcp
bc67b975b3ff	deteccion-botnets-tfm-traffic_gatherer	traffic_gatherer	"uvicorn main:app --..."	About a minute ago	Up About a minute	0.0.0.0:9000->9000/tcp, [::]:9000->9000/tcp
1550e2571ffc	prom/prometheus	prometheus	"/bin/prometheus --c..."	About a minute ago	Up About a minute	0.0.0.0:9090->9090/tcp, [::]:9090->9090/tcp

Figure 6.1: Los contenedores se encuentran operativos

El primer paso del proceso sería la recogida de tráfico, por tanto, el usuario debería realizar una solicitud POST al servicio de recogida indicando la duración de la misma.

```
1 curl -X POST http://localhost:9000/gather -H "x-token: DUMMY\
2 _TOKEN" -H "Content-Type: application/json" -d '{"duration":
60\}'
```

Listing 6.1: Ejemplo de llamada curl

El proceso se ejecuta correctamente y genera los 3 ficheros esperados, por un lado el PCAP del tráfico capturado, por otro lado el CSV en bruto de los flujos y, por último, los flujos procesados con las métricas deseadas (similarmenete a las descritas en Narang et al. (2014)).

```
traffic_gatherer INFO: Iniciando captura de tráfico durante 60 segundos...
traffic_analyzer INFO: 172.18.0.3:40870 - "GET /metrics HTTP/1.1" 200 OK
traffic_gatherer INFO: 172.18.0.3:56046 - "GET /metrics HTTP/1.1" 200 OK
traffic_analyzer INFO: 172.18.0.3:35966 - "GET /metrics HTTP/1.1" 200 OK
traffic_gatherer INFO: 172.18.0.3:49830 - "GET /metrics HTTP/1.1" 200 OK
traffic_gatherer INFO: tshark terminó correctamente (timeout o éxito)
traffic_gatherer INFO: Running as user "root" and group "root". This could be dangerous.
traffic_gatherer Capturing on 'any'
traffic_gatherer tshark: Promiscuous mode not supported on the "any" device.
traffic_gatherer 51 packets captured
traffic_gatherer
traffic_gatherer Running as user "root" and group "root". This could be dangerous.
traffic_gatherer INFO: Processing tshark output...
traffic_gatherer INFO: Converted timestamps and IPs to human-readable format.
traffic_gatherer INFO: Grouping conversations by source, destination, and protocol...
traffic_gatherer INFO: Grouped conversations into 5 unique pairs of source and destination IPs.
traffic_gatherer INFO: Processed 5 flows with 6 summary entries.
traffic_gatherer INFO: Converted flows summary to DataFrame.
traffic_gatherer INFO: Saved flows summary to CSV.
```

Figure 6.2: Captura correcta de tráfico

La captura en la red en la que se ha hecho esta prueba no incluye ninguna Botnet, por tanto, el resto del proceso empleará un fichero CSV de uno de los datasets de prueba.

Conclusiones y Trabajo Futuro

Conclusiones del trabajo y líneas de trabajo futuro.

Antes de la entrega de actas de cada convocatoria, en el plazo que se indica en el calendario de los trabajos de fin de máster, el estudiante entregará en el Campus Virtual la versión final de la memoria en PDF. En la portada de la misma deberán figurar, como se ha señalado anteriormente, la convocatoria y la calificación obtenida. Asimismo, el estudiante también entregará todo el material que tenga concedido en préstamo a lo largo del curso.

Chapter 8

Conclusions and Future Work

Conclusions and future lines of work. This chapter contains the translation of Chapter 7.

Bibliografía

Amoo, O. O., Osasona, F., Atadoga, A., Ayinla, B. S., Farayola, O. A., and Abrahams, T. O. (2024). Cybersecurity threats in the age of iot: A review of protective measures. *International Journal of Science and Research Archive*, 11(01):1304–1310.

Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012). Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, page 13–16, New York, NY, USA. Association for Computing Machinery.

Buduma, N. and Locascio, N. (2017). *Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms*. O'Reilly Media, 1 edition.

Doriguzzi-Corin, R., Millar, S., Scott-Hayward, S., Martinez-del Rincon, J., and Siracusa, D. (2020). Lucid: A practical, lightweight deep learning solution for ddos attack detection. *IEEE Transactions on Network and Service Management*, 17(2):876–889.

Doshi, R., Apthorpe, N., and Feamster, N. (2018). Machine learning ddos detection for consumer internet of things devices. In *2018 IEEE Security and Privacy Workshops (SPW)*, page 29–35. IEEE.

Elsayed, N., ElSayed, Z., and Bayoumi, M. (2023). Iot botnet detection using an economic deep learning model.

Ergul Aydin, Z. and Kamisli Ozturk, Z. (2021). Performance analysis of xgboost classifier with missing data.

Ferrag, M. A., Maglaras, L., Ahmim, A., Derdour, M., and Janicke, H. (2020). Rdtids: Rules and decision tree-based intrusion detection system for internet-of-things networks. *Future Internet*, 12(3).

Foreman, J., Waters, W. L., Kamhoua, C. A., Hemida, A. H. A., Acosta, J. C., and Dike, B. C. (2024). Detection of hacker intention using deep packet inspection. *Journal of Cybersecurity and Privacy*, 4(4):794–804.

- Garcia, S., Parmisano, A., and Erquiaga, M. J. (2020). IoT-23: A labeled dataset with malicious and benign IoT network traffic. Data set.
- Ibrahim, M. H., Sayagh, M., and Hassan, A. E. (2021). A study of how docker compose is used to compose multi-component systems. *Empirical Software Engineering*, 26(6).
- Javeed, D., Saeed, M. S., Ahmad, I., Kumar, P., Jolfaei, A., and Tahir, M. (2023). An intelligent intrusion detection system for smart consumer electronics network. *IEEE Transactions on Consumer Electronics*, 69(4):906–913.
- Koroniotis, N., Moustafa, N., Sitnikova, E., and Turnbull, B. (2019). BoT-IoT Dataset. Data set.
- McDermott, C., Haynes, W., and Petrovksi, A. (2018). Threat detection and analysis in the internet of things using deep packet inspection. *International Journal on Cyber Situational Awareness*, 4:61–83.
- Meidan, Y., Bohadana, M., Mathov, Y., Mirsky, Y., Shabtai, A., Breitenbacher, D., and Elovici, Y. (2018). N-baiot: Network-based detection of iot botnet attacks using deep autoencoders. *IEEE Pervasive Computing*, 17(3):12–22.
- Moubayed, A., Injadat, M., and Shami, A. (2020). Optimized random forest model for botnet detection based on dns queries.
- Moustafa, N., Creech, G., and Slay, J. (2021). TON_IoT Dataset. Data set.
- Narang, P., Ray, S., Hota, C., and Venkatakrishnan, V. (2014). Peershark: Detecting peer-to-peer botnets by tracking conversations. In *2014 IEEE Security and Privacy Workshops*, pages 108–115.
- Nazir, A., He, J., Zhu, N., Wajahat, A., Ma, X., Ullah, F., Qureshi, S., and Pathan, M. S. (2023). Advancing iot security: A systematic review of machine learning approaches for the detection of iot botnets. *Journal of King Saud University - Computer and Information Sciences*, 35(10):101820.
- Ouhssini, M., Karim, A., Agherrabi, E., Akouhar, M., and Abarda, A. (2024). Deepdefend: A comprehensive framework for ddos attack detection and prevention in cloud computing. *Journal of King Saud University - Computer and Information Sciences*, 36:101938.
- Richards, M. and Ford, N. (2020). *Fundamentals of Software Architecture: An Engineering Approach*. O’Reilly Media.
- Sarkar, V., Maharana, S. K., Kumar, J. D., and Bansa, N. (2025). Edge computing for iot: A survey on architectures, technologies, and applications. *Harbin Engineering University Journal*, 46(6):145–148.

Snehi, M., Bhandari, A., and Verma, J. (2024). Foggier skies, clearer clouds: A real-time iot-ddos attack mitigation framework in fog-assisted software-defined cyber-physical systems. *Computers & Security*, 139:103702.

Song, W., Beshley, M., Przystupa, K., Beshley, H., Kochan, O., Pryslupskyi, A., Pieniak, D., and Su, J. (2020). A software deep packet inspection system for network traffic analysis and anomaly detection. *Sensors*, 20(6):1637.

Watanabe, N., Yamazaki, T., Miyoshi, T., Yamamoto, R., Nakahara, M., Okui, N., and Kubota, A. (2024). Self-adaptive traffic anomaly detection system for iot smart home environments. *arXiv preprint arXiv:2403.02744*.

