Universidad Zaragoza
1542

Trabajo Fin de Máster

Detección de *botnets* en escenarios de IoT mediante técnicas de Inteligencia Artificial

Botnet detection in IoT scenarios using Artificial Intelligence techniques

Autora

María Rodríguez García

Directores

José García Moros

Álvaro Alesanco Iglesias

Máster Universitario en Ingeniería de Telecomunicación

Escuela de Ingeniería y Arquitectura

2024

Agradecimientos

En primer lugar, quiero mostrar mi agradecimiento a mis directores, José García y Álvaro Alesanco, por su implicación, entusiasmo, confianza y cercanía, y por permitirme aprender tanto de vosotros. Gracias por ser mis guías en esta etapa de mi vida. Asimismo, agradezco el apoyo brindado por el Instituto de Investigación en Ingeniería de Aragón a través del Programa de Iniciación a la Investigación. También me gustaría agradecer a todos esos profesores y profesoras que, con su esfuerzo y dedicación, me han enseñado y acompañado durante mi trayectoria académica. En último lugar, quiero dar las gracias a toda mi familia por acompañarme y apoyarme, y a Fran, por estar siempre a mi lado cuando lo necesito. Gracias a mis amigos Pedro y Álvaro, que, a pesar de mis ausencias, siguen ahí. Y a Lorena, por sen tan buena compañera.

Detección de *botnets* en escenarios de IoT mediante técnicas de Inteligencia Artificial

RESUMEN

El objetivo del Trabajo Fin de Máster (TFM) es evaluar la efectividad de diversos algoritmos de clasificación en la detección de tráfico malicioso en el entorno de Internet de las Cosas (IoT) utilizando conjuntos de datos específicos que incluyen tanto tráfico benigno como diversos tipos de ataques. Además, este trabajo muestra y discute el análisis multi-base de datos y examina cómo la heterogeneidad de estas bases afecta los resultados conjuntos complicando su aplicabilidad final.

En este trabajo fin de máster se ha analizado el entorno IoT actual, abarcando desde los dispositivos más utilizados hasta los más atacados, así como las amenazas más comunes, con un enfoque especial en la *botnet* Mirai y sus ataques típicos. Se han estudiado diversos conjuntos de datos IoT, seleccionando IoTD20, IoT-23 y CIC-IoT-2023, que incluyen tráfico benigno y ataques como DDoS, fuerza bruta y escaneo de puertos.

Se ha propuesto un exhaustivo banco de pruebas para evaluar nueve algoritmos de clasificación (*Decision Tree, Gaussian Naive* Bayes, Bernouilli *Naive* Bayes, *Stochastic Gradient Descent*, *Random Forest*, *Bagging* con *Decision Tree, AdaBoost* con *Decision Tree, NearestCentroid* y *Multilayer Perceptron*) que permitan diferenciar entre tráfico benigno y malicioso. Se utilizó la herramienta Zeek para extraer y etiquetar la información de los flujos de tráfico. Las librerías *Scikit-learn,* Pandas y Dask se emplean para el preprocesado y análisis de datos.

Las pruebas se han realizado en clasificación binaria y multiclase, demostrando que los algoritmos basados en árboles de decisión (*Decision Tree, Random Forest, Boosting Tree* y *Bagging Tree*) resultaron ser los más eficientes, alcanzando valores F1 superiores a 0.99 en la evaluación individual de los *datasets* y a 0.9 en la mayoría de las pruebas de evaluación con *datasets* combinados. Los resultados indican que los modelos entrenados con datos distintos al conjunto de evaluación son subóptimos, resaltando la necesidad de incluir datos y ataques variados para obtener resultados más realistas y eficientes en la detección de *botnets*.

Los resultados obtenidos en este trabajo subrayan la importancia de usar *datasets* adecuados para evaluar la efectividad de los modelos de *Machine Learning* en la detección de *botnets*, asegurando su aplicabilidad en entornos reales y diversos.

Índice

Índice de figuras

Índice de tablas

LISTA DE ACRÓNIMOS

AA: Aprendizaje Automático
AB: *Adaptative Boosting*
ACK: *Acknowledgement flag*
APT: *Advanced persistent threat*
ARC: Argonaut RISC Core
ARP: *Address Resolution Protocol*
ASCII: *American Standard Code for Information Interchange*
BBDD: Bases de datos
BPS: *Bytes per second*
C2: *Command And Control*
CAIDA: *Center for Applied Internet Data Analysis*
CART: *Classification and Regression Trees*
CDX: *Cyber Defense Exercise*
CFS: *Correlation-based Feature Subset Selection*
CIC: *Canadian Institute for Cybersecurity*
CPU: *Central Processing Unit*
CWR: *Congestion Window Reduced*
DARPA: *Defense Advanced Research Projects Agency*
DDoS: *Distributed Denial of Service*
DGA: *Domain Generation Algorithm*
DL: *Deep Learning*
DoS: *Denial of Service*
DNS: *Domain Name System*
DVR: *Digital Video Recorder*
ECE: *Explicit Congestion Notification*
FIN: *Finish flag*
FN: *False Negative*
FP: *False Positive*
FS: *Feature Selection*
FTP: *File Transfer Protocol*
GRE: *Generic Routing Encapsulation*
HTTP: *Hypertext Transfer Protocol*
IA: Inteligencia Artificial
IAT: *Interarrival Time*
ICMP: *Internet Control Message Protocol*
IDS: *Intrusion Detection System*
IoMT: *Internet of Medical Things*
IoT: *Internet of Things*
IP: *Internet Protocol*
IRC: *Internet Relay Chat*
KNN: *K Nearest Neighbors*
LAN: *Local Area Network*
LBNL: *Lawrence Berkeley National Laboratory*
ML: *Machine Learning*
MLP: *Multilayer Perceptron*
NAS: *Network-Attached Storage*
NIDS: *Network* IDS
OOBM: *Out-Of-Band Management*
OS: *Operative System*
OT: *Operational Technology*
OVA: *One Versus All*
PSH: Push *flag*
POP3: *Post Office Protocol version* 3
PPS: *Packets per second*
QP: *Quadratic Programming*
R2L: *Remote to Local*
RAM: *Random Access Memory*
ROC: *Receiver Operating Characteristic*
RDP: *Remote Desktop Protocol*
RST: Reset *flag*
S-IDS: *Signature based* IDS
SGD: *Stochastic Gradient Descent*
SMB: *Server Message Block*
SMO: *Sequential Minimal Optimization*
SSH: *Secure Shell*
SSL: *Secure Socket Layer*
STA: Estación
SVM: *Support Vector Machine*
SYN: Synchronization *flag*
SQL: *Structured Query Language*
TCP: *Transmission Control Protocol*
TFM: Trabajo Fin de Máster
TLS: *Transport Layer Security*
TP: *True Positive*
TPR: *True Positive Rate*
TS: *True Negative*
UDP: *User Datagram Protocol*
U2R: *User to Root*
URG: *Urgent Flag*
VSE: *Valve source engine*
XSS: *Cross Site Scripting*

1. Introducción

En este capítulo se presenta la temática que aborda el trabajo, comenzando con una breve revisión del estado del arte de los sistemas de detección de intrusos en entornos *IoT (Internet of Things)* basados en la utilización de técnicas de Inteligencia Artificial (en cuyos aspectos más importantes se irá profundizando a lo largo de la memoria), y los principales objetivos que se pretenden cumplir. A continuación, se describe el contexto en que se ha llevado a cabo el trabajo y se muestra un cronograma del mismo. Finalmente se indica cómo se ha organizado la memoria.

1.1. Estado del arte

Hoy en día, la preocupación por la seguridad en las redes de comunicaciones y sistemas de información es evidente, lo que ha llevado al desarrollo de nuevas técnicas tanto preventivas como reactivas para abordar este problema. En este contexto, surgieron los Sistemas de Detección de Intrusiones (IDS), cuyo objetivo es identificar actividades maliciosas en redes y sistemas. Las técnicas de aprendizaje automático (*Machine Learning*, ML) y aprendizaje profundo (*Deep Learning*) son particularmente útiles en este ámbito, ya que pueden automatizar la detección de ataques y diferenciar entre diversos tipos de tráfico gracias a la inteligencia artificial.

La seguridad en el ámbito de Internet de las Cosas (IoT) y en redes domésticas y de trabajo también se ha vuelto crucial. Los dispositivos IoT, debido a su interconexión y susceptibilidad a ataques, requieren medidas de seguridad especializadas. Los IDS aplicados a entornos IoT utilizan avanzados algoritmos de ML y *Deep Learning* para analizar grandes volúmenes de datos en tiempo real, identificar patrones anómalos y detectar actividades maliciosas antes de que causen daños significativos.

Recientemente, se han creado nuevos conjuntos de datos (*datasets*) que incluyen tanto tráfico benigno como diversos tipos de ataques en entornos IoT, los cuales se utilizan como bancos de prueba para los sistemas de detección de intrusos. Estos *datasets* permiten evaluar el rendimiento de los algoritmos y técnicas de ML que sustentan los IDS propuestos. Varios estudios previos ofrecen revisiones sistemáticas sobre la detección de ataques de ciberseguridad en el escenario de IoT, utilizando diferentes métodos de inteligencia artificial, incluyendo técnicas de aprendizaje profundo (DL) y de aprendizaje automático (ML (da Costa et al., 2019; Zarpelão et al., 2017)). El trabajo en (Zarpelão et al., 2017) presenta una revisión sobre los avances en IDS para IoT, identificando las principales tendencias, problemas abiertos y posibles líneas de investigación futuras. Además, el trabajo en (da Costa et al., 2019) presenta una revisión de técnicas de ML aplicadas en IoT para la detección de intrusiones. En los capítulos 2 y 3, donde se detallan los materiales y métodos de este trabajo, se profundiza en el estado del arte específico de cada tecnología utilizada.

1.2. Objetivos

El objetivo principal de este trabajo es el estudio y análisis de diversos conjuntos de datos de tráfico de comunicaciones en entornos IoT, centrándose en la identificación de ataques de *botnets* como Mirai y Gafgyt, mediante la aplicación de técnicas de ML.

Además, los objetivos específicos del TFM son:

- Realizar un estudio detallado e individualizado de los conjuntos de datos, extrayendo parámetros comunes para poder realizar una evaluación conjunta.
- Realizar un estudio a partir de la unión de los tres conjuntos seleccionados, y obtener métricas detalladas, para posteriormente estudiar el funcionamiento de los modelos con datos provenientes de diferentes conjuntos de datos.
- Estudiar el rendimiento y escalabilidad de los modelos de clasificación al ser entrenados mediante un conjunto de datos diferente al empleado en la evaluación, analizando su comportamiento frente a ataques similares extraídos de diversos conjuntos de datos, pero empleando parámetros comunes.

1.3. Contexto

Este trabajo se enmarca en la línea de investigación de ciberseguridad del departamento de Ingeniería Electrónica y Comunicaciones, específicamente del grupo de investigación *Communications Networks and Information Technologies* (CeNIT) del Instituto de Investigación en Ingeniería de Aragón (I3A). Tiene relación directa con la materia de formación obligatoria de "Redes y servicios" del Máster en Ingeniería de Telecomunicación.

En este TFM se aborda el estudio y comparación de diversos conjuntos de datos de tráfico de comunicaciones en entornos IoT, centrándose en la identificación y detección de ataques de *botnets*. Las técnicas de *Machine Learning* evaluadas en este estudio corresponden a la categoría de aprendizaje automático supervisado. Este trabajo se ha llevado a cabo en el marco de una beca de investigación concedida por el I3A, que permite iniciarse en tareas de investigación vinculadas con los estudios y facilita una futura orientación profesional o investigadora. Además, se enmarca en las tareas de investigación del proyecto “Optimización de redes WLAN coordinadas de última generación basadas en arquitecturas programables y virtualizadas (NeWLAN)” PID2022-136476OB-I00.

1.4. Cronograma

En la Figura 1 se presenta de forma gráfica la organización temporal del trabajo. Por orden cronológico, se distribuyó en las siguientes etapas:

- Estudio del estado del arte y búsqueda de bases de datos para su evaluación. 5 semanas.
- Adaptación de las bases de datos. 3 semanas.
- Obtención de nuevos parámetros comunes con Zeek y etiquetado de flujos de tráfico. 4 semanas.
- Aplicación de técnicas de ML y DL sobre las bases de datos. 4 semanas.
- Evaluación y discusión de los resultados obtenidos. 4 semanas.
- Redacción de la memoria. 4 semanas.

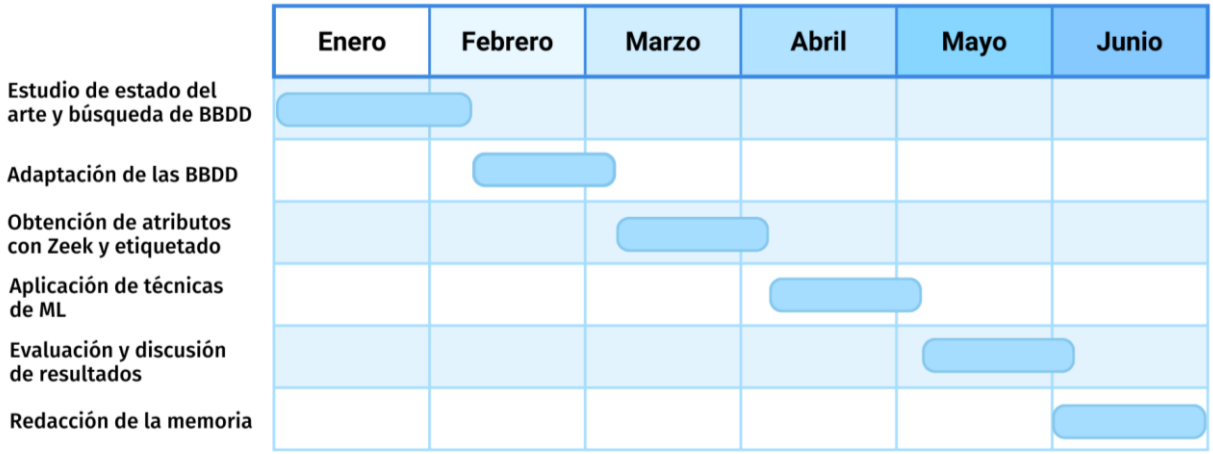


Figura 1: Diagrama de Gantt.

1.5. Estructura del documento

El documento se organiza de la siguiente forma:

- En el **capítulo 1** se incluye la **Introducción**, donde se presenta el estado del arte, y se han planteado los objetivos principales del trabajo, contexto y cronograma.
- En el **capítulo 2** se realiza un **Análisis exhaustivo del entorno IoT**, donde se introducen los dispositivos IoT más usados, los dispositivos atacados con mayor frecuencia, los ataques más comunes (entrando en detalle en la *botnet* Mirai), así como los principales conjuntos de datos.
- En el **capítulo 3** se introducen las **Técnicas de *Machine Learning***, desarrollando en mayor profundidad aquellas empleadas en este estudio, junto con los indicadores de rendimiento utilizados para evaluar los resultados.
- En el **capítulo 4** se presenta el **Sistema de detección de *botnets***, que abarca todo el proceso de transformación de las capturas de tráfico de los *datasets* originales mediante la herramienta Zeek y diferentes librerías de Python para poder transformar los conjuntos de datos, eliminar flujos de datos erróneos, asignarles etiquetas y unificar los atributos. En este capítulo se desarrolla la metodología empleada y se analiza la estructura de los datos transformados.
- En el **capítulo 5** se presenta la **Aplicación de técnicas de ML sobre los conjuntos de datos**. Se describe la metodología empleada para la clasificación, seguida de la presentación y discusión de los resultados obtenidos para los dos escenarios de pruebas planteados: por un lado, análisis de los tres conjuntos de datos de forma individual con clasificación multiclase, y por otro lado, análisis empleando los tres conjuntos de datos de forma combinada. Para este segundo escenario de pruebas, en una primera evaluación se unieron los tres *datasets* y se generaron los conjuntos de *train* y *test*, realizando clasificación binaria y multiclase; y en la segunda prueba se utilizó como conjunto de *test* un *dataset* diferente al de evaluación, también de forma binaria y multiclase.
- Finalmente, en el **capítulo 6** se abordan las **Conclusiones** más relevantes de este TFM y posibles **líneas futuras**.

2. Análisis del entorno IoT

2.1. Dispositivos IoT más atacados

En el ámbito de Internet de las Cosas, los dispositivos conectados a la red pueden estar expuestos a ataques debido a diversas razones: *firmware* desactualizado, sistemas de autenticación débiles, protocolos de comunicación inseguros o configuraciones de red inseguras. Es importante destacar que muchos usuarios de estos dispositivos no son conscientes de estas vulnerabilidades. Según el informe (Cybersecurity Report 2023: Consumer Devices Under Threat, 2022), el 67% de los hogares sufren una amenaza online al mes. Además, algunos fabricantes no invierten suficiente esfuerzo en securizar adecuadamente los productos. Los atacantes, conscientes de estas deficiencias, aprovechan estas brechas de seguridad para acceder o comprometer los dispositivos con fines maliciosos. Dentro de las marcas de dispositivos más atacados, encontramos *Hikivision* en primer puesto, seguida de *D-Link* y de *Apple*. Sin embargo, es necesario mencionar que el número de dispositivos influye significativamente en esta estadística.

El ecosistema de Internet de las Cosas consta de miles de millones de dispositivos conectados a Internet, y estos se presentan en formas muy variadas, desde electrodomésticos, impresoras, hasta cámaras IP y sensores. Como se puede apreciar en la Figura 2, según el mismo informe (Cybersecurity Report 2023: Consumer Devices Under Threat, 2022), siete tipos de dispositivos concentran el 90% de las amenazas: ordenadores y teléfonos móviles componen aproximadamente el 46%, mientras que las cámaras IP, a pesar de representar solo el 1.2% de los dispositivos, sufren un 24% de los ataques.

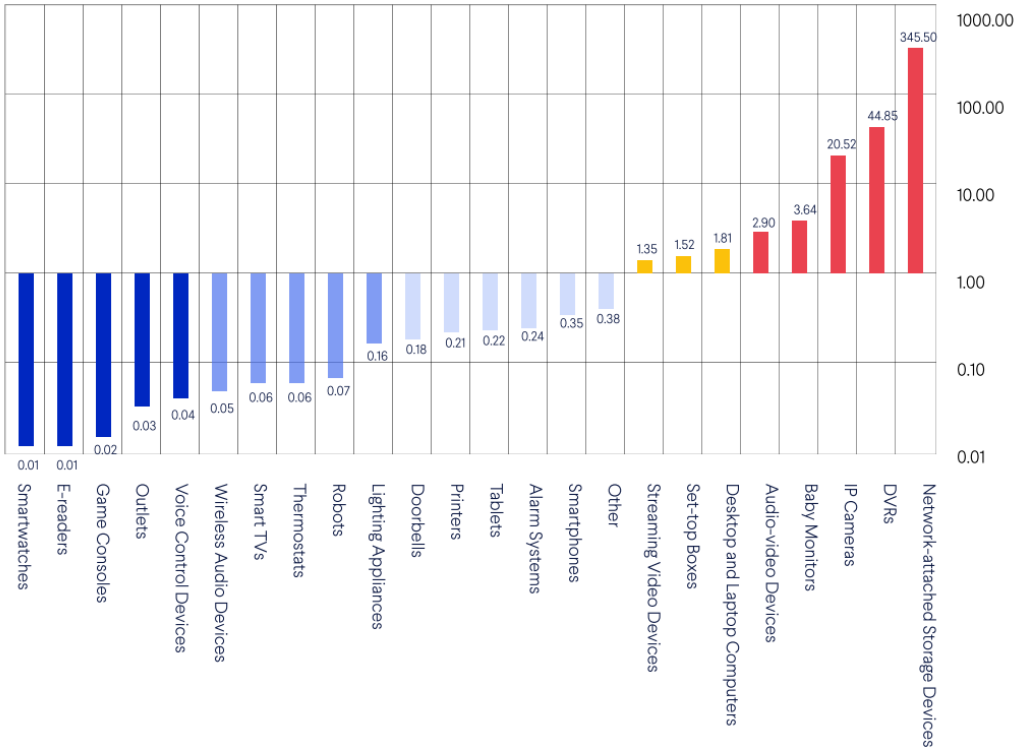


Figura 2: Índice de amenaza por tipo de dispositivo.

Con esto se concluye que los dispositivos que se encuentran bajo mayor riesgo de amenaza son: en primer lugar los *NAS (Network-Attached Storage)* o dispositivos de almacenamiento conectados a la red, ya que necesitan tener determinados puertos abiertos para su funcionamiento, seguidos de los *DVR (Digital Video Recorders)* o dispositivos de grabación de vídeo digital, cuya función es grabar el vídeo proveniente de elementos como cámaras IP, y poseen una configuración de fábrica muy poco segura, a continuación encontramos las cámaras IP, conocidas por la poca seguridad que presentan de fábrica, pueden convertirse fácilmente en parte de una *botnet*. Finalmente, también cabe mencionar los monitores de bebé y los dispositivos de audio y vídeo.

La información del informe "(The Riskiest Connected Devices in 2023, 2023)" complementa los datos sobre las vulnerabilidades en dispositivos IoT. Según el informe, los dispositivos IoT que mayor riesgo tienen de ser atacados incluyen cámaras IP, impresoras y VoIP, que suelen estar expuestos en Internet y han sido históricamente blanco de APTs (*Advanced persistent threat*). Además, se identifican dos nuevas entradas problemáticas: los dispositivos NAS y OOBM (*out-of-band management*). Los dispositivos NAS han ganado popularidad entre los actores de *ransomware* debido a los valiosos datos que almacenan y sus numerosas vulnerabilidades. Por otro lado, los dispositivos de gestión *out-of-band* (OOBM) permiten la administración remota de equipos a través de interfaces alternativas, pero enfrentan serias vulnerabilidades críticas, algunas de las cuales han sido explotadas por *malware* sofisticado, incluso hasta finales de 2022. Esta situación subraya la necesidad urgente de mejorar las medidas de seguridad y la concienciación entre fabricantes y usuarios de dispositivos IoT, especialmente en dispositivos específicos como NAS y OOBM, que pueden comprometer redes críticas si no se protegen adecuadamente.

En cuanto a los sistemas operativos utilizados por los dispositivos, el mismo informe indica que predominan los sistemas operativos "tradicionales" como Windows, Linux, Mac y UNIX. Esto incluye varios dispositivos especializados de IoT/OT/IoMT que ejecutan Linux o Windows.

Finalmente, el infome concluye que más de 4000 vulnerabilidades afectan a los dispositivos en el conjunto de datos analizado. De estas vulnerabilidades, el 78% afecta a dispositivos IT, el 14% a dispositivos IoT, el 6% a dispositivos OT y el 2% a dispositivos IoMT. Aunque la mayoría de las vulnerabilidades afectan a dispositivos IT, casi el 80% de estas tienen solo alta severidad. Por otro lado, los dispositivos IoMT tienen menos vulnerabilidades, pero el 80% de ellas son críticas, lo que típicamente permite la toma de control completa de un dispositivo. De manera similar, más de la mitad de las vulnerabilidades que afectan a dispositivos OT e IoT son críticas.

2.2. Amenazas más frecuentes en entornos IoT

Considerando la tendencia hacia la automatización inteligente en todos los ámbitos, los hogares de todo el mundo en 2022 ya poseían dispositivos inteligentes y se espera que esta cifra siga aumentando. Junto con los altavoces inteligentes, otros dispositivos de alta demanda incluyen sistemas de seguridad, grandes y pequeños electrodomésticos, detectores de humo y *hubs* y *gateways*, como se puede observar en la Figura 3.

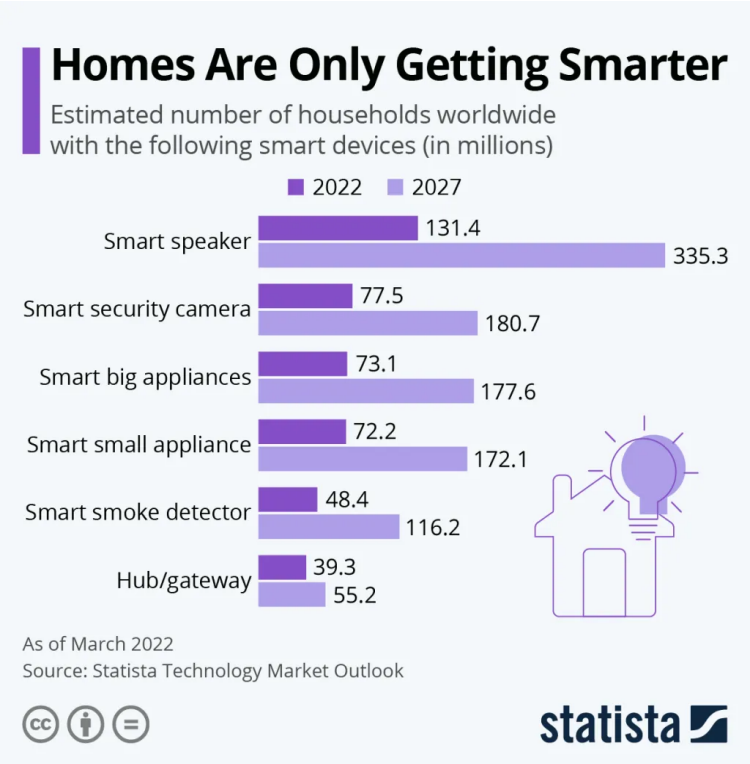


Figura 3: Número de dispositivos en la actualidad y previsión para 2027 (Armstrong, 2022).

El aumento masivo de dispositivos IoT ha ampliado significativamente la superficie de ataque, convirtiéndolos tanto en puntos de acceso inicial como en posibles atacantes. Las amenazas más comunes incluyen:

- **Explotación de Firmware:** Vulnerabilidades en el software de bajo nivel que controla el hardware.
- **Explotación de vulnerabilidades en endpoints conectados a dispositivos IoT:** Debido a sistemas operativos desactualizados o configuraciones inseguras.
- **Ataques de Ransomware:** Especialmente dirigidos a dispositivos IoT como cámaras IP y dispositivos de almacenamiento NAS, debido a los datos valiosos que pueden contener.
- **Hardware no protegido:** Dispositivos IoT con hardware vulnerable que puede ser comprometido fácilmente.
- **Acceso no autorizado a dispositivos IoT:** Debido a la falta de autenticación adecuada y configuraciones de red inseguras.

Además, el 98% del tráfico IoT no está encriptado, exponiendo información personal y confidencial al riesgo de interceptación (Law, 2023). Los puertos abiertos en dispositivos IoT representan uno de los factores de riesgo más críticos debido a su capacidad para exponer vulnerabilidades conocidas y desconocidas, incluyendo *exploits zero-day*. Entre los protocolos más comúnmente explotados en 2022 se encuentran el Protocolo SMB (*Server Message Block*), utilizado por sistemas Windows para compartir archivos y acceder a servicios remotos; el Protocolo RDP (*Remote Desktop Protocol*), que facilita la gestión remota mediante una interfaz gráfica; SSH (*Secure Shell*), empleado para la gestión remota mediante una interfaz de línea de comandos, especialmente en servidores Linux/UNIX y dispositivos IoT; y Telnet, utilizado

principalmente para la gestión remota de dispositivos especializados heredados (The Riskiest Connected Devices in 2023, 2023).

En cuanto a la exposición a Internet, los dispositivos de infraestructura de red IT y los dispositivos de seguridad son los más vulnerables, ya que actúan como el perímetro entre las redes internas y externas. Después de estos, las cámaras IP son los dispositivos más expuestos, representando el 23% del total de exposición, seguidas por dispositivos NAS con un 7% y VoIP con un 3%, según la Figura 4.

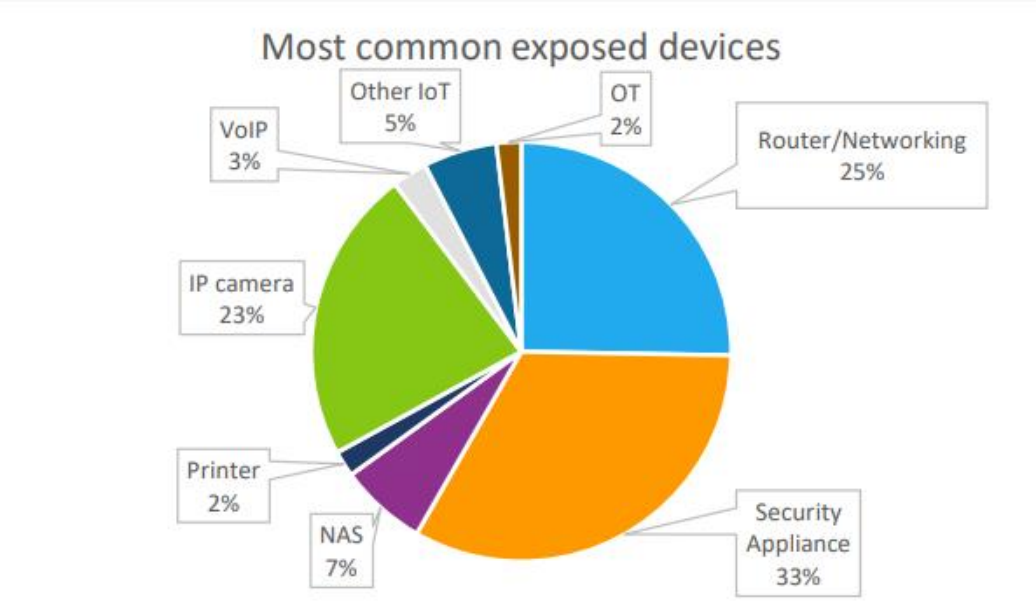


Figura 4: Dispositivos más expuestos (The Riskiest Connected Devices in 2023, 2023).

2.3. *Botnets* en entornos IoT

Las *botnets* representan una amenaza significativa en entornos IoT, consistiendo en redes de dispositivos interconectados, que han sido infectados con *malware* y son controlados de manera remota por un actor o grupo malicioso conocido como "*bot herder*". Según el informe (Altares et al., 2023), entre las familias de *malware* más comunes se destacan variantes como Mirai, Gafgyt, y sus híbridos como Kyton o Keksec, además de otras *botnets* como RapperBot y Zerobot. Estas *botnets* son conocidas por lanzar una variedad de ataques, incluyendo ataques de denegación de servicio distribuido (DDoS), robo de datos, campañas de spam y amenazas persistentes avanzadas (APTs).

El aumento de dispositivos IoT, a menudo con medidas de seguridad deficientes, los convierte en objetivos principales para las infecciones de *botnets*. Estos dispositivos son comprometidos fácilmente explotando procesos de autenticación y autorización débiles, permitiendo al *botherder* controlar una red extensa de dispositivos infectados con propósitos maliciosos. Además, el informe (IoT Botnet activity in Consumer Networks, 2023) revela que las *botnets* suelen aprovechar múltiples *exploits*, con un aumento significativo en el uso de vulnerabilidades recientemente descubiertas, indicando un incremento en la sofisticación y frecuencia de los ataques.

Los tipos de *botnets* varían en función de su arquitectura de comando y control (C&C). Las arquitecturas centralizadas, como las *botnets* IRC y HTTP, dependen de servidores específicos para la comunicación de C&C. Las *botnets* IRC utilizan la red IRC para enviar comandos a los bots, aprovechando su simplicidad, disponibilidad amplia y anonimato. Estas *botnets* contactan con el C&C a través de IRC para registrar nuevos bots y comenzar a recibir órdenes. El método más común para este procedimiento es tomar el control de uno o varios servidores IRC para enviar órdenes a los nodos de la red. Por otro lado, las *botnets* HTTP utilizan servidores web para distribuir comandos a los bots, lo que las hace más difíciles de detectar y bloquear que las *botnets* IRC, ya que su tráfico puede mezclarse con el tráfico web regular y pasar por políticas de firewall existentes (Imam et al., 2014).

Las *botnets* POP3 utilizan protocolos de correo electrónico para la comunicación C&C, donde los bots recuperan comandos de servidores de correo POP3 mediante la descarga de mensajes de correo electrónico que contienen instrucciones adjuntas. Esta forma de comunicación es menos detectable que las *botnets* IRC, proporcionando un canal encubierto efectivo para los comandos maliciosos.

Las *botnets* P2P representan una evolución en la arquitectura de *botnets*, eliminando la necesidad de un servidor centralizado para C&C. En lugar de eso, los bots en una *botnet* P2P se comunican directamente entre sí, utilizando sistemas de publicación/suscripción para distribuir comandos. Esta estructura descentralizada hace que las *botnets* P2P sean más resistentes a los intentos de desmantelamiento y más difíciles de monitorear para los defensores.

Para evitar la detección, los diseñadores de *botnets* suelen utilizar protocolos ampliamente utilizados para su C&C, como IRC, HTTP, POP3 o P2P, y en ocasiones incluso redes sociales en línea. Estos protocolos ofrecen diferentes niveles de anonimato, resistencia y capacidad de mezclarse con el tráfico normal de Internet, dificultando los esfuerzos para mitigar las amenazas de *botnets*.

Además de las diversas arquitecturas de *botnets* mencionadas, los adversarios emplean múltiples técnicas de ofuscación y cifrado para ocultar y proteger las comunicaciones de comando y control (C&C). Estas técnicas incluyen el cifrado de datos utilizando técnicas convencionales como ASCII, Unicode o Base64, así como compresión de datos mediante esquemas como gzip. Para dificultar aún más la detección, los adversarios utilizan ofuscación de datos, que incluye la inserción de datos basura en el tráfico del protocolo, técnicas para ocultar información dentro de archivos de imagen u otros medios digitales, y la impersonación de protocolos válidos para disfrazar las comunicaciones. Además, los adversarios pueden emplear técnicas de resolución dinámica, como el uso de DNS de flujo rápido, algoritmos de generación de dominios (DGAs), y cálculos DNS, para cambiar dinámicamente los dominios, direcciones IP y números de puerto utilizados por la infraestructura de comando y control. Estas estrategias permiten a las *botnets* eludir las detecciones convencionales y adaptarse rápidamente a las contramedidas implementadas (*What Is Command and Control (C&C or C2) in Cybersecurity? - Zenarmor.Com*, 2023).

La generación de una *botnet* consta de tres pasos:

- **Explotación**: Se busca una debilidad para explotar. Esta debilidad podría encontrarse en un sitio web, en el acceso sin protección a una aplicación o en un software mal configurado.

- **Creación de Bots**: Una vez que el dispositivo ha sido infectado, se convierte en un *zombie*, listo para seguir las órdenes del *bot herder*. El *bot herder* repite este proceso una y otra vez.
- **Ataque**: Una vez que han infectado cientos, miles o incluso decenas de miles de dispositivos, se enlazan y lanzan ataques.

2.3.1. Principales amenazas: *Botnet* Mirai

En la sección anterior, se ha discutido cómo las *botnets* pueden explotar dispositivos IoT vulnerables. Un ejemplo destacado de esto es la *botnet* Mirai. A diferencia de otras ciberamenazas, el *malware* Mirai afecta principalmente a dispositivos inteligentes conectados a la red, como *routers*, termostatos, monitores para bebés, frigoríficos, etc. Al apuntar al sistema operativo Linux que muchos dispositivos IoT utilizan, el *malware* Mirai está diseñado para explotar vulnerabilidades en los *gadgets* inteligentes y enlazarlos en una red de dispositivos infectados. Una vez que forman parte de la *botnet*, el hardware es empleado para llevar a cabo ataques adicionales como parte de un enjambre de máquinas *zombies*. Tradicionalmente, las *botnets* se han utilizado para realizar campañas de *phishing* y ataques masivos de spam, pero la naturaleza de los dispositivos IoT hace que las *botnets* Mirai sean ideales para saturar sitios web o servidores mediante ataques DDoS (*Distributed Denial of Service*).

Primero, el *malware* Mirai escanea direcciones IP para identificar dispositivos inteligentes que ejecutan determinadas versiones de Linux en procesadores ARC. Luego, Mirai explota vulnerabilidades de seguridad en el dispositivo IoT para obtener acceso a la red mediante combinaciones de nombre de usuario y contraseña predeterminadas. Si estas configuraciones no se han cambiado o actualizado, Mirai puede iniciar sesión en el dispositivo e infectarlo con *malware* (The Mirai Botnet – Threats and Mitigations). La mayoría de los dispositivos que ataca la *botnet* Mirai son routers domésticos y cámaras, pero casi cualquier dispositivo inteligente es susceptible a las *botnets* IoT. La misma conexión de red que da funcionalidad a las aspiradoras robotizadas, intercomunicadores IP, electrodomésticos de cocina en un hogar inteligente, también es una puerta trasera potencial para el *malware*. En su apogeo en septiembre de 2016, Mirai paralizó temporalmente varios servicios de alto perfil, como OVH, Dyn y Krebs on Security, a través de ataques masivos DDoS. OVH informó que estos ataques superaron 1 Tbps, el más grande registrado públicamente. Lo notable de estos ataques récord es que se llevaron a cabo a través de pequeños y aparentemente inofensivos dispositivos IoT como *routers* domésticos, monitores de calidad del aire y cámaras de vigilancia personales. En su punto máximo, Mirai infectó más de 600000 dispositivos IoT vulnerables (*Inside the Infamous Mirai IoT Botnet: A Retrospective Analysis*, 2017).

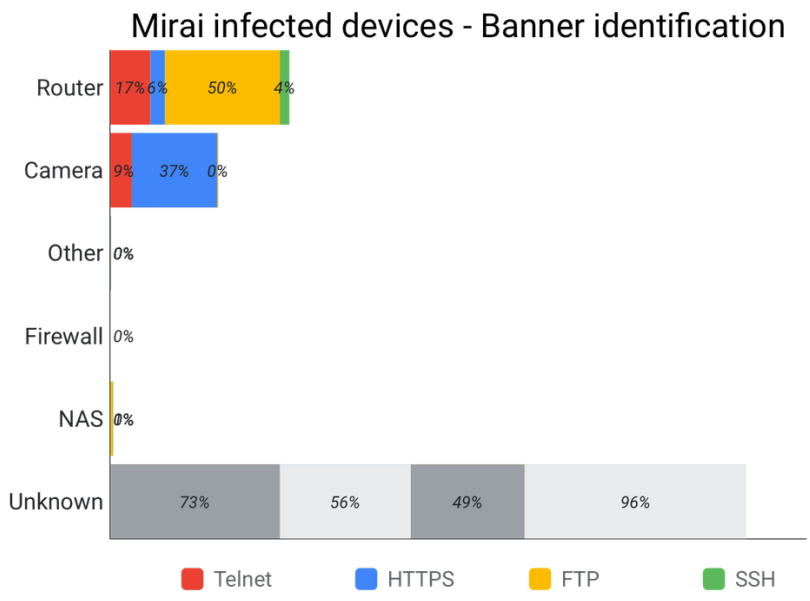


*Figura 5: Tipos de dispositivos infectados por Mirai (*Inside the Infamous Mirai IoT Botnet: A Retrospective Analysis*, 2017).*

Mirai comienza como un gusano auto-replicante, es decir, es un programa malicioso que se replica encontrando, atacando e infectando dispositivos IoT vulnerables. También se considera una *botnet* porque los dispositivos infectados son controlados a través de un conjunto central de servidores de comando y control (C&C). Estos servidores indican a los dispositivos infectados qué sitios atacar a continuación. En general, Mirai está compuesto por dos componentes clave: un módulo de replicación y un módulo de ataque.

El módulo de replicación (véase Figura 6) es responsable de aumentar el tamaño de la *botnet* esclavizando tantos dispositivos IoT vulnerables como sea posible. Lo hace escaneando aleatoriamente todo Internet en busca de objetivos viables y atacándolos. Una vez que se compromete un dispositivo vulnerable, el módulo informa a los servidores C&C para que pueda ser infectado con la última carga útil de Mirai. Para comprometer dispositivos, la versión inicial de Mirai se basó exclusivamente en un conjunto fijo de 64 combinaciones de inicio de sesión/contraseña predeterminadas ampliamente conocidas y comúnmente utilizadas por los dispositivos IoT.

El módulo de ataque es responsable de llevar a cabo ataques DDoS contra los objetivos especificados por los servidores C&C. Este módulo implementa la mayoría de las técnicas de ataque DDoS, como la inundación HTTP, la inundación UDP y todas las opciones de inundación TCP. Esta amplia gama de métodos permitió a Mirai realizar ataques volumétricos, ataques a nivel de aplicación y ataques de agotamiento de estado TCP (*Inside the Infamous Mirai IoT Botnet: A Retrospective Analysis*, 2017).

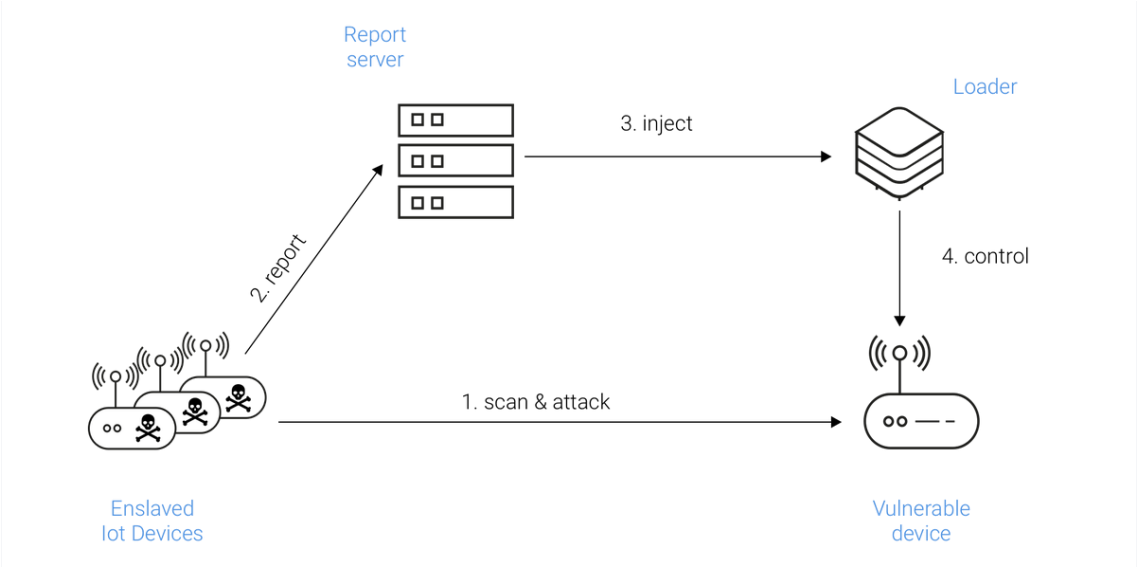


*Figura 6: Módulo de replicación de Mirai (*Inside the Infamous Mirai IoT Botnet: A Retrospective Analysis*, 2017).*

El código fuente de Mirai sigue activo y ha dado lugar a variantes como Okiru, Satori, Masuta y PureMasuta. Por ejemplo, PureMasuta es capaz de explotar la vulnerabilidad HNAP en dispositivos D-Link, mientras que la cepa OMG transforma los dispositivos IoT en *proxies* que permiten a los ciberdelincuentes permanecer en el anonimato. Además, se ha descubierto recientemente una poderosa *botnet* conocida como IoTrooper y Reaper, que es capaz de comprometer dispositivos IoT a un ritmo mucho más rápido que Mirai. Reaper puede dirigirse a un mayor número de fabricantes de dispositivos y tiene un control mucho mayor sobre sus bots *(¿Qué Es La Botnet Mirai? | Cloudflare)*.

Según el informe (Lella et al., 2023), los ataques DDoS se están construyendo cada vez más sobre dispositivos IoT. Los dispositivos y sensores son objetivos adecuados para los ataques DDoS debido a sus recursos limitados que a menudo resultan en una seguridad deficiente. Estos dispositivos son fáciles de corromper, ya que a menudo vienen con configuraciones incorrectas (por ejemplo, contraseñas débiles). La creciente complejidad de estos sistemas móviles hace que la falta de habilidades de seguridad de los usuarios sea cada vez más relevante. Esta tendencia también ha sido confirmada por Microsoft, que observa que los ataques DDoS consistentemente utilizan dispositivos IoT. Varios ataques han adaptado *malware* existente (por ejemplo, Mirai) y *botnets* para involucrar IoT.

Muchos de los vectores de ataque en Mirai están basados en tipos tradicionales de ataques DDoS, pero han sido personalizados y/o mejorados para su uso específico en esta *botnet*. A continuación, se detallan en la Tabla I los diferentes tipos de ataques incluidos en el código original de Mirai, así como algunos que quedaron incompletos. La *botnet* Mirai originalmente estaba diseñada para permitir la multitenencia y el acceso transaccional. Una vez que el servidor de comando y control (C2) y la *botnet* eran establecidos, se podían añadir usuarios adicionales a la plataforma. Esto significaba que el acceso público a la *botnet* era tan sencillo como realizar una transacción comercial (Winward, 2018).

Tabla I: Ataques de Mirai.

Protocolo	Ataque	Perfil Ancho Banda	Descripción
TCP	SYN *Flood*	Moderado BPS, Alto PPS	Clásico SYN *Flood* que envía una cantidad masiva de solicitudes de sincronización.
	ACK *Flood*	Alto BPS, Medio PPS	Inunda con paquetes ACK y causa que genere respuestas RST hasta ser abrumado.
	STOMP *Flood*	Alto BPS, Bajo PPS	Supera ciertas técnicas de mitigación de DDoS. Establece una conexión TCP legítima y luego inunda con paquetes ACK.
	HTTP *Flood*	Bajo BPS, Bajo PPS	Es muy flexible y personalizable. Ejecuta ataques HTTP GET repetidos para agotar los recursos del objetivo.
UDP	UDP *Flood*	Alto BPS, Moderado PPS	Único debido a su capacidad para aleatorizar puertos de origen y destino, haciendo difícil su identificación.
	VSE *Flood*	Medio BPS, Alto PPS	Ataque a servidores que ejecutan juegos de *Valve Corporation*.
	DNS *Flood*	Medio BPS, Alto PPS	Inundación de consultas DNS de subdominios aleatorios dentro del dominio especificado. Envía esta solicitud a su servidor DNS recursivo local.
	UDPPLAIN *Flood*	Alto BPS, Medio PPS	Tiene menos opciones que el ataque UDP normal, permitiendo mayor PPS.
GRE	GREIP *Flood*	Alto BPS, Medio PPS	Interesante por su velocidad y flexibilidad. Encapsula paquetes dentro de GRE. Direcciones IP y puertos aleatorios.
	GREETH *Flood*	Alto BPS, Medio PPS	Paquetes GRE encapsulados con tramas Ethernet transparentes, dificultando distinguirlo. Incluye una trama L2.

2.3.2. Otros ataques: DDoS, fuerza bruta, escaneo de puertos y de sistema operativo.

Otros ataques que aparecen con frecuencia en entornos IoT incluyen DDoS-PSHACK *Flood*, DDoS-RSTFIN *Flood*, *Dictionary Brute Force*, *OS Scan* y *Port Scan,* y representan una variedad de métodos que los atacantes pueden emplear para comprometer la seguridad y funcionalidad de los dispositivos IoT. Todos estos tipos de ataques se incluyen en las bases de datos que se emplearán en el trabajo. En el Anexo I se detalla cada uno de ellos.

Ataque de Fuerza Bruta de Diccionario: Utiliza un diccionario predefinido de palabras comunes y frases para descifrar contraseñas. Automatiza el proceso de introducir y verificar cada palabra del diccionario contra el sistema de autenticación. Es efectivo contra contraseñas débiles o previsibles.

Escaneo de Sistema Operativo (*OS Scan*): Determina el sistema operativo en un dispositivo de red enviando paquetes específicos y analizando las respuestas para identificar patrones característicos de diferentes sistemas operativos. Herramientas como Nmap son comúnmente usadas.

Escaneo de Puertos (*Port Scan*): Identifica qué puertos están abiertos en un dispositivo de red enviando solicitudes a diferentes puertos y analizando las respuestas. Los puertos abiertos revelan servicios activos y posibles puntos de entrada para ataques.

DDoS PSHACK *Flood*: Ataca un servidor enviando una gran cantidad de paquetes TCP con los *flags* PSH y ACK activados, saturando los recursos del servidor y provocando una denegación de servicio. Puede recibir o no un paquete RST en respuesta. *(ACK-PSH Flood | Knowledge Base | MazeBolt)*

DDoS RSTFIN *Flood*: Utiliza paquetes TCP con los *flags* RST y FIN activados para saturar un servidor. El servidor intenta cerrar las conexiones repetidamente, gastando recursos en el proceso. Este ataque puede provocar la degradación del rendimiento del servidor y la denegación de servicio. *(RST-FIN Flood | Knowledge Base | MazeBolt)*

2.4. Conjuntos de datos en entornos IoT

En esta sección se presentan los conjuntos de datos más relevantes en el escenario de ataques en IoT (véase la Tabla II), describiendo sus principales características y los tipos de ataques que incluyen. Es crucial contar con conjuntos de datos representativos para poder evaluar de forma correcta un escenario IoT. Cada uno de estos conjuntos de datos ha sido analizado en función de los ataques presentes, la disponibilidad de las capturas de tráfico originales y de las reglas de etiquetado utilizadas para garantizar su utilidad y validez en la evaluación de sistemas de detección de *botnets*. Tras este análisis, se decidió que los conjuntos de datos que mejor cumplían con las características que permitirían llevar a cabo la propuesta de evaluación de este trabajo eran los siguientes: IoTD20, IoT-23 y CIC-IoT-2023. A continuación se describe en mayor detalle cada uno de ellos.

Tabla II: Listado de Conjuntos de Datos en entornos IoT estudiados.

Dataset	**Características**	**Ataques**
BOT-IoT (Koroniotis et al., 2018)	Servicios como DNS, FTP, HTTP y SSH **32 características** 72,000,000 de registros Simulados tráfico normal y ataques DoS y DDoS.	**DoS y DDoS:** SYN, TCP, UDP, HTTP. **Escaneo** de Puertos y Sistemas Operativos **Robo** de **Información** ***Keylogging***
HIKARI (Ferriyan et al., 2021)	517,582 flujos de tráfico benigno 37,696 flujos de tráfico malicioso Etiquetado con categorías Benigno o Ataque Ataques simulados	***Brute Force*** tradicional ***Brute Force* con diferentes vectores de ataque (XMLRPC)** ***Probing*** ***Botnet* XMRIGCC CryptoMiner**
IoT-BDA (Trajanovski & Zhang, 2021)	**Honeypots** simulando servicios vulnerables con analizadores estáticos y dinámicos. 4077 muestras únicas de *botnets* **39 columnas** en total. Nombre del archivo, *Botnet*, hash MD5, arquitectura de CPU, técnicas anti-análisis, resultados de análisis de VirusTotal.	Comunicaciones **C2** **Ataques DDoS** **Escaneo de puertos**
AWID-3 (Chatzoglou et al., 2021)	Cada captura de menos de 2.5 millones de frames y duración total de 10 minutos Entorno de laboratorio físico simulando infraestructura empresarial **16 dispositivos.** Variaciones de tráfico normal y de ataque. **254 características extraídas.**	Explotación de **vulnerabilidades como Krack y Kr00k** y creación de **puntos de acceso falsos.** **Fuerza bruta** y creación de ***botnets*** mediante infección de STAs **Inyección SQL** para manipular bases de datos web y la amplificación SSDP (**DDoS**) **Evil_Twin** que combina envenenamiento ARP y DNS para ***Website Spoofing***

MedBIoT (Guerra-Manzanares et al., 2020)	**80 dispositivos virtuales y 3 físicos**. Red de Internet para configuración de dispositivos. Red de monitoreo para almacenamiento y análisis. Red IoT LAN con dispositivos IoT. **100 características** estadísticas de tráfico en diferentes ventanas de tiempo (100ms, 500ms, 1.5s, 10s, 1min). 4,143,276 paquetes de BashLite, 842,674 paquetes de Mirai, 319,139 paquetes de Torii, y 12,540,478 paquetes de tráfico benigno.	**Mirai**: infectó 25 dispositivos **BashLite** (Yakuza version): infectó 40 dispositivos **Torii**: infectó 12 dispositivos.
Edge-IIoT (Al Nuaimi et al., 2023)	Generación de tráfico benigno y **14 tipos de ataques** simulados **61 características** seleccionadas de las 1176 características iniciales encontradas	DoS/DDoS, recolección de información, ataques de hombre en el medio, inyección de código y ataques de *malware*.
MBB-IoT (Qing et al., 2023)	**Dispositivos IoT reales** para simular entornos de ataque, generando tráfico normal. Controlados por LAN de Raspberry Pi. Simulan dos escenarios: tráfico normal y de alto tráfico **87 características** para analizar ataques DDoS. Etiquetado como “anómalo" o "benigno"	*Malware* **Mirai y BASHLITE** descargados y ejecutados en dispositivos IoT mediante servidores en la nube. Ataques **DDoS** hacia servidores WEB utilizando los dispositivos infectados, con diferentes variantes de ataques
N-BaIoT (Meidan et al., 2018)	Dispositivos IoT infectados utilizando binarios C2 dentro del entorno de laboratorio 115 características estadísticas para describir el comportamiento del tráfico, ventanas temporales de 100ms, 500ms, 1.5s, 10s y 1 minuto.	**BASHLITE:** exploración, UDP*Flood* y TCP*Flood* y envío de datos no deseados **Mirai**: exploración, ACK*Flood*, SYN*Flood* y UDP*Flood*, optimizados para aumentar la tasa de paquetes por segundo (PPS).

2.4.1. Descripción de IoTD20

El *dataset* IoTID20 (Ullah & Mahmoud, 2020) se centra en un entorno doméstico inteligente IoT, utilizando dispositivos como el SKT NGU y la cámara Wi-Fi EZVIZ como dispositivos víctimas. Además, incluye dispositivos adicionales como portátiles, *tablets* y *smartphones* que actúan como dispositivos atacantes. Se simularon diversos tipos de ataques dentro de este entorno, capturando el tráfico en archivos pcap. Los dispositivos están conectados a un *router* Wi-Fi doméstico, lo que proporciona una topología de red típica para un hogar inteligente.

El *dataset* incluye ataques simulados y reales. Entre los ataques simulados se encuentran UDP/ACK/HTTP *Flood*, típicos de la *botnet* Mirai. Además, se capturaron ataques reales utilizando herramientas como Nmap para escaneos de hosts y puertos, y ataques de *spoofing* ARP. Para la *botnet* Mirai, los paquetes fueron generados en un portátil y fueron manipulados para simular haber sido generados desde el dispositivo IoT. La implantación de *malware* incluye la simulación de ataques de la *botnet* Mirai, donde los dispositivos comprometidos generan tráfico malicioso como parte del *dataset*. La captura de tráfico se realizó en modo monitor utilizando adaptadores de red inalámbrica, con eliminación de cabeceras de red.

El *dataset* está compuesto por 42 archivos pcap que contienen paquetes de red capturados en diferentes momentos. El tamaño total del *dataset* es de aproximadamente 1.45 GB. En la Figura 7 se muestra la distribución de los ataques. En la Tabla III se detalla el número de instancias para cada clase; estos datos corresponden al número de paquetes capturados; y en el Anexo II se detallan en forma de instancias del *dataset* en formato csv.

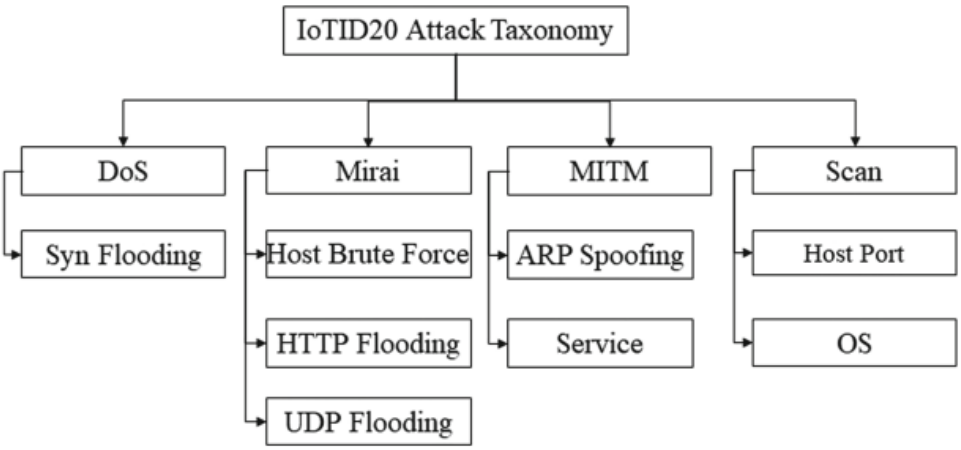


Figura 7: Distribución de ataques de IoTD20 en clases y subclases (Ullah & Mahmoud, 2020).

Tabla III: Número de paquetes por categoría y subcategoría en IoTD20. (Ullah & Mahmoud, 2020).

Categoría	Subcategoría	Paquetes
Normal	Normal	1,756,276
Scanning	Host Discovery	2,454
	*Port Scan*ning	20,939
	OS/Version Detection	1,817
Man in the Middle (MITM)	ARP Spoofing	101,885
Denial of Service (DoS)	SYN *Flood*ing	64,646
Mirai *Botnet*	Host Discovery	673
	Telnet *Brute Force*	1,924
	UDP *Flood*ing	949,284
	ACK *Flood*ing	75,632
	HTTP *Flood*ing	10,464

El *dataset* IoTID20 se caracteriza por sus **83 atributos de red y tres atributos de etiqueta**, diseñados para la detección de intrusiones en entornos de IoT. Estos atributos se extraen de archivos pcap utilizando la aplicación CICflowmeter y se presentan en formato CSV.

2.4.2. Descripción de IoT-23

El *dataset* IoT-23 (Garcia et al., 2020) es una compilación de tráfico de red capturado de dispositivos IoT, que se divide en escenarios benignos y maliciosos. Desarrollado por el Laboratorio Stratosphere de la Universidad CTU en Praga, proporciona datos etiquetados, capturas de tráfico originales y las reglas empleadas para el etiquetado.

En los 20 escenarios maliciosos de IoT-23, se ejecutaron muestras de *malware* en dispositivos Raspberry Pi dentro de un entorno controlado. Cada captura de tráfico, en formato .pcap, registra las interacciones de red generadas por el *malware*, generalmente limitados a 24 horas debido al volumen creciente de tráfico. Este enfoque asegura la captura de comportamientos significativos y realistas de las infecciones por *malware* en dispositivos IoT. Cada uno contiene tráfico de una *Botnet*, como se muestra en el Anexo II, donde también se indica el número de flujos y la duración de cada escenario. Para este trabajo, se han utilizado todos los escenarios que contienen Mirai y los escenarios con tráfico benigno. En contraste, se incluyen tres capturas

de tráfico de dispositivos IoT no infectados: una lámpara LED inteligente Philips HUE, un asistente personal inteligente Amazon Echo y una cerradura inteligente Somfy.

El análisis de las capturas de tráfico se realiza utilizando Zeek, una herramienta para el análisis de redes que genera archivos conn.log detallando flujos de conexión. Cada archivo conn.log es posteriormente etiquetado utilizando un proceso manual y automático para caracterizar el comportamiento del tráfico, este último recibe el nombre de Flaber, un *script* personalizado desarrollado en Zeek. Estas etiquetas proporcionan una descripción detallada de las interacciones de red observadas, y poseen una estructura en la que se concatenan todos los nombres de los ataques en el flujo. Las características presentes en el archivo conn.log se adjuntan en el Anexo II, junto a la explicación de cada una de ellas. Contiene **21 atributos** que corresponden al número de parámetros obtenidos del archivo conn.log además de 2 campos de etiquetado. Cada carpeta de captura incluye un archivo README.md que detalla información relevante, como el nombre y características del *malware*, hash (MD5, SHA1, SHA256) de la muestra, la duración de la captura en segundos y enlaces a análisis adicionales de muestras de *malware* en VirusTotal. Además, también se adjuntan los archivos .pcap originales y conn.log.labeled.

Los posibles valores que pueden encontrarse en las etiquetas son los mostrados en la Tabla IV:

Tabla IV: Valores posibles del campo "etiqueta" en el conjunto de datos IoT-23.

Attack	FileDownload	PartOfAHorizontal*PortScan*
Benign	HeartBeat	Torii
C&C (Command & Control)	Mirai	
DDoS	Okiru	

El *dataset* IoT-23 ofrece una combinación única de datos reales y simulados de tráfico de dispositivos IoT. A continuación, se detalla cómo se distribuye la simulación dentro del *dataset*:

Datos Reales: Los escenarios benignos del *dataset* IoT-23 están compuestos por capturas de tráfico de dispositivos IoT reales y no infectados. Estos incluyen una lámpara LED inteligente Philips HUE, un asistente personal inteligente Amazon Echo y una cerradura inteligente Somfy. Estos dispositivos funcionan en un entorno controlado y proporcionan un reflejo auténtico de los patrones de tráfico normales en dispositivos IoT no comprometidos.

Datos Simulados: Por otro lado, los escenarios maliciosos del *dataset* implican la ejecución controlada de muestras de *malware* en dispositivos Raspberry Pi. Cada captura de *malware* captura el comportamiento y las interacciones generadas por el *malware* durante un período limitado, generalmente menos de 24 horas debido al volumen de tráfico generado.

En el Anexo VIII se adjunta las reglas empleadas para el etiquetado, generadas a partir de atributos de Zeek. Si un flujo cumple más de una regla, se concatenan todas las etiquetas correspondientes. En el Anexo II se muestra la distribución de etiquetas para cada escenario empleado en este trabajo.

2.4.3. Descripción de CIC-IoT-2023

El *dataset* CICIoT2023 (Neto et al., 2023) destaca por simular un entorno realista de IoT con dispositivos distribuidos en un laboratorio que imita un hogar inteligente. Se emplean 105 dispositivos IoT, divididos en categorías como dispositivos domésticos inteligentes, cámaras,

sensores y microcontroladores. Estos dispositivos están configurados tanto para comportamientos benignos como para ejecutar ataques maliciosos. El tráfico de red capturado incluye tanto actividades benignas como maliciosas. Las actividades benignas incluyen interacciones humanas como datos de sensores y solicitudes de dispositivos, mientras que los ataques maliciosos cubren diversas técnicas como DDoS, DoS, y explotación de vulnerabilidades web.

La topología de red se divide en dos partes principales conectadas a través de un *router* ASUS un switch, y un punto de acceso. Esta configuración simula un entorno típico de hogar inteligente, con dispositivos distribuidos físicamente en diferentes ubicaciones dentro del laboratorio. Se utilizan *botnets* simuladas para llevar a cabo ataques como DDoS, DoS, y explotación de vulnerabilidades web. Estos ataques son ejecutados por dispositivos IoT maliciosos dirigidos a otros dispositivos vulnerables dentro del mismo entorno simulado. En el Anexo II se especifica la distribución de las etiquetas por ataque y el número de instancias de cada clase.

El tráfico de red es capturado mediante un Gigamon Network Tap, que proporciona acceso pasivo y no intrusivo al tráfico completo de la red. Los datos capturados son analizados y almacenados utilizando herramientas como Wireshark. Los datos capturados se almacenan en archivos pcap y csv. Los archivos pcap contienen datos originales capturados, mientras que los archivos csv contienen características extraídas de ventanas de paquetes fijos para análisis posterior. Se extraen múltiples características de los datos capturados utilizando herramientas como DPKT. Estas características incluyen estadísticas de paquetes, patrones de tráfico y comportamientos anómalos. Incluye un total de **47 características extraídas** de los datos capturados (véase Anexo II).

Los datos capturados son preprocesados para limpiar y estructurar adecuadamente los paquetes de red. Además, cada conjunto de datos se etiqueta según el tipo de actividad, ya sea benigna o maliciosa, facilitando así el entrenamiento de modelos de aprendizaje automático. El *dataset* incluye múltiples flujos de datos capturados durante un período de tiempo específico, con un total de aproximadamente 548 GB de tráfico. Las herramientas utilizadas incluyen TCPDUMP para la conversión de archivos pcap a csv, DPKT para la extracción de características, y Pandas para el procesamiento y análisis de datos. El etiquetado se realiza asignando la misma etiqueta a todos los flujos procedentes de la captura de tráfico del ataque en cuestión.

3. Técnicas de ML

La Inteligencia Artificial es la capacidad de un dispositivo para realizar tareas de manera similar a un humano, utilizando la computación para imitar las funciones cognitivas humanas. Este concepto abarca más que el aprendizaje automático o *Machine Learning* (ML), que se puede considerar como una subárea de la Inteligencia Artificial enfocada en la capacidad de las máquinas para procesar conjuntos de datos y realizar predicciones basadas en estos, adaptando los algoritmos conforme aprenden de manera continua por sí mismas (Kubat, 2021; Molina López & García Herrero, 2006). En la Figura 8 se presentan las principales técnicas de ML, mientras que en la siguiente sección se profundiza en aquellas utilizadas en este trabajo. Las principales categorías de aprendizaje automático son: el aprendizaje supervisado, el aprendizaje no supervisado, el semisupervisado, el aprendizaje profundo y el aprendizaje de refuerzo.

Figura 8: Categorías de Técnicas de Machine Learning.

3.1. Técnicas de Machine Learning para clasificación e indicadores de rendimiento

En esta sección se detallan las técnicas utilizadas para la clasificación de flujos de tráfico. Primero, se explican los fundamentos de las técnicas de clasificación basadas en aprendizaje supervisado y, posteriormente, se describen los principales indicadores de rendimiento de estos clasificadores.

Un algoritmo de clasificación supervisado tiene como objetivo extraer conocimiento de un conjunto de datos (*training* set) y modelar dicho conocimiento para aplicarlo en la toma de decisiones sobre un nuevo conjunto de datos (*test* set). Matemáticamente, en el aprendizaje supervisado se trabaja con un conjunto de datos compuesto por ejemplos etiquetados $(x_i, y_i)\}$

con i=1 … N. Cada elemento x_i es un vector de características, en el que cada dimensión j=1, ..., D contiene un valor que describe el ejemplo. Este valor se llama característica y se denota como $x_i(j)$. Para todos los ejemplos en el conjunto de datos, la característica en la posición j del vector de características siempre contiene el mismo tipo de información (por ejemplo, el número de paquetes de los que consta un flujo de tráfico). La etiqueta (*label*) denotada como y_i puede pertenecer a un conjunto finito de clases {1, 2, ... ,C}, representando una categoría a la que pertenece una instancia, como un tipo específico de ataque. El objetivo de un algoritmo de aprendizaje supervisado es usar el conjunto de datos $\{(x_i, y_i)\}$ para producir un modelo de clasificación que permita tomar un nuevo vector de características x como información de entrada y que como salida pueda deducir la etiqueta que debería asignarse a dicho vector (Kubat, 2021). Las etiquetas de un conjunto de datos pueden ser cuantitativas (valores continuos) o cualitativas (valores discretos que pertenecen a una clase). En este trabajo, las etiquetas son cualitativas ya que corresponden a los nombres de los diferentes ataques o a la etiqueta de tráfico benigno. Generalmente, los datos con etiquetas cualitativas se asocian a algoritmos de clasificación, mientras que los datos con etiquetas cuantitativas se asocian a algoritmos de regresión.

3.2. Técnicas de clasificación

El problema de clasificación se aborda utilizando atributos simbólicos. Si se emplean atributos numéricos, es necesario discretizarlos previamente en intervalos para representar adecuadamente los valores de la clase. A continuación, se describen las familias de clasificadores más comunes, junto con una breve explicación de los clasificadores específicos empleados en este trabajo.

Clasificadores Bayesianos

Los métodos bayesianos ofrecen una medida probabilística cuantitativa de la relevancia de las variables en un problema de clasificación. Al aplicar estos métodos, es crucial evitar la presencia de correlaciones entre los atributos del conjunto de entrenamiento, ya que esto podría invalidar los resultados obtenidos.

- ***Naive* Bayes**: Un clasificador Naïve Bayes es un método probabilístico que utiliza el teorema de Bayes y las probabilidades condicionales, asumiendo que todas las variables predictoras son independientes entre sí (García et al., 2018; John, 1995; Kubat, 2021). Esta simplificación crea un modelo con un único nodo raíz (la clase) y nodos hoja (los atributos). Una ventaja del clasificador Naïve Bayes es que requiere pocos datos de entrenamiento para estimar los parámetros necesarios para la clasificación. Aunque la hipótesis de independencia es difícil de cumplir y puede ser distorsionada por atributos altamente correlacionados, esta técnica puede funcionar bien cuando se combina con técnicas de selección de atributos para eliminar redundancias.

- El clasificador Naïve Bayes de Bernoulli es adecuado para problemas de clasificación binaria o multiclase donde las características son binarias (por ejemplo, la presencia o ausencia de una palabra en un documento). Este clasificador modela la probabilidad de cada característica dada la clase como una distribución de Bernoulli. Por lo tanto, esta clase requiere que las muestras se representen como vectores de características binarias; si se le proporciona cualquier otro tipo de datos, una instancia de BernoulliNB puede binarizar su entrada.

- Por otro lado, el clasificador Naïve Bayes Gaussiano se utiliza cuando las características son continuas y se asume que siguen una distribución gaussiana (normal). Este clasificador modela la probabilidad de cada característica dada la clase como una distribución gaussiana.

Funciones

En este grupo de métodos se han incluido aquellos que generan una función de clasificación y que no obtienen de forma explícita un árbol o conjunto de reglas.

- ***Multi-layer Perceptron* (MLP)**: El perceptrón multicapa (MLP) es una red neuronal artificial diseñada para resolver problemas no linealmente separables (García et al., 2018; Kubat, 2021) Suele tener una capa de entrada para atributos, una o más capas ocultas donde se calculan sumas ponderadas de las entradas multiplicadas por los pesos sinápticos, y una capa de salida que clasifica las instancias según las clases deseadas. Durante el entrenamiento, se ajustan los pesos de las conexiones utilizando retropropagación, un proceso que minimiza el error entre la salida predicha y el resultado esperado.

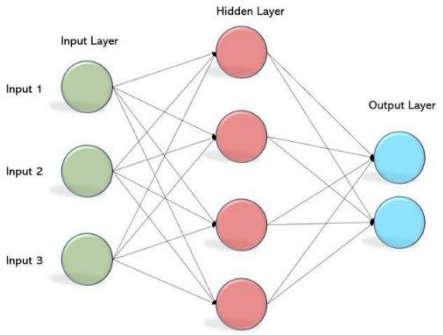


Figura 9: Ejemplo de perceptrón multicapa (MLP) con una capa oculta.

Aprendizaje basado en instancias

En este enfoque de aprendizaje, se mantienen almacenados los ejemplos de entrenamiento. Cuando se necesita clasificar una nueva instancia, se identifican las instancias previamente clasificadas más similares y se utiliza su etiqueta para clasificar la nueva instancia. Este tipo de métodos se conocen como "aprendizaje perezoso" (*lazy learners*), donde el proceso de aprendizaje inicial es mínimo y el tiempo se consume principalmente en la fase de clasificación.

- **Nearest Centroid:** es un algoritmo sencillo que representa cada clase mediante el centroide de sus miembros. Esto lo hace similar a la fase de actualización de etiquetas del algoritmo KMeans (Nearest Centroid Classification — Scikit-Learn 0.18.2 Documentation). A diferencia de otros métodos, no tiene parámetros para ajustar. El método de K-Means es un enfoque de clasificación no paramétrico que determina la clase de una instancia según las clases de sus k instancias de entrenamiento más cercanas. Durante el entrenamiento, se almacenan los vectores de características y las etiquetas de las clases de los ejemplos en un espacio multidimensional. K-Means investiga cada instancia, calculando sus distancias a todos los centroides. El centroide más cercano define el *cluster* al que pertenece la instancia. Si ya está en ese *cluster*, no se realiza ninguna acción; de lo contrario, se transfiere al *cluster* correcto. Después de la reubicación, se recalculan los centroides de los *clusters* afectados. En la fase de

clasificación, la instancia se asigna a la clase más frecuente entre sus k vecinos más cercanos, asumiendo que estos vecinos proporcionan una buena clasificación basada en la similitud en el espacio de características.

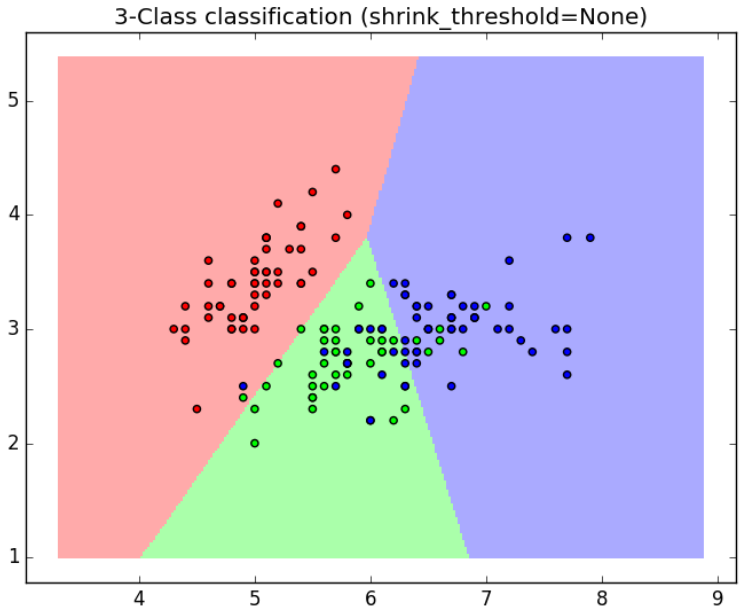


Figura 10: Ejemplo de funcionamiento del algoritmo Nearest Centroid (Nearest Centroid Classification — Scikit-Learn 0.18.2 Documentation).

Metaclasificadores

En esta categoría se encuentran los clasificadores complejos, los cuales son obtenidos mediante la composición de clasificadores simples o incluyen preprocesamiento de datos.

- ***Adaptive Boosting* (AB):** AdaBoost es un meta-algoritmo de clasificación que utiliza una combinación secuencial de clasificadores débiles para mejorar la precisión del clasificador final. En cada iteración, se ajusta un clasificador débil que se centra en corregir los errores de clasificación cometidos por los clasificadores anteriores. La contribución de cada clasificador débil a la predicción final se pondera según su desempeño, favoreciendo aquellos que tienen mejor capacidad predictiva. Este algoritmo se aplica principalmente en problemas de clasificación binaria, aunque puede extenderse al caso multiclase. La característica distintiva de AdaBoost es su capacidad para mejorar progresivamente el rendimiento del modelo combinando múltiples clasificadores débiles.

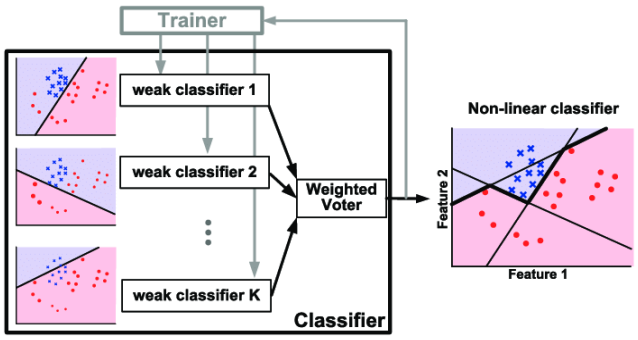


Figura 11: Ejemplo de funcionamiento del algoritmo Adaptive Boosting.

- **Bagging:** es una técnica que utiliza múltiples clasificadores para mejorar la precisión. Varios clasificadores operan simultáneamente para predecir las etiquetas de clase de los ejemplos. Un clasificador principal combina estas predicciones mediante votación mayoritaria. Cada clasificador en el conjunto aborda diferentes aspectos del problema, lo que mejora el rendimiento global sobre clasificadores individuales (Kubat, 2021). Para implementar *Bagging*, se generan subconjuntos de entrenamiento T_1 ,...,T_n mediante *bootstrapping* del conjunto original T. Cada subconjunto se utiliza para entrenar un clasificador C_i , como árboles de decisión ajustados con parámetros definidos por el usuario. Esta técnica aprovecha la diversidad de los clasificadores para reducir errores; si un clasificador falla en un ejemplo, es probable que los otros clasificadores lo clasifiquen correctamente.

Árboles de decisión

Un árbol de decisión es un clasificador que trata de hallar la mejor opción en cada paso o decisión que se toma en el árbol, de modo que cada partición seleccionada maximice algún criterio de discriminación (error de clasificación, ganancia de entropía, etc.)(García et al., 2018; Kubat, 2021). Los árboles constituyen un modo intuitivo para visualizar la clasificación de un conjunto de datos.

- ***Decision Tree:*** *Se bas*a en la creación de reglas de decisión simples derivadas de las características de los datos. Esta estructura jerárquica se construye mediante divisiones recursivas de los datos en subconjuntos cada vez más homogéneos en términos de la variable objetivo. Una ventaja de los árboles de decisión es su capacidad para manejar datos sin requerir normalización ni transformación de variables, y algunos algoritmos pueden manejar automáticamente valores faltantes. Además, el coste computacional crece de manera logarítmica con el tamaño del conjunto de entrenamiento, lo cual los hace eficientes para conjuntos de datos extensos. No obstante, los árboles de decisión pueden sufrir de sobreajuste, donde el modelo se ajusta demasiado a los datos de entrenamiento y no generaliza bien a nuevos datos. Para mitigar este problema, se emplean técnicas de poda que limitan la profundidad del árbol o reducen el número de nodos, mejorando así su capacidad de generalización. El algoritmo ***Decision TreeClassifi*****er** de *Scikit-Learn i*mplementa árboles de decisión utilizando una versión optimizada del algoritmo CART (*Classification and Regression Trees*). CART es uno de los métodos más comunes y efectivos para construir árboles de decisión.

- ***Random Forest*** **(RF):** En esta técnica se construyen bosques aleatorios (*Random Forest*) creando conjuntos de árboles aleatorios o *random trees* (Breiman, 2001; Kubat, 2021). Los árboles creados con el algoritmo de *Random Tree* consideran un número específico de características aleatorias en cada nodo, sin realizar poda. El algoritmo explora aleatoriamente una variedad de modelos, lo que permite combinar cientos de árboles de decisión y entrenar cada uno con una selección diferente de instancias. Las predicciones finales del bosque aleatorio se obtienen promediando las predicciones de cada árbol individual (ver Figura 12). Usando *Random Forest*, se puede mitigar el efecto de sobreajuste de los árboles de decisión individuales al promediar los resultados de predicción de múltiples árboles, aunque esto también incrementa la complejidad computacional del método.

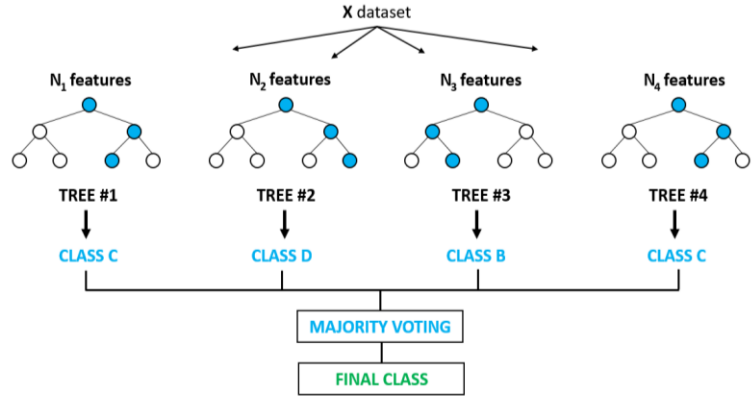


Figura 12: Ejemplo de funcionamiento del algoritmo Random Forest.

Stochastic Gradient Descent (SGD)

SGD es una técnica de optimización y no corresponde a una familia específica de modelos de aprendizaje automático. Las ventajas del descenso de gradiente estocástico (SGD) son su eficiencia y facilidad de implementación (muchas oportunidades para ajustar el código). La clase *SGDClassifier* de *Scikit-Learn* admite diferentes funciones de pérdida y penalizaciones para la clasificación. SGD ajusta un modelo lineal a los datos de entrenamiento. Admite la clasificación multiclase combinando múltiples clasificadores binarios en un esquema "uno contra todos" (OVA). Para cada una de las K clases, se aprende un clasificador binario que discrimina entre esa clase y las otras K−1 clases. En el momento de la prueba, se calcula la medida de confianza (es decir, las distancias al hiperplano) para cada clasificador y se elige la clase con la mayor confianza. Se muestra un ejemplo de clasificación multiclase en la Figura 13.

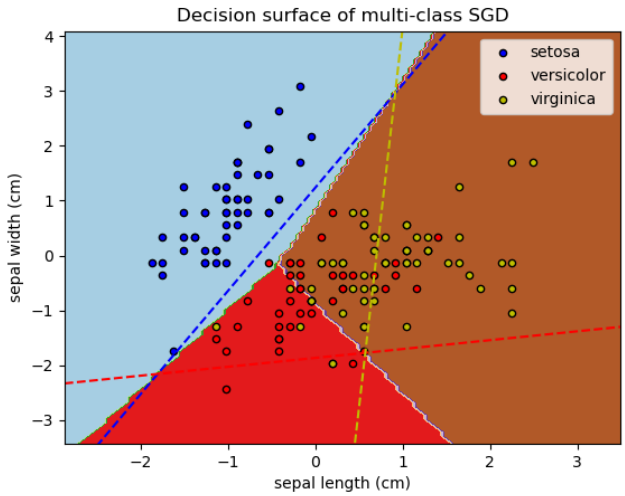


Figura 13: Ejemplo de superficie de decisión de SGD. (Stochastic Gradient Descent — Scikit-Learn 1.5.0 Documentation).

3.3. Indicadores de rendimiento de los clasificadores

A continuación, se presentan las métricas o indicadores de rendimiento más comúnmente utilizados en la evaluación de técnicas de clasificación, especialmente aplicados al problema de la detección de *botnets*. Se definen los siguientes factores de clasificación: TP (verdaderos positivos) es el número de flujos de ataques correctamente identificados, TN (verdaderos negativos) es el número de flujos correctamente identificados como normales, FP (falsos positivos) es el número de flujos normales incorrectamente clasificados como ataques, y FN (falsos negativos) es el número de flujos de ataques incorrectamente clasificados como normales. La matriz de confusión muestra el número de flujos clasificados de manera correcta o incorrecta, tal como se presenta en la Tabla V. A partir de estos elementos, se definen los siguientes indicadores (Kubat, 2021).

.

Tabla V: Matriz de confusión para clasificación de tráfico anómalo.

Class\Prediction	***Normal***	***Attack***
Normal	TN	FP
Attack	FN	TP

Exactitud *(accuracy)*: Representa la proporción de flujos de tráfico clasificados correctamente respecto al número total de flujos. Es una métrica común para evaluar la eficacia de los algoritmos de clasificación y también se conoce como tasa de clasificación (CR). Su complementario (1-Acc) es la tasa de error.

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

En dominios desequilibrados (*imbalanced*), donde las instancias de una clase superan en número a las de la otra (por ejemplo, un número significativamente mayor de flujos de tráfico normales frente a flujos de ataques), la exactitud puede resultar engañosa. En estos casos, es necesario emplear otros indicadores como el *recall* y la precisión.

***Recall* o Tasa de verdaderos positivos (TPR):** Representa la probabilidad de que un ejemplo positivo sea correctamente identificado por el clasificador. También se conoce como tasa de detección o sensibilidad.

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

Precisión: Representa la proporción de verdaderos positivos respecto a todos los ejemplos clasificados como positivos. Es una medida de la probabilidad estimada de una predicción positiva correcta y también se denomina valor predictivo positivo. Cuando los TP son 0 (ningún ataque clasificado correctamente) y los FP son 0 (todos los benignos clasificados correctamente), no se obtiene un resultado numérico válido.

$$Precisión = \frac{TP}{TP + FP}$$

Mientras que la precisión indica la frecuencia de verdaderos positivos (ataques reales) entre todos los ejemplos considerados como positivos por el clasificador (flujos clasificados como ataques), el *recall* mide la frecuencia de verdaderos positivos (ataques reales) entre todos los ejemplos positivos en el conjunto de datos (ataques en el *dataset*).

***F-Measure* o F1**: El indicador F combina precisión y *recall* en un único valor ponderado. Si se asigna el mismo peso a ambos, se obtiene F1.

$$F1 = \frac{2\,TP}{2\,TP + FP + FN} = 2\frac{precision\ \ recall}{precision + recall}$$

***Receiver Operating Characteristic* (ROC)**: este diagrama gráfico se utiliza para evaluar el rendimiento de un algoritmo de clasificación binario. La curva ROC se crea trazando la tasa de verdaderos positivos frente a la tasa de falsos positivos en diversas configuraciones de funcionamiento, y el área bajo la curva ROC indica la calidad del clasificador.

Además de los indicadores de rendimiento descritos previamente, existen diferentes métodos de validación de los algoritmos que permiten obtener los indicadores de distinta forma. La técnica más básica es la conocida como validación simple o *train-test*, en la que se elabora el modelo utilizando el conjunto de entrenamiento y se aplica sobre el conjunto *test*. Se pueden establecer diferentes divisiones (porcentaje de *split*) para dividir un conjunto original en los subconjuntos de *train* y *test*. Otras técnicas utilizadas son la validación cruzada, técnicas de *Bootstrap*, etc. que son más costosas computacionalmente.

4. Sistema de detección de *botnets*

4.1. Arquitectura general del sistema

En este capítulo se desarrolla el procedimiento seguido para realizar la detección de *botnets* a partir de los nuevos parámetros de tráfico obtenidos aplicando Zeek sobre las capturas de paquetes con las que se generaron los *datasets* originales IoTD20, IoT-23 y CIC-IoT-2023, así como el análisis de los resultados de clasificación obtenidos utilizando dichos parámetros. Este se ilustra en la Figura 14. El objetivo es **detectar en primer lugar los flujos de tráfico utilizando Zeek** a partir de los ficheros que contienen las capturas directas de los paquetes generados durante la construcción de los *datasets* (ficheros de captura de paquetes de tráfico en formato .pcap), y posteriormente **encontrar atributos o características** de dichos flujos de tráfico para ser utilizados como datos de entrada de los clasificadores. Gracias a esto, los atributos son comunes en los tres conjuntos de datos, y se puede estudiar el rendimiento de un único modelo que es entrenado con un conjunto de datos diferente al que se emplea para su evaluación. Además, se podrá comparar cómo varía el rendimiento en la clasificación de los flujos de tráfico al utilizar los atributos obtenidos utilizando Zeek respecto al rendimiento obtenido utilizando los atributos que contienen los flujos en los *datasets* originales. Para llevar a cabo estas pruebas se empleó el software Zeek, un analizador de tráfico de red pasivo y de código abierto. Esta sección se subdivide en los apartados 5.1, que especifica la metodología empleada y el apartado 5.2, en el que se presentan y analizan los resultados obtenidos.

El esquema general de trabajo seguido en este estudio se representa en la Figura 14, donde se pueden observar las diferentes etapas de este. En primer lugar, la etapa de **generación de logs** se utiliza para obtener información y estadísticas a partir de los flujos de tráfico extraídos de las capturas. Tras obtener los logs, se combinaron en un único archivo. El siguiente paso es el **etiquetado de los flujos** de tráfico, aplicando tanto las etiquetas originales también contenidas en los *datasets* originales, como otras que serán comunes entre los tres conjuntos de datos. Como se puede apreciar en la Figura 14, cada uno de estos ficheros fue transformado para la unificación de formato y limpieza de datos. Una vez se tienen los conjuntos de datos listos, se puede llevar a cabo la división en conjunto de entrenamiento y de *test*. En función de la prueba realizada, puede realizarse una selección de atributos para emplear solamente los elegidos en dicha prueba. Finalmente, se entrena el modelo y se evalúa, para obtener resultados.

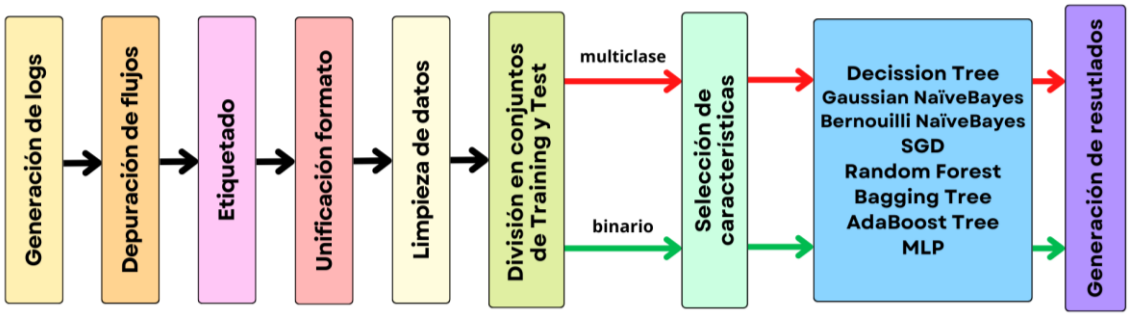


Figura 14: Esquema general de la metodología utilizada.

4.2. Obtención de logs y atributos mediante Zeek

En esta sección se detallan las transformaciones aplicadas para obtener los flujos de tráfico y los atributos que los caracterizan (que después se emplean en la clasificación), así como el etiquetado de dichos flujos. La Figura 14 muestra este procedimiento. En líneas generales, primero se empleó Zeek para obtener los distintos logs a partir del fichero de captura de paquetes .pcap, ejecutando un *script* que permite obtener los datos de cada conexión completa (con_statistics.zeek), es decir, de cada flujo de tráfico. Para estas pruebas, se obtuvieron los atributos propios el archivo conn.log de Zeek, que contiene detalles de cada conexión a nivel de los protocolos IP, TCP, UDP e ICMP, además de otras estadísticas relacionadas con los atributos originales. En un futuro se podrían añadir atributos a partir de otros logs más específicos. Después se realiza la conversión a CSV mediante otro *script* desarrollado en Python. Tras esto, se unen todos los ficheros csv correspondientes a un único *dataset* en un mismo archivo csv mediante otro *script* de Python. Posteriormente, se añaden a los flujos las etiquetas originales además de las etiquetas nuevas que se emplearán en la clasificación, y se realiza la limpieza de los datos. En la limpieza, se intercambia cualquier posible valor vacío por 0 o el carácter correspondiente si es un atributo categórico (además de unificar los valores de los atributos local_orig y local_resp). Por último, se codifican todos aquellos atributos que sean de tipo *string* para asegurar la compatibilidad con todos los algoritmos.

Para obtener los flujos a partir de las capturas de tráfico, se ha utilizado Zeek. Cuando se analiza una captura de tráfico con Zeek, obtenemos los logs del Anexo III. Para este trabajo, se empleó el *script* personalizado *conn_statistics.zeek* (véase Anexo IV).

El primer fichero se encarga de obtener información de los distintos campos de los paquetes para generar un archivo .log en el que, para cada flujo, indica direcciones IP origen y destino, puertos, protocolo, servicio, *bytes* enviados y recibidos, entre otros parámetros. A continuación, se muestran todos los parámetros obtenidos del conn_statistics.log en la Tabla VI. En el Anexo III se explican en mayor detalle los atributos. Este fichero nos aporta la información principal de los flujos, además de medidas estadísticas como la media, desviación estándar, valor máximo y mínimo de atributos como los *bytes* enviados desde origen y en respuesta, o la cantidad de paquetes que no tienen *payload* vacío. Estas medidas pueden ser de utilidad para caracterizar el comportamiento de ciertos tipos de ataque, como pueden la media de *bytes* de origen y de destino para ataques que se basan en inundar con paquetes a la víctima.

Tabla VI: *Nuevos atributos obtenidos con Zeek a partir de conn_statistics.log*.

ATRIBUTOS		
TimeStamp	*Bytes* IP respuesta	Paquetes origen cero
IP origen	Tunnel parents	Paquetes resp. cero
Puerto origen	Media *bytes* orig.	Media tiempo
IP destino	Media *bytes* resp.	Desv. estándar tiempo
Puerto destino	Desv. estándar *bytes* orig.	Mín tiempo
Protocolo	Desv. estándar *bytes* resp.	Máx tiempo
Servicio	Media *bytes* orig no cero	Media tiempo origen

Duración	Media *bytes* resp no cero	Desv. estándar tiempo origen
Bytes origen	Desv. estándar *bytes* orig no cero	Mín tiempo origen
Bytes respuesta	Desv. estándar *bytes* resp no cero	Máx tiempo origen
Conn_state	Mín *bytes* orig.	Media tiempo resp.
Missed *Bytes*	Mín *bytes* resp.	Desv. estándar tiempo resp.
History	Máx *bytes* orig.	Mín tiempo resp.
Paquetes origen	Máx *bytes* resp.	Máx tiempo resp.
Bytes IP origen	Paquetes origen no cero	
Paquetes respuesta	Paquetes resp. no cero	

4.3. Manipulación y limpieza de datos

Una vez obtenidos los logs con los atributos de Zeek, se realizó la conversión al formato .csv mediante un *script* de Python (véase Anexo V y Anexo VI). Este paso corresponde a la **selección de flujos** en la Figura 14. Al analizar los logs mediante la herramienta Zui, se observó que había cierta pérdida de paquetes en las capturas de tráfico de los conjuntos de datos. Por ello, se decidió analizar en mayor profundidad y se diseñaron *scripts* de Python (véase Anexo IX y Anexo X) para obtener aquellos flujos de tráfico con pérdidas superiores a un umbral.

Para poder estimar la cantidad de flujos que podían presentar este defecto, se decidió representar gráficamente el porcentaje de *bytes* perdidos por flujos para cada log individual. En la Figura 15 y Figura 16 se pueden observar dos ejemplos de las gráficas generadas, en este caso en la base de datos IoTD20:

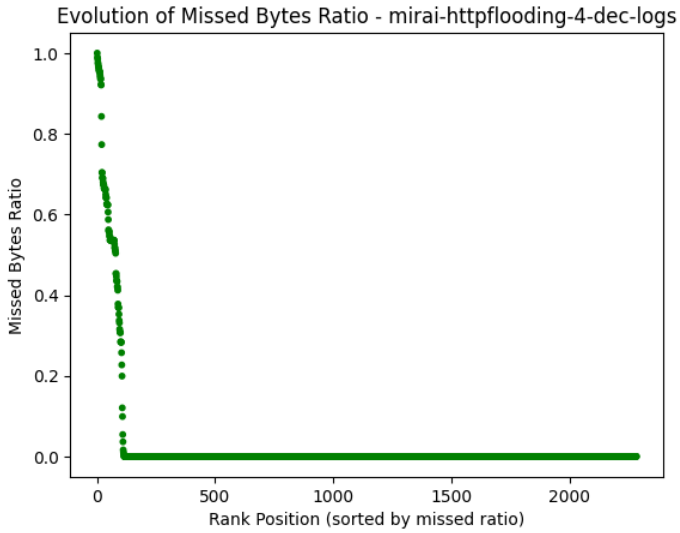


Figura 15: Porcentaje de Bytes perdidos para cada flujo de tráfico.

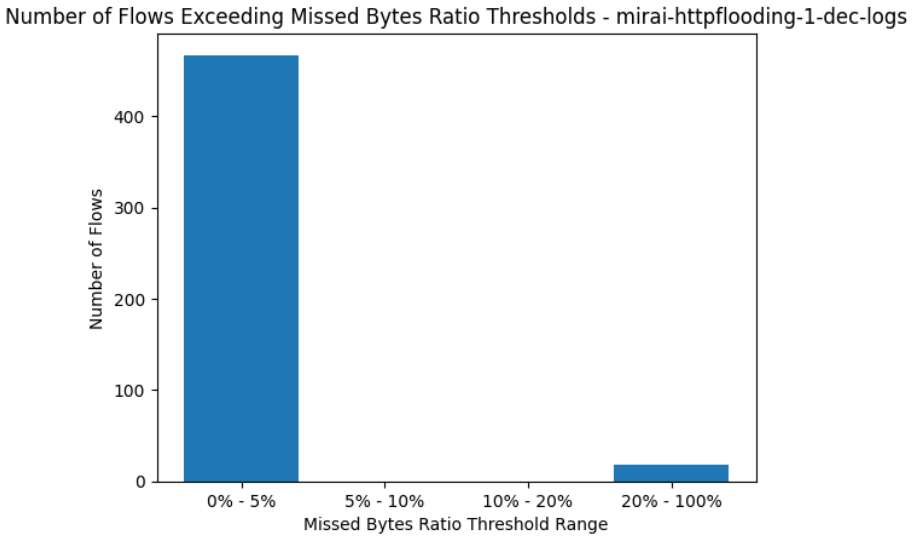


Figura 16: Distribución de número de flujos según porcentaje de bytes perdidos.

Como se puede apreciar, varios flujos de tráfico presentan pérdidas considerables de *bytes*. Por ello, mediante los *scripts* adjuntos en el Anexo IX, se decidió identificar todos aquellos flujos que superasen el 1% de *bytes* perdidos, extraerlos de los ficheros que contenían todos los logs de cada conjunto de datos, y almacenarlos por separado para poder eliminarlos.

En la Tabla VII se presentan estadísticas para cada base de datos y los flujos eliminados:

Tabla VII: Número de flujos de tráfico totales y eliminados.

Dataset	**Flujos totales**	**Flujos eliminados**	**Flujos > 10% *missed bytes***
IoTD20	123185	1765	1631
IoT-23	128693450	30	16
CIC-IoT-2023	205611728	78050	51570

Se consideró relevante su eliminación, ya que introducirían información errónea a los modelos. Esto sucede debido a que, si se intentan relacionar los parámetros de *bytes* enviados con el número de paquetes enviados, no concuerda. En la Tabla VIII se muestra un ejemplo de ello. El número de paquetes de respuesta es demasiado pequeño y no corresponde con la cantidad de *bytes* de respuesta.

Tabla VIII: Ejemplo de atributos para un flujo con pérdida de bytes.

Atributo	**Valor**
orig_*bytes*	9936
resp_*bytes*	1478577359
missed_*bytes*	1478547639
orig_pkts	962
orig_ip_*bytes*	56270
resp_pkts	3044
resp_ip_*bytes*	4437560

Para la lectura de los logs una vez fueron convertidos a formato csv, se emplearon las librerías Pandas y Dask para poder realizar el resto de operaciones de tratamiento de datos. Ambas librerías, convierten el archivo leído en un formato de tabla llamado *Dataframe*. Lee la cabecera del archivo csv, que contiene el nombre de cada una de las columnas (siendo estas las características), y permite realizar operaciones sobre columnas completas o sobre filas, en función de las necesidades. Con Pandas, es posible realizar una amplia gama de operaciones de limpieza y transformación de datos, como eliminar columnas, filtrar filas, y reemplazar valores faltantes o incorrectos. Pero esta presenta un inconveniente, requiere muchos recursos de memoria RAM si el tamaño de los datos es elevado, y, si bien es cierto que puede solventar este problema realizando las tareas de forma incremental fragmentando los datos en *chunks* o bloques, no es capaz de realizar determinadas tareas en las que el tamaño de las bases de datos supera al espacio en memoria, también llamadas tareas OOM (*Out Of Memory*).

Una vez filtrados los conjuntos de datos, el siguiente paso que se llevó a cabo fue el etiquetado de los flujos. Primero se recrearon las etiquetas originales para agruparlas más tarde en los grupos de ataques deseados. Para ello, se siguieron las reglas empleadas por los autores de los *datasets*, véase Anexo VIII y se generaron los *scripts* correspondientes, véase Anexo VII. Una vez obtenidas las etiquetas, se pudo observar que existía un patrón de ataques común entre los tres conjuntos de datos, por lo que se creó otro grupo de etiquetas nuevo para poder unificar las categorías de ataques entre *datasets*. Estos tres valores son: *DoS, Scan, Brute Force y Benign.* Se eliminaron determinados flujos de la base de datos IoT-23 que quedaron excluidos de este nuevo grupo de etiquetas.

Siguiendo el esquema de la Figura 14, el último paso previo al entrenamiento de los modelos de aprendizaje automático es la "limpieza" de datos y codificación de aquellos atributos que lo necesiten. Se seleccionan todos los atributos excepto las direcciones IP destino, IP origen, puerto origen y puerto destino, que son excluidas del estudio, así como el ID del flujo y el *TimeStamp*, puesto que todos ellos se han utilizado para definir un mismo flujo de tráfico o están asociados a los paquetes pertenecientes al mismo. Al considerarse que forma parte de la información que define el flujo, para un mejor entrenamiento y creación de los modelos, es mejor no disponer de información de puertos y direcciones IP en dicha fase. Si los ataques se generaron desde la misma IP y en los mismos puertos, no se representará fielmente la realidad y serán fácilmente identificables como ataques al aparecer flujos de tráfico con los mismos valores tanto en el conjunto de *training* como en el de *test*, pudiendo llevar al modelo a generar respuestas dependientes de la implementación de los escenarios y por ello se ha optado por no emplear ninguno de dichos atributos. Adicionalmente, se decidió eliminar el campo *tunnel_parents* ya que estaba vacío en la gran mayoría de flujos de los tres conjuntos de datos.

Analizando todos los posibles valores de las bases de datos, se observó que en muchos casos había valores vacíos para determinados atributos. Por ello, se rellenaron aquellos campos sin valor con el carácter necesario, siguiendo la Tabla IX. Además, se unificaron los valores de los campos "*local_orig*" y "*local_resp*", ya que se detectaron varios valores que representaban el concepto "*True*" y "*False*". El código empleado para poder lograrlo se adjunta en el Anexo XI.

Tabla IX: Valores referidos a campos vacíos y valores sustitutos.

Atributo	Valores sustituidos	Valor sustituto
Atributos numéricos	"", " ", "-", "[]", <NA> o NaN	0
"*history*" y "*conn_state*"	"", " ", "[]", <NA> o NaN	"-"
"*service*" y "*proto*"	"", " ", "-", "[]", <NA> o NaN	"*unknown*"
"*local_orig*" y "*local_resp*"	"T", "F"	"*True*" o "*False*"

Tras unificar los valores, se procedió a codificar lo valores de los atributos categóricos, siendo estos: "*history*", "*conn_state*", "*service*", "*proto*" "*local_orig*" y "*local_resp*". Como se buscaba que la codificación fuese común entre los tres conjuntos de datos para posteriormente realizar pruebas con los datos de los *datasets* combinados, el primer paso fue obtener los valores únicos estos atributos mediante la función *unique* o *drop_duplicates*. La librería empleada para la codificación, *Scikit-Learn,* es una biblioteca de Python especializada en *Machine Learning* y análisis de datos. *Scikit-Learn t*ambién proporciona herramientas robustas para la selección de modelos, preprocesamiento de datos, ajuste de modelos, validación de modelos y evaluación de resultados. En la siguiente sección se utiliza para realizar la clasificación de los datos.

Para optimizar todo el proceso de manipulación y limpieza de datos, se realizó de forma paralela mediante la clase ThreadPoolExecutor. Esta facilita la ejecución de operaciones de entrada/salida y otros trabajos en paralelo, aprovechando múltiples hilos de ejecución. Fue especialmente útil ya que en este trabajo se tratan grandes volúmenes de datos que pueden dividirse en fragmentos más manejables. En este caso, se empleó para leer y procesar múltiples archivos simultáneamente, aplicando transformaciones como la limpieza de datos o la normalización en paralelo. Esto no solo mejora el rendimiento, sino que también reduce significativamente el tiempo de procesamiento en comparación con la ejecución secuencial. Gracias a ella, se aprovecharon los recursos de CPU y RAM de forma eficiente.

Una vez codificados los atributos, los conjuntos de datos ya están listos para ser entregados a los modelos de *Machine Learning.* Se organizó el escenario de pruebas en dos subgrupos, por un lado, se obtuvieron resultados alimentando a los modelos con los conjuntos de datos de forma separada para así comparar los resultados obtenidos con los resultados de los autores; por otro lado, se entrenó primero el modelo con una base de datos y se evaluó con las dos restantes, y en otra prueba se entrenó y evaluó el modelo con una mezcla de los tres conjuntos de datos.

A continuación, se realiza una comparación entre los flujos obtenidos con la herramienta Zeek y los flujos originales de los *datasets*. Los flujos de CIC-IoT-2023 se construyeron a partir de archivos .pcap utilizando la herramienta CICFlowMeter, que genera archivos .csv donde cada fila corresponde a un flujo. CICFlowMeter define un flujo como un intercambio bidireccional de paquetes de red que pertenecen a la misma tupla de dirección IP de origen, dirección IP de destino, puerto de origen, puerto de destino y protocolo de capa de transporte, dentro de un período de tiempo determinado. Un flujo finaliza cuando se agota el tiempo de espera o cuando se cierra la conexión. La estructura de datos de Zeek es una conexión que sigue los mecanismos típicos de identificación de flujo, siguiendo el enfoque de 5 tuplas mencionado anteriormente. Para un protocolo orientado a la conexión como TCP, la definición de una conexión es más clara; sin embargo, para otros como UDP e ICMP, Zeek implementa una abstracción similar a un flujo para agregar paquetes. Cada paquete pertenece a una conexión. En la Tabla X se muestra el número de flujos tanto en los *datasets* originales como los obtenidos en este trabajo utilizando Zeek. En IoTD20 es importante recalcar que los autores obtuvieron los flujos mediante la herramienta CIC-FlowMeter. Además, el número de flujos obtenidos con Zeek para este trabajo no incluye aquellos que contienen pérdidas de *bytes* superiores al 1%.

Tabla X: Flujos de cada conjunto de datos.

Dataset	**Fichero**	**Flujos originales**	**Flujos Zeek**
IoTD20	*Mirai-UDP Flooding*	183554	500
	Mirai-Ackflooding	55124	38609
	Mirai-Hostbruteforce	121181	163
	Mirai-HTTP Flooding	55818	3882
	DoS-Synflooding	59391	59489
	Scan Port OS	53073	35
	Scan Hostport	22192	16251
	Normal	40073	12211
IoT-23	*Malware-Capture-7-1*	11454715	11454714
	Malware-Capture-34-1	23153	8990
	Malware-Capture-35-1	10447788	8257565
	Malware-Capture-43-1	67321810	67321799
	Malware-Capture-44-1	238	228
	Malware-Capture-48-1	3394346	3393634
	Malware-Capture-49-1	5410562	6021586
	Malware-Capture-52-1	19781379	32232712
	Honeypot-Capture-4-1	453	735
	Honeypot-Capture-5-1	1375	1358
	Honeypot-Capture-7-1	131	131
CIC-IoT-2023	*Mirai Greeth*	991867	193689830
	Mirai Greip	751683	144690088
	Mirai UDP Plain	890577	286556
	PortScan	82285	207402
	OSScan	98260	182672
	DDoS HTTP	28791	616156
	DDoS PSHACK	4094756	69941338
	DictionaryBruteForce	13065	6735
	Benign	1098196	589912

4.4. Aplicación de técnicas de ML

Como se comentará más adelante, para la generación de resultados se dividen los *datasets* en conjunto de entrenamiento y conjunto de *test*. Para ello, se empleó la función *train_test*_split, propia de la librería de Python *SciKit-Learn.* Se indica el porcentaje deseado de datos de *test*, en este caso un 60%, y se indica que realice la división de forma aleatoria.

En determinadas pruebas realizadas, la lectura del conjunto de datos en un sólo *DataFrame* requería más memoria de la disponible en la máquina. Por ello, se optó por emplear la librería *Dask*. *Dask* es una biblioteca de Python diseñada para manejar cálculos paralelos y distribuidos, permitiendo el procesamiento de grandes conjuntos de datos que no caben en la memoria de un solo ordenador. Dask extiende las funcionalidades de Pandas y NumPy para trabajar de manera eficiente con datos en entornos de big data. Al igual que Pandas, Dask permite leer y escribir datos en varios formatos, pero lo hace distribuyendo la carga de trabajo a través de múltiples hilos o incluso múltiples máquinas.

Una vez divididos, determinados modelos requieren un escalado de datos previo para funcionar correctamente. Estos son: MLP, *Nearest Centroid* y SGD. Para ello se emplea la clase StandardScaler(), que necesita ser entrenada con los datos a utilizar mediante la función fit(), y posteriormente se transforman para hacer efectivo el escalado (véase Anexo XII). Si se decide realizar la clasificación con sólo una fracción de las características, se realiza una selección de atributos. En este trabajo, una de las pruebas realizadas emplea esta técnica de preprocesado. Para ello, se empleó la función de *Scikit-Learn mutual_info_classif*, que obtiene una métrica que permite traducir numéricamente a una métrica qué atributos son más relevantes. Esta métrica se entrega a un selector de características, como por ejemplo KBest, y escoge el número de características deseado (véase Anexo XIII).

A continuación, se introducen en el modelo empleando la función fit() para su entrenamiento. Entre los algoritmos disponibles se incluyen regresión lineal, árboles de decisión, máquinas de soporte vectorial (SVM), k-vecinos más cercanos (KNN), y *clustering* con k-means, entre otros. Una vez entrenado el modelo, procede a ser evaluado con los datos de *test*. La evaluación devuelve una predicción de las etiquetas que el modelo ha estimado, y, a partir de estas y de las etiquetas originales, se realizan comparaciones y obtienen distintas métricas para evaluar el rendimiento del modelo. *Scikit-Learn t*ambién facilita la visualización de resultados a través de gráficos, por lo que se han generado curvas ROC y matrices de confusión para todos los modelos, lo que permite medir la eficacia de los modelos de manera detallada.

4.5. Entorno de trabajo de las pruebas

El puesto de trabajo empleado para el procesado de los datos y posterior clasificación posee las siguientes características:

Tabla XI: Especificaciones de la máquina empleada en el trabajo.

Sistema Operativo	Debian 11
Nº procesadores disponibles	16
Modelo CPU	Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz
Memoria RAM	156 GB
Zeek version	7.0.0-dev.247
Python version	Python 3.9.2
Sci-kit Learn version	1.4.2
Pandas version	2.2.2

5. Resultados

En este capítulo se presenta el proceso seguido para obtener los resultados tras aplicar las técnicas de clasificación descritas en el capítulo 3. Los diferentes modelos de clasificación fueron generados a partir de los tres conjuntos de datos descritos en el capítulo anterior por separado, así como a partir de un único *dataset* obtenido a partir de los tres. Para llevar a cabo la clasificación del tráfico se ha hecho uso de las técnicas de ML empleando la librería de *Scikit-Learn s*obre los archivos obtenidos que contienen todos los atributos de los flujos de tráfico (ficheros en formato .csv), tal y como se ha descrito en el capítulo 4. En la sección 5.1 se describe la metodología seguida en el banco de pruebas realizadas, y los resultados obtenidos y la discusión de los mismos se presentan en la sección 5.2. Las principales conclusiones de este capítulo se recopilan en el capítulo de Conclusiones.

5.1. Banco de pruebas realizadas

Tras conseguir tres conjuntos de datos con etiquetas comunes entre ambos, y un conjunto de datos obtenido a partir de la combinación de los tres anteriores, se procedió a realizar las correspondientes pruebas con diferentes algoritmos de *Machine Learning*.

La clasificación en todas las pruebas realizadas se ha llevado a cabo definiendo los conjuntos de entrenamiento y *test*, empleando el 40% del fichero para entrenamiento, y el resto para *test.* Se considera que es mejor no utilizar validación cruzada ya que, si el *dataset* se dividiese por ejemplo en 10 subconjuntos, empleando 9 partes para entrenar el modelo, aumentaría la probabilidad de que información de un mismo flujo de tráfico se hallase en el conjunto de entrenamiento y en el de prueba, conllevando un sobreajuste (*overfitting*) de clasificación. Para la clasificación se seleccionaron los algoritmos: *Decission Tree, Nearest Centroid, Random Forest, Gaussian Naïve Bayes, Bernoulli Naïve Bayes, Stochastic Gradient Descend, Bagging* con *Decission Tree, AdaBoost* con *Decission Tree* y *Multilayer Perceptron.*

El banco de pruebas llevado a cabo se ha organizado en **dos etapas** (véanse Figura 17 y Figura 18), realizando por un lado, la evaluación sobre los *datasets* IoTD20, IoT-23 y CIC-IoT-2023 de forma independiente (Figura 17); y por otro lado, se lleva a cabo la evaluación empleando los tres *datasets* de forma combinada (Figura 18). En la **primera etapa de pruebas** se definen los conjuntos de entrenamiento (*training*) y de prueba (*test*) en cada *dataset*. Se generan los correspondientes modelos de clasificación (aplicando las diferentes técnicas de ML) en cada conjunto de entrenamiento, y posteriormente, cada modelo es evaluado sobre el conjunto de prueba asociado, tal y como se ilustra en la Figura 17.

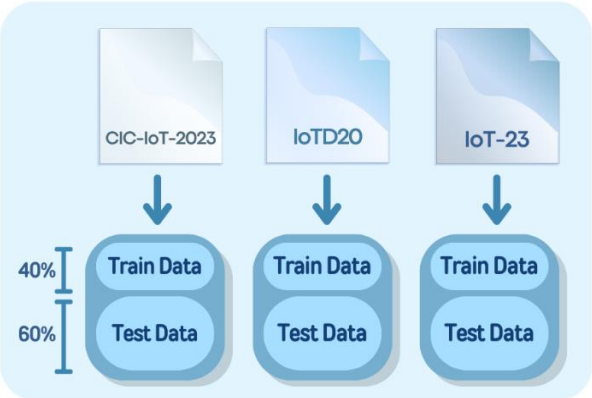


Figura 17: Primera etapa de pruebas: evaluación sobre los datasets IoTD20-Zeek, IoT-23-Zeek y CIC-IoT-2023-Zeek de forma independiente.

En la **segunda etapa de pruebas**, se consideran dos escenarios de evaluación. Se plantea por un lado el entrenamiento y evaluación de los modelos con una mezcla aleatoria de los tres *datasets* (**escenario 1** de la Figura 18), y por otro, el entrenamiento mediante un único conjunto de datos (CIC-IoT-2023-Zeek) y la evaluación sobre los otros dos: IoTD20-Zeek e IoT-23-Zeek (**escenario 2** de la Figura 18). En este punto del proceso (véase la Figura 14), se distinguen además dos formas diferentes de llevar a cabo la definición de los conjuntos de *training* y *test*: la primera corresponde a las pruebas **de clasificación multiclase (*multiclass*)**, y la segunda corresponde a las pruebas de **clasificación binaria (*binary*)**. En la clasificación multiclase se distinguen los diferentes tipos de ataques determinados por las etiquetas comunes (e.g. *Scan,* DoS, etc.). Por otro lado, para llevar a cabo la clasificación binaria, se han agrupado los ataques presentes en cada archivo mediante un *script* de Python. De este modo, se han eliminado las etiquetas que los distinguían y se ha incluido una genérica que indica 0 para tráfico benigno, y 1 para tráfico maligno. Por lo tanto, el clasificador tendrá que llevar a cabo una distinción binaria entre 0 y 1.

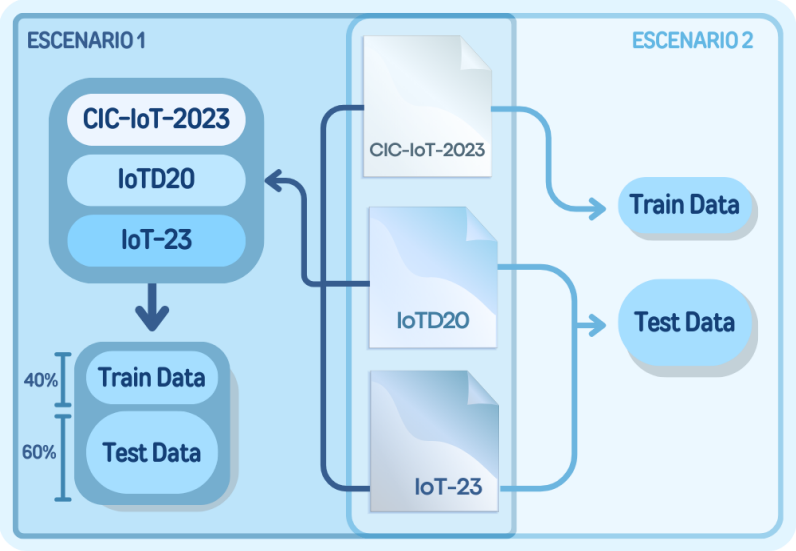


Figura 18: Segunda etapa de pruebas: evaluación sobre el dataset combinado a partir de IoTD20-Zeek, IoT-23-Zeek y CIC-IoT-2023-Zeek (escenario 1), y evaluación sobre IoTD20-Zeek e IoT-23-Zeek mediante el modelo generado con CIC-IoT-2023-Zeek.

5.2. Resultados y discusión

5.2.1. Primera etapa de pruebas: clasificación individual sobre IoTD20, IoT-23 y CIC-IoT-2023 (clasificación multiclase)

En esta sección se presentan y analizan los resultados obtenidos al aplicar las técnicas de ML para la clasificación multiclase de los flujos de tráfico, por lo tanto, diferenciando entre los distintos ataques que hay en un mismo fichero, y utilizando todos los atributos disponibles (42 características para cada flujo). En la Tabla XII se muestran los resultados de rendimiento correspondientes al parámetro F1 para las diferentes técnicas, mientras que los resultados de los parámetros de *precision* y *recall*, pueden consultarse en las correspondientes tablas del Anexo XIV. Como comentario general de los resultados obtenidos sobre cada uno de los *datasets* (Tabla XII), se puede decir que **los métodos de ML han funcionado con elevadas tasas de clasificación en la mayoría de los tipos de ataque y en la clase benigna**.

En general se observa un mejor comportamiento en la clasificación de los flujos de tipo DoS y *Scan* respecto a la clase *Brute Force*, lo que resulta razonable teniendo en cuenta que los ficheros de los *datasets* contienen clases desbalanceadas (el número de instancias de flujos *Brute Force* respecto a flujos de otros ataques es mucho menor, véase la Tabla X de la sección 4.3). Como es habitual, se presenta la dicotomía clásica entre dos enfoques: diseñar *datasets* que favorezcan la presencia de clases minoritarias o considerar una situación que refleje lo más fielmente posible un entorno real en términos de la frecuencia de ataques presentes.

Al comparar los resultados de las diferentes técnicas (Tabla XII) se puede observar que **destaca el rendimiento de los algoritmos *Decision Tree, Random Forest, Bagging Tree, Boosting Tree y MLP,*** que obtienen valores de F1 por encima del 98% en 7 de los 8 ataques presentes en los tres *datasets* (solo el caso de *Brute Force* en CIC-IoT-2023 se halla por debajo de dicho valor). Como puede observarse también, el valor de F1 promediado para todas las clases de tráfico en cada *dataset* (incluyendo ataques y benigno) supera el valor de 0,99 mediante la aplicación de estos algoritmos. En el *dataset* CIC-IoT-2023, la clasificación fue más compleja, lo que puede apreciarse en los resultados. Principalmente, se debe al desbalance de clases, que se hace presente de forma más visible en este conjunto de datos, ya que la diferencia de número de instancias entre las clases *Brute Force*, y por ejemplo DoS, es del orden de casi 10^5. En los tres conjuntos de datos, los algoritmos ***Decision Tree, Random Forest, Bagging Tree y Boosting Tree*** obtienen resultados muy similares, pero *Decision Tree* lo lleva a cabo en el menor tiempo, por lo que puede ser un buen candidato para tareas más enfocadas a analizar datos en tiempo real. En esta primera etapa de pruebas no se consideró necesario llevar a cabo la clasificación binaria dado que los resultados en clasificación multiclase (más compleja que la clasificación binaria) son excelentes.

Se incluyen a continuación algunos resultados del **análisis de la complejidad computacional** de los algoritmos utilizados. Como ejemplo, en la Figura 19 se presentan los tiempos de ejecución (escala logarítmica en segundos) para el *dataset* CIC-IoT-2023 tanto en la fase de construcción del modelo (*training*) como en la de evaluación del mismo (*test*). Tal y como se podía prever, aquellas técnicas de mayor complejidad computacional dan lugar a tiempos de cálculo más elevados y, por tanto, a un menor número de flujos por segundo analizados. En los otros dos *datasets* evaluados se observó este mismo comportamiento en los tiempos de ejecución de las técnicas de ML aplicadas.

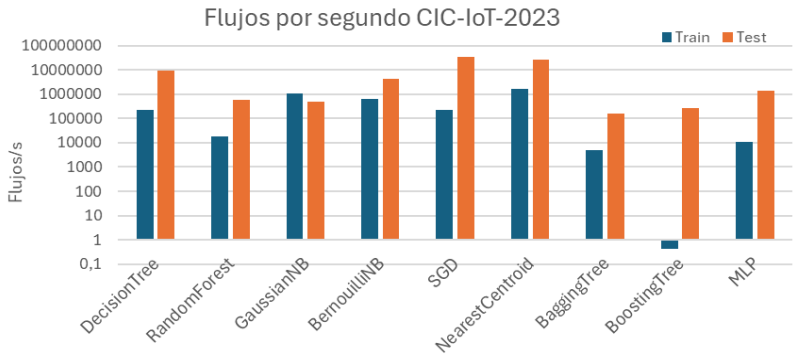


Figura 19: Tiempos de cálculo (expresado en flujos por segundo analizados) de los algoritmos de ML en el dataset CIC-IoT-2023.

Tabla XII: Medida F1 para la primera etapa de pruebas, clasificación multiclase individual de IoTD20, IoT-23 y CIC-IoT-2023.

Dataset name	Label	*DecissionTree*	Gaussian NB	BernouilliNB	SGD	*Random Forest*	*BaggingTree*	BoostingTree	NearestCentroid	MLP
IoTD20	*Brute Force*	1,00	0,01	0,15	0,42	1,00	1,00	0,98	0,06	1,00
	DoS	1,00	0,57	0,9	0,98	1,00	1,00	1,00	0,88	0,99
	Scan	1,00	0,00	0,97	0,98	1,00	1,00	1,00	0,64	1,00
	Benign	0.96	0,10	0,26	0,7	0,96	0,96	0,95	0,39	0,92
	Average	0,99	0,46	0,86	0,96	0,99	0,99	0,99	0,81	0,99
IoT-23	DoS	1,00	0,84	0,81	0,99	1,00	1,00	1,00	0,86	1,00
	Scan	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,98	1,00
	Benign	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,98	1,00
	Average	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,98	1,00
CIC-IoT-2023	*Brute Force*	0,84	0,00	0,00	0,00	0,83	0,88	0,85	0,01	0,47
	DoS	1,00	0,21	0,94	1,00	1,00	1,00	1,00	1,00	1,00
	Scan	0,98	0,00	0,05	0,59	0,99	0,99	0,99	0,63	0,96
	Benign	0,98	0,01	0,03	0,41	0,97	0,98	0,98	0,37	0,88
	Average	1,00	0,21	0,94	1,00	1,00	1,00	1,00	0,99	1,00

A continuación, se presentan algunos ejemplos del funcionamiento de los clasificadores que mejores resultados han logrado. Una muestra de ello es el árbol de decisión creado por *Decision Tree para la* base de datos de IoTD20. En las primeras divisiones, es capaz de diferenciar una clase con una probabilidad de 0,938 empleando características como *history*, *proto* u *orig_ip_bytes*.

En la Figura 20 se presentan, respectivamente, la matriz de confusión genérica y la matriz de confusión normalizada para el algoritmo *Random Forest* en el conjunto de datos IoT-23, donde se puede apreciar que el número de instancias incorrectamente clasificadas es muy reducido.

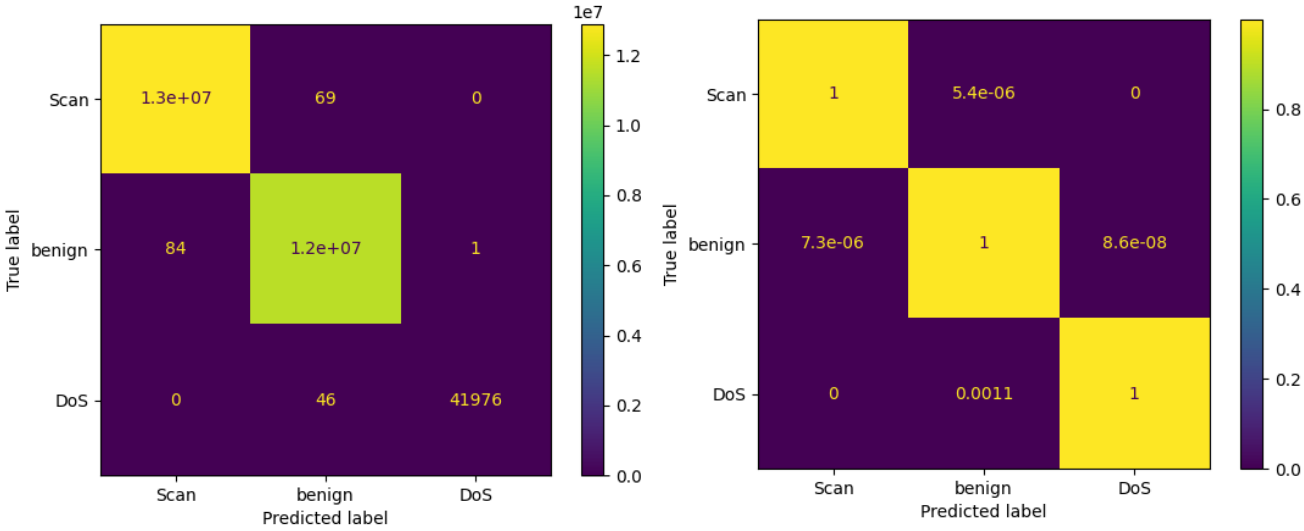


Figura 20: Matrices de confusión para IotD20 utilizando RandomForest.

Si comparamos los resultados de nuestro banco de pruebas con los resultados obtenidos por los autores de las respectivas bases de datos, por ejemplo, para el conjunto de datos CIC-IoT-2023, podemos apreciar resultados considerablemente mejores en términos de F1 en nuestro estudio. La Tabla XIII muestra los resultados obtenidos en (Neto et al., 2023) sobre todo el *dataset*, incluyendo pruebas con diferentes clases de ataques. Comparando con las clasificaciones de 8 y 2 clases, se puede observar que los algoritmos *Perceptron, Adaboost* y *Random Forest*, también utilizados en este trabajo, obtienen resultados similares a los obtenidos en la tabla. Al igual que en este trabajo, *Random Forest* y Adaboost (análogo a ***Boosting Tree)*** son aquellos con mejores resultados, que en nuestro banco de pruebas han alcanzado valores de F1 incluso más elevados.

Tabla XIII: Medida F1 obtenida por los autores de CIC-IoT-2023.

		Logistic regression	**Perceptron**	**Adaboost**	**Random Forest**	**Deep Neural Network**
CIC-IoT-2023 (Neto et al., 2023)	**Binary**	0,88	0,81	0,96	0,96	0,94
	8 classes	0,54	0,55	0,37	0,72	0,70

También se llevó a cabo la comparación con los resultados obtenidos por los autores de IoTD20 (Ullah & Mahmoud, 2020). Si comparamos los valores de F1 presentados en dicho trabajo, podemos apreciar ciertas similitudes en el comportamiento de los algoritmos. No obstante, en nuestro estudio, **los algoritmos *Decision Tree, Random Forest, Bagging, BoostingTree y MLP*** mostraron valores de F1 muy superiores a los alcanzados en (Ullah & Mahmoud, 2020).

Tabla XIV: Medida F1 obtenida por los autores de IoTD20.

		SVM	Gaussian NB	LDA	Logistic regression	Decision Tree	Random Forest	Ensemble
IoTD20 (Ullah & Mahmoud, 2020)	**Binary**	0,16	0,62	0,70	0,30	0,88	0,84	0,87

Finalmente, respecto al conjunto de datos de IoT-23, los autores no presentaron ningún resultado de evaluación, y sólo se dispone del propio *dataset* e información relacionada con el etiquetado de los flujos.

5.2.2. Segunda etapa de pruebas, escenarios 1 y 2: clasificación sobre IoTD20, IoT-23 y CIC-IoT-2023 combinados (clasificaciones multiclase y binaria)

Escenario 1

Tras haber obtenido las métricas para los tres conjuntos de datos de forma individual, se plantearon dos escenarios de pruebas más en una segunda etapa. A continuación, se analiza el primer escenario, en el que se entrenan los modelos con un subconjunto de la unión de los tres *datasets*, y se evalúa con el subconjunto restante (Figura 18). Al igual que en el análisis de la sección previa, se emplea el 40% de los datos para el entrenamiento y 60% para el *test*. Para este **escenario 1**, la división en subconjuntos de entrenamiento y de evaluación se llevó a cabo previamente sobre cada conjunto de datos, y posteriormente se unieron los correspondientes subconjuntos de entrenamiento y *test* entre sí. Se decidió evaluar primero la clasificación de forma binaria, ya que computacionalmente es más simple, y posteriormente se llevó a cabo con las distintas clases de ataque.

A continuación, se presentan y analizan los resultados de clasificación binaria, mostrados en la Tabla XV. Se puede apreciar que los valores de F1 son superiores a 0.99 para todos los clasificadores excepto GaussianNB, BernouilliNB y Nearest Centroid. Esto se puede deber a la complejidad de los datos, ya que, en este caso, se está intentando obtener el perfil de tráfico benigno y de ataques procedentes de conjuntos de datos diferentes, generados mediante procedimientos y dispositivos IoT diferentes. Por lo tanto, algoritmos más simples no consiguen diferenciarlos de forma correcta. Para todos los algoritmos basados en árboles (***Decision Tree, Random Forest, Bagging Tree, y Boosting Tree***) se han obtenido valores de F1=1, lo que pone de manifiesto que al utilizar un conjunto de entrenamiento combinado se incluyen los diferentes comportamientos de los flujos de tráfico provenientes de distintos *datasets* y se pueden lograr resultados de clasificación muy buenos.

Después, se realizó la clasificación multiclase cuyos resultados también se muestran en la Tabla XV. En general, se han obtenido valores de F1 elevados para todas las clases de tráfico, exceptuando la clase minoritaria *Brute Force*, en la que se aprecia un descenso notable. Para el resto de clases, todos los clasificadores excepto GaussianNB y BernoulliNB han obtenido valores de F1 superiores a 0,99.

Como se puede observar en el ejemplo de la Figura 20, en esta clasificación, el número de instancias incorrectamente clasificadas como benignas siendo ataques, y el tráfico benigno clasificado como ataque, es inusualmente alto. Tras analizar los conjuntos de datos, se plantea la posibilidad de que se deba a que el etiquetado realizado por los autores de CIC-IoT-2023 difiere respecto al empleado en los *datasets* restantes. Mientras que en CIC-IoT-2023 se etiquetan todos los flujos de una misma captura de tráfico como ataque o como benigno, en IoT-23 e IoTD20 se proporcionan reglas específicas para separar flujos o paquetes malignos del tráfico benigno que se encuentran en las capturas de ataque.

Figura 21: Matriz de confusión para clasificador Bagging

Tabla XV: Medida F1 para la segunda etapa de pruebas.

Dataset name	Label	*DecissionTree*	Gaussian NB	BernouilliNB	SGD	*Random Forest*	BaggingTree	BoostingTree	NearestCentroid	MLP
Escenario 1 Binary	Benign	1,00	0,4	0,82	0,99	1,00	1,00	1,00	0,63	0,99
	Malign	1,00	0,15	0,94	1,00	1,00	1,00	1,00	0,79	1,00
	Average	1,00	0,21	0,91	0,99	1,00	1,00	1,00	0,75	1,00
Escenario 1 Multiclass	*Brute Force*	0,41	0,00	0,00	0,00	0,44	0,44	0,41	0,02	0,3
	DoS	1,00	0,21	0,86	0,99	1,00	1,00	1,00	0,98	1,00
	Scan	1,00	0,00	0,91	0,99	1,00	1,00	1,00	0,91	1,00
	Benign	1,00	0,00	0,83	0,99	1,00	1,00	1,00	0,91	1,00
	Average	1,00	0,11	0,87	0,99	1,00	1,00	1,00	0,95	1,00
Escenario 1 Multiclass & FS	*Brute Force*	0,38	0,00	0,00	0,00	0,42	0,42	0,39	0,02	0,00
	DoS	1,00	0,21	0,73	0,99	1,00	1,00	1,00	0,99	1,00
	Scan	1,00	0,00	0,97	0,93	1,00	1,00	1,00	0,93	0,99
	Benign	1,00	0,00	0,00	0,90	0,99	0,99	0,99	0,91	0,99
	Average	1,00	0,11	0,62	0,90	1,00	1,00	1,00	0,96	0,99
Escenario 2 Binary	Benign	0,00	0,64	0,00	0,12	0,00	0,01	0,00	0,00	0,12
	Malign	0,69	0,00	0,69	0,01	0,69	0,69	0,69	0,69	0,01
	Average	0,37	0,30	0,37	0,06	0,37	0,37	0,37	0,37	0,06
Escenario 2 Multiclass	*Brute Force*	0,00	0,00	0,07	0,00	0,00	0,00	0,03	0,03	0,07
	DoS	0,00	0,00	0,00	0,01	0,01	0,00	0,01	0,01	0,56
	Scan	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,94	1,00
	Benign	0,00	0,00	0,00	0,12	0,24	0,24	0,24	0,00	0,99
	Average	0,12	0,00	0,00	0,06	0,12	0,12	0,12	0,49	0,99

Por último, se decidió analizar cuáles eran las características más relevantes para la clasificación incluidas en el banco de pruebas. Para ello se consideró este conjunto de datos generado a partir de la unión de los tres *datasets* correspondiente al escenario 1. Es bien sabido que las técnicas de selección de atributos (***feature selection)*** se utilizan frecuentemente como un paso previo al entrenamiento de los modelos de clasificación para reducir la dimensionalidad de los datos y mejorar la precisión al eliminar características irrelevantes y/o redundantes (Khalid et al., 2014). Por ejemplo, el método de la ganancia de información (*information gain*) mide la reducción en la entropía o incertidumbre de los datos tras dividirlos según un atributo específico, es decir, la ganancia de información con respecto a la clase. De este modo, el método de FS proporciona una lista ordenada de atributos en la que se prioriza la reducción de la entropía sobre la cantidad de información que aporta. En Tabla XVI se muestra el resultado obtenido al aplicar el método de FS donde los atributos se hallan ordenados según su *information gain.* Obviamente, existen otros métodos de FS más sofisticados, por ejemplo, basados no en un ranking individual de atributos sino en la selección de subconjuntos óptimos de atributos, como el método CFS (*Correlation-based Feature Subset Selection*) en el que se considera la capacidad predictiva individual de los atributos junto con el grado de redundancia entre ellos (Rodríguez et al., 2022). Para poder tener una primera aproximación al efecto de incorporar los métodos de FS, se realizó la clasificación con los 15 primeros atributos según su valor de *information gain*. Los resultados de F1 obtenidos se muestran en laTabla XV. Los resultados obtenidos en esta clasificación son similares o idénticos a los alcanzados sin aplicar la selección de atributos, excepto en algún caso aislado como Bernoulli para la clase benigna. La aplicación de FS puede ser una opción interesante cuando el tiempo de cálculo es crítico, dado que en la mayoría de los casos éste se ha visto reducido considerablemente.

Tabla XVI: Listado de características ordenadas según la media Infomation Gain.

nº	Característica	InfoGain	nº	Característica	InfoGain
11	orig_ip_*bytes*	0.939904	12	resp_pkts	0.011722
28	orig_pkts_cero	0.804201	4	resp_*bytes*	0.004483
33	time_max	0.622759	25	resp_*bytes*_max	0.004215
2	duration	0.622723	19	resp_*bytes*_mean_nocero	0.004075
30	time_mean	0.622511	15	resp_*bytes*_mean	0.003886
31	time_std	0.622102	38	resp_time_mean	0.001945
10	orig_pkts	0.558826	41	resp_time_max	0.001838
9	history	0.367382	27	resp_pkts_nocero	0.001695
34	orig_time_mean	0.100194	17	resp_*bytes*_std	0.001382
37	orig_time_max	0.099999	39	resp_time_std	0.001152
18	orig_*bytes*_mean_nocero	0.048777	21	resp_*bytes*_std_nocero	0.001078
24	orig_*bytes*_max	0.048595	32	time_min	0.000009
3	orig_*bytes*	0.048549	0	proto	0.000000
14	orig_*bytes*_mean	0.048504	8	missed_*bytes*	0.000000
35	orig_time_std	0.044491	6	local_orig	0.000000
16	orig_*bytes*_std	0.042997	5	conn_state	0.000000
20	orig_*bytes*_std_nocero	0.042741	1	service	0.000000
26	orig_pkts_nocero	0.034696	22	orig_*bytes*_min	0.000000
13	resp_ip_*bytes*	0.020950	23	resp_*bytes*_min	0.000000
7	local_resp	0.017107	36	orig_time_min	0.000000
29	resp_pkts_cero	0.016615	40	resp_time_min	0.000000

Escenario 2

El **escenario 2** de la segunda etapa de pruebas consistió en considerar como conjunto de entrenamiento un *dataset*, y como conjunto de evaluación, otro distinto. En concreto, se empleó el conjunto de datos CIC-IoT-2023 para entrenamiento y se evaluó con IoTD20 e IoT-23. Primero se analizan los resultados de utilizar el *dataset* de CIC-IoT como conjunto de *train*, tanto para clasificación binaria como multiclase. En esta prueba, se decidió muestrear la clase DoS en un 15%, para que la computación fuese viable, manteniendo intactas las otras tres clases. Tras el muestreado, el número de instancias de DoS pasó a ser de 30 millones, superando igualmente al resto de clases en un factor 100.

Los resultados obtenidos en la clasificación binaria (véase la Tabla XV) indican que la clase "maligna" se etiqueta de forma correcta, pero surgen dificultades a la hora de etiquetar correctamente la clase benigna. Esto puede deberse al método de etiquetado ya comentado anteriormente. Si existe tráfico realmente benigno dentro de las capturas de tráfico malignas, pero está siendo considerado durante el entrenamiento con la etiqueta "maligna", en la evaluación, al analizar los flujos benignos de IoTD20 e IoT-23, estos podrán ser etiquetados como malignos. Esto se puede observar en la matriz de confusión para el algoritmo *Random Forest* mostrada en la Figura 22. Los resultados ponen de manifiesto en este caso que al generar el modelo de clasificación en un *dataset* y evaluarlo en otro no se logran incluir los diferentes comportamientos de los flujos de tráfico provenientes de distintos *datasets*, o bien como se ha comentado que el propio etiquetado de los flujos en cada *dataset* no los hace coherentes, y esto se traduce en resultados de clasificación mucho peores. El descenso en los valores de F1 es especialmente significativo en todos los algoritmos basados en árboles.

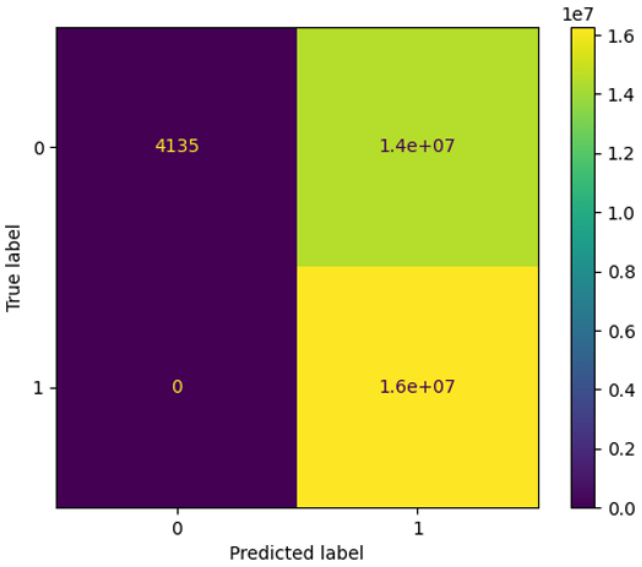


Figura 22: Matriz de confusión para clasificación binaria en el segundo escenario empleando Random Forest

Si ahora se analizan las métricas obtenidas para la clasificación multiclase (véase la Tabla XV), los valores de F1 siguen siendo muy reducidos. El algoritmo MLP obtiene los mejores resultados, y la matriz de confusión presenta una diagonal con valores significativos. Aunque se sigue observando que la etiqueta "*benign*" presenta un número muy alto de instancias etiquetadas como ataque, y la mayoría de clasificaciones erróneas se producen en la clase DoS. Esto puede deberse a la presencia predominante de la clase en el conjunto de entrenamiento.

6. Conclusiones y líneas futuras

6.1. Conclusiones

En este trabajo fin de máster se ha hecho un análisis del entorno IoT actual, desde los dispositivos más utilizados a los atacados con mayor frecuencia, las amenazas más comunes, detallando una de las más conocidas, la *botnet* Mirai, y los ataques típicos de esta, además de otros ataques comunes que están presentes en los conjuntos de datos empleados.

Además, se han estudiado en profundidad diferentes conjuntos de datos en el ámbito IoT disponibles en la actualidad. Todos ellos presentan gran diversidad de ataques, y en la mayoría de estos está presente el tráfico de *botnets*. Tras el análisis efectuado, se seleccionaron los conjuntos de datos IoTD20, IoT-23 y CIC-IoT-2023, organizados en capturas de tráfico. Dichos *datasets* contienen tráfico benigno y diferentes ataques: diferentes tipos de DDoS como Mirai GREETH, Mirai GREIP, Mirai UDPPlain, DDoS HTTP o DDoS PSHACK, ataque de fuerza bruta, escaneo de puertos y de sistema operativo.

También se han estudiado distintas técnicas de aprendizaje automático con el objetivo de clasificar los diferentes ataques característicos de una *botnet* y se ha profundizado en aquellas que han sido empleadas. Dentro de las técnicas de clasificación se escogieron nueve algoritmos para caracterizar y poder diferenciar el tráfico benigno de las clases malignas DoS, *Scan* y BruteFoce: *Decision Tree, Gaussi*an *Naive* Bayes, Bernouilli *Naive* Bayes, Stochastic Gradient Descent, *Random Forest*, Bagging con *Decision Tree, AdaBoo*st con *Decision Tree, Nearest*Centroid y Mulltilayer Perceptron.

Después, a partir de las capturas de tráfico, se analizaron los flujos y se extrajo la información detallada de las conexiones en distintos logs mediante la herramienta Zeek utilizando un *script* personalizado que proporciona datos a nivel de los protocolos IP, TCP, UDP, ICMP, etc., y se etiquetó el tráfico mediante *scripts* que comparan la dirección IP, los puertos origen y destino, el protocolo empleado y atributos como *history* y *conn_state*. Durante el etiquetado también se eliminaron algunos flujos que presentaron pérdidas de *bytes* debido a una posible mala configuración en la captura de tráfico por parte de los autores originales, y se llevaron a cabo las tareas de conversión de formato, limpieza de datos y unificación de estructura de determinados atributos, para finalmente obtener tres conjuntos de datos que contienen los mismos atributos y etiquetas.

Para aplicar los algoritmos de ML se emplearon las librerías *Scikit-learn,* Pandas y Dask que permiten llevar a cabo funciones de lectura de datos, tratamiento y transformación, preprocesado de datos, clasificación, selección de atributos, asociación y visualización de datos. Se planteó un extenso banco de pruebas considerando la evaluación sobre los *datasets* IoTD20-Zeek, IoT-23-Zeek y CIC-IoT-2023-Zeek de forma independiente y de forma combinada, tanto en clasificación binaria como multiclase. En cada una de estas pruebas se consideró un 40% del *dataset* para entrenamiento y el resto para *test*. Es importante señalar que no se incluyeron atributos como direcciones IP de origen o destino, ni puertos de origen o destino, debido a que estos valores son altamente dependientes de la implementación, lo que haría que el experimento no fuera realista. Además, la inclusión de estos atributos facilitaría la identificación de los ataques. Para medir cuantitativamente los resultados obtenidos, se analizó en profundidad la medida de rendimiento F1. A partir de los resultados obtenidos, se puede concluir que los algoritmos que alcanzan los mejores resultados, manteniendo un tiempo de cálculo razonable, son *DecisionTree, Random Forest, Boosting Tree* y *Bagging Tree*. Por lo

tanto, los métodos basados en árboles de decisión han demostrado ser los más eficientes, lo cual es beneficioso para su implementación en un entorno real, dado que estas técnicas son las más fácilmente interpretables. Estos algoritmos lograron valores de F1 superiores a 0.99 sin selección de atributos en el análisis individual de cada conjunto de datos, y superiores a 0.9 en la mayoría de las clases en todas las pruebas en las que se consideró la unión de conjuntos de datos, presentando además una variación mínima con la selección de atributos (con 15 atributos seleccionados usando el método de *Information Gain*). Un tiempo reducido para generar el modelo y realizar las pruebas es esencial si se pretende aplicar el modelo en un sistema que funcione en tiempo real. Los resultados obtenidos en la clasificación binaria mostraron también que al reducir la complejidad de la clasificación se pueden obtener mejores resultados en menos tiempo. En cuanto a la selección de atributos, es destacable la reducción de tiempo a costa de una ligera disminución en el valor de F1.

Es relevante destacar que los resultados obtenidos en la clasificación llevada a cabo al entrenar con un conjunto de datos diferente al de evaluación fueron subóptimos. Esto deja entrever la problemática que surge de generar conjuntos de datos siguiendo procedimientos tan diferentes. Aunque se obtengan resultados prometedores al evaluar de forma aislada estos conjuntos de datos, es necesario analizar el comportamiento de los modelos con datos generados por diferentes dispositivos y ataques capturados en condiciones diferentes. De este modo los modelos desarrollados podrán enfrentarse a situaciones más realistas, y así se podrá evaluar mejor el potencial de la aplicación de técnicas de *Machine Learning* a la detección de *botnets*.

6.2. Líneas futuras

A continuación se proponen algunas líneas futuras de investigación que se podrían abordar:

- Plantear la posibilidad de emplear Zeek en tiempo real, ya que las pruebas actuales se han basado en capturas de tráfico almacenadas.
- Analizar en mayor profundidad qué atributos de Zeek son más importantes para una clasificación efectiva.
- Analizar únicamente el primer minuto de cada fichero desde el inicio del ataque para comprobar si es posible detectar los ataques de forma temprana.
- Plantear el estudio y clasificación empleando técnicas de *Deep Learning*.
- Considerar la inclusión de información proveniente de otros ficheros generados por Zeek.
- Realizar capturas propias de tráfico benigno en diferentes hogares y con nuevos dispositivos IoT para mejorar los análisis efectuados.

Bibliografía

1.5. Stochastic Gradient Descent — scikit-learn 1.5.0 documentation. (n.d.). Retrieved June 25, 2024, from https://scikit-learn.org/stable/modules/sgd.html

ACK-PSH Flood | Knowledge Base | MazeBolt. (n.d.). Retrieved June 25, 2024, from https://kb.mazebolt.com/knowledgebase/ack-psh-flood/

Al Nuaimi, T., Al Zaabi, S., Alyilieli, M., AlMaskari, M., Alblooshi, S., Alhabsi, F., Yusof, M. F. Bin, & Al Badawi, A. (2023). A comparative evaluation of intrusion detection systems on the edge-IIoT-2022 dataset. *Intelligent Systems with Applications*, *20*. https://doi.org/10.1016/j.iswa.2023.200298

Altares, E., Salvio, J., & Tay, R. (2023). *2022 FORTINET IoT Threat Review*.

Armstrong, M. (2022). *Chart: Homes Are Only Getting Smarter | Statista*. https://www.statista.com/chart/27324/households-with-smart-devices-global-iot-mmo/

Chatzoglou, E., Kambourakis, G., & Kolias, C. (2021). Empirical Evaluation of Attacks against IEEE 802.11 Enterprise Networks: The AWID3 Dataset. *IEEE Access*, *9*, 34188–34205. https://doi.org/10.1109/ACCESS.2021.3061609

Cybersecurity Report 2023: Consumer Devices Under Threat. (2022).

da Costa, K. A. P., Papa, J. P., Lisboa, C. O., Munoz, R., & de Albuquerque, V. H. C. (2019). Internet of Things: A survey on machine learning-based intrusion detection approaches. *Computer Networks*, *151*, 147–157. https://doi.org/10.1016/J.COMNET.2019.01.023

Ferriyan, A., Thamrin, A. H., Takeda, K., & Murai, J. (2021). Generating network intrusion detection dataset based on real and encrypted synthetic attack traffic. *Applied Sciences (Switzerland)*, *11*(17). https://doi.org/10.3390/app11177868

García, J., Molina, J. M., Berlanga, A., Patricio, M. A., Bustamante, Á. L., & Padilla, W. R. (2018). *Ciencia de datos: técnicas analíticas y aprendizaje estadístico en un enfoque práctico.*

Garcia, S., Parmisano, A., & Erquiaga, M. J. (2020). *IoT-23: A labeled dataset with malicious and benign IoT network traffic*. https://doi.org/10.5281/ZENODO.4743746

Guerra-Manzanares, A., Medina-Galindo, J., Bahsi, H., & Nomm, S. (2020). MedBIoT: Generation of an IoT Botnet Dataset in a Medium-sized IoT Network. *International Conference on Information Systems Security and Privacy*, 207–218. https://doi.org/10.5220/0009187802070218

Imam, M., Paul Nir, M., Mahmoud, M., Nir, M., & Matrawy, A. (2014). A Survey on Botnet Architectures, Detection and Defences. In *International Journal of Network Security* (Vol. 0, Issue 0). https://www.researchgate.net/publication/259932835

Inside the infamous Mirai IoT Botnet: A Retrospective Analysis. (2017). https://blog.cloudflare.com/inside-mirai-the-infamous-iot-botnet-a-retrospective-analysis/

IoT botnet activity in consumer networks. (2023).

John, G. H. (1995). *338 Estimating Continuous Distributions in Bayesian Classifiers*. 338–345. http://robotics.stanford.edu/-gjohn/

Khalid, S., Khalil, T., & Nasreen, S. (2014). A survey of feature selection and feature extraction techniques in machine learning. *Proceedings of 2014 Science and Information Conference, SAI 2014*, 372–378. https://doi.org/10.1109/SAI.2014.6918213

Koroniotis, N., Moustafa, N., Sitnikova, E., & Turnbull, B. (2018). *Towards the Development of Realistic Botnet Dataset in the Internet of Things for Network Forensic Analytics: Bot-IoT Dataset*. http://arxiv.org/abs/1811.00701

Kubat, M. (2021). An Introduction to Machine Learning. In *An Introduction to Machine Learning*. Springer International Publishing. https://doi.org/10.1007/978-3-030-81935-4

Law, M. (2023). *Security essential in the growing Internet of Things network | Cyber Magazine*. https://cybermagazine.com/articles/security-essential-in-the-growing-internet-of-things-network

Lella, I., Tsekmezoglou, E., Theocharidou, M., Magonara, E., Malatras, A., Naydenov, R. S., & Ciobanu, C. (2023). *ENISA THREAT LANDSCAPE 2023*. https://doi.org/10.2824/782573

Meidan, Y., Bohadana, M., Mathov, Y., Mirsky, Y., Shabtai, A., Breitenbacher, D., & Elovici, Y. (2018). N-BaIoT-Network-based detection of IoT botnet attacks using deep autoencoders. *IEEE Pervasive Computing*, *17*(3), 12–22. https://doi.org/10.1109/MPRV.2018.03367731

Molina López, J. M., & García Herrero, J. (2006). *TÉCNICAS DE ANÁLISIS DE DATOS APLICACIONES PRÁCTICAS UTILIZANDO MICROSOFT EXCEL Y WEKA*.

Nearest Centroid Classification — scikit-learn 0.18.2 documentation. (n.d.). Retrieved June 25, 2024, from https://scikit-learn.org/0.18/auto_examples/neighbors/plot_nearest_centroid.html

Neto, E. C. P., Dadkhah, S., Ferreira, R., Zohourian, A., Lu, R., & Ghorbani, A. A. (2023). CICIoT2023: A Real-Time Dataset and Benchmark for Large-Scale Attacks in IoT Environment. *Sensors*, *23*(13). https://doi.org/10.3390/s23135941

Qing, Y., Liu, X., & Du, Y. (2023). MBB-IoT: Construction and Evaluation of IoT DDoS Traffic Dataset from a New Perspective. *Computers, Materials and Continua*, *76*(2), 2095–2115. https://doi.org/10.32604/cmc.2023.039980

¿Qué es la botnet Mirai? | Cloudflare. (n.d.). Retrieved June 25, 2024, from https://www.cloudflare.com/es-es/learning/ddos/glossary/mirai-botnet/

Rodríguez, M., Alesanco, Á., Mehavilla, L., & García, J. (2022). Evaluation of Machine Learning Techniques for Traffic Flow-Based Intrusion Detection. *Sensors*, *22*(23). https://doi.org/10.3390/s22239326

RST-FIN Flood | Knowledge Base | MazeBolt. (n.d.). Retrieved June 25, 2024, from https://kb.mazebolt.com/knowledgebase/rst-fin-flood/

The Mirai Botnet – Threats and Mitigations. (n.d.). Retrieved June 25, 2024, from https://www.cisecurity.org/insights/blog/the-mirai-botnet-threats-and-mitigations

The Riskiest Connected Devices in 2023. (2023).

Trajanovski, T., & Zhang, N. (2021). An Automated and Comprehensive Framework for IoT Botnet Detection and Analysis (IoT-BDA). *IEEE Access*, *9*, 124360–124383. https://doi.org/10.1109/ACCESS.2021.3110188

Ullah, I., & Mahmoud, Q. H. (2020). A Scheme for Generating a Dataset for Anomalous Activity Detection in IoT Networks. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *12109 LNAI*, 508–520. https://doi.org/10.1007/978-3-030-47358-7_52

What is Command and Control (C&C or C2) in Cybersecurity? - zenarmor.com. (2023). https://www.zenarmor.com/docs/network-security-tutorials/what-is-command-and-control-c2

Winward, R. (2018). *IoT Attack Handbook A Field Guide to Understanding IoT Attacks from the Mirai Botnet to Its Modern Variants*.

Zarpelão, B. B., Miani, R. S., Kawakani, C. T., & de Alvarenga, S. C. (2017). A survey of intrusion detection in Internet of Things. *Journal of Network and Computer Applications*, *84*, 25–37. https://doi.org/10.1016/J.JNCA.2017.02.009

7. Anexos

Para facilitar la lectura del código diseñado en este trabajo, se adjunta un enlace al repositorio donde se recopilan todos los scripts.

https://github.com/MariaRodriguezGarcia/TFM.git

Anexo I: Otros ataques comunes

A continuación, se detallan más ataques que aparecen con frecuencia en conjuntos de datos en entornos IoT.

Ataques de Fuerza Bruta de Diccionario

Un ataque de fuerza bruta de diccionario es un tipo de ataque de fuerza bruta que utiliza un diccionario predefinido de palabras y combinaciones comunes para intentar descifrar contraseñas. En lugar de probar todas las combinaciones posibles de caracteres, un ataque de diccionario se basa en la probabilidad de que las contraseñas más utilizadas estén presentes en una lista de palabras comunes, frases, combinaciones de números y letras, y otros patrones frecuentes. Estos ataques son efectivos cuando los usuarios utilizan contraseñas débiles o previsibles.

El proceso de un ataque de diccionario implica lo siguiente:

1. **Recopilación de diccionarios**: Los atacantes recogen listas de palabras que contienen contraseñas comunes y frases utilizadas frecuentemente.
2. **Automatización del proceso**: Usando software automatizado, el atacante introduce cada palabra del diccionario como posible contraseña.
3. **Verificación**: Cada intento se verifica contra el sistema de autenticación objetivo hasta encontrar una coincidencia correcta o agotar las opciones del diccionario.

Escaneo de Sistema Operativo (OS *Scan*)

El escaneo de sistema operativo, o *OS scan,* es una técnica utilizada por los atacantes para determinar el sistema operativo que se está ejecutando en un dispositivo de red. Este tipo de escaneo es crucial para los ciberdelincuentes, ya que les permite identificar vulnerabilidades específicas del sistema operativo identificado. Herramientas como Nmap son comúnmente utilizadas para realizar este tipo de escaneo.

El proceso de escaneo de sistema operativo incluye:

1. **Envío de paquetes**: El atacante envía paquetes diseñados específicamente para obtener respuestas que revelen características del sistema operativo.
2. **Análisis de respuestas**: Las respuestas de los dispositivos son analizadas para identificar patrones específicos que son característicos de diferentes sistemas operativos.
3. **Identificación del sistema operativo**: Basándose en los datos recogidos, el atacante puede determinar con alta probabilidad el sistema operativo del dispositivo objetivo.

Escaneo de Puertos (*Port Scan*)

El escaneo de puertos es una técnica utilizada para identificar qué puertos están abiertos en un dispositivo de red. Los puertos abiertos pueden revelar servicios activos y posibles puntos de entrada para ataques. Este tipo de escaneo es a menudo el primer paso en un ataque, proporcionando información vital sobre la estructura y vulnerabilidades de la red objetivo.

Los pasos en un escaneo de puertos incluyen:

1. **Envío de solicitudes**: El atacante envía solicitudes a diferentes puertos en el dispositivo objetivo.
2. **Recepción de respuestas**: Las respuestas indican si un puerto está abierto, cerrado o filtrado.
3. **Análisis de servicios**: Los puertos abiertos son analizados para identificar los servicios que están corriendo y sus posibles vulnerabilidades.

DDoS-PSHACK *Flood*

El ataque DDoS PSHACK *Flood* es un tipo de ataque que se enfoca en utilizar los paquetes TCP con los *flags* PSH (Push) y ACK (Acknowledgment) activados. Este tipo de ataque está diseñado para saturar al servidor objetivo enviando una gran cantidad de estos paquetes, consumiendo los recursos del servidor y provocando una denegación de servicio. A veces se recibe un RST en respuesta al paquete ACK-PSH original porque la pila TCP que recibe el paquete ACK-PSH nunca tuvo una secuencia correspondiente de SYN - SYN+ACK +ACK (handshake TCP). Algunos entornos pueden optar por no enviar un paquete RST de vuelta al origen del paquete ACK-PSH.

Características del PSHACK *Flood*:

1. **Protocolo**: TCP
2. **Perfil de Ancho de Banda**: Alto BPS (bits por segundo), Medio PPS (paquetes por segundo)

Proceso del PSHACK *Flood*:

1. **Envío de Paquetes**: Los bots envían una gran cantidad de paquetes TCP con las *flags* PSH y ACK activadas.
2. **Respuesta del Servidor**: El servidor intenta procesar cada paquete, lo que consume sus recursos.
3. **Agotamiento de Recursos**: La saturación del servidor con estos paquetes lleva a una denegación de servicio.

DDoS-RSTFIN *Flood*

El ataque DDoS RSTFIN *Flood* utiliza paquetes TCP con los *flags* RST (Reset) y FIN (Finish) activados. Este ataque se aprovecha del comportamiento de los servidores que intentan cerrar conexiones TCP. Para cerrar una sesión TCP SYN, se intercambian paquetes RST o FIN entre el cliente y el host. Durante un RST o FIN *flood*, el servidor víctima recibe paquetes RST o FIN falsificados a alta velocidad que no están relacionados con ninguna de las sesiones en la base de datos del servidor. Como resultado, el servidor víctima se ve obligado a asignar una cantidad significativa de recursos del sistema para emparejar los paquetes entrantes con las conexiones actuales, lo que provoca un rendimiento de servidor degradado e inaccesibilidad parcial.

Características del RSTFIN *Flood*:

1. **Protocolo**: TCP
2. **Perfil de Ancho de Banda**: Moderado a Alto BPS, Alto PPS
3. **Tamaño del Paquete**: Pequeño a Medio
4. **Notas**: Utiliza las banderas RST y FIN para cerrar conexiones, lo que puede confundir y sobrecargar al servidor.

Proceso del RSTFIN *Flood*:

1. **Envío de Paquetes**: Los bots envían una gran cantidad de paquetes TCP con las banderas RST y FIN activadas.
2. **Respuesta del Servidor**: El servidor intenta cerrar las conexiones repetidamente, gastando recursos en el proceso.
3. **Agotamiento de Recursos**: La saturación con estos paquetes lleva a la denegación de servicio.

Anexo II: Detalles de las bases de datos seleccionadas

En este anexo se recopilan detalles como la distribución de etiquetas para los ataques y subataques, incluyendo número de flujos por clase.

Número de instancias para el dataset IoTD20 (fichero .csv)

Table 2. Normal and attacked instances in IoTID20 Dataset

Binary label distribution		Subcategory distribution	
Normal	40073	Type	Instances
Anomaly	585710	Normal	40073
		DoS	59391
Category label distribution		Mirai Ack Flooding	55124
Type	Instances	Mirai Brute force	121181
Normal	40073	Mirai HTTP Flooding	55818
DoS	59391	Mirai UDP Flooding	183554
Mirai	415677	MITM	35377
MITM	35377	Scan Host Port	22192
Scan	75265	Scan Port OS	53073

Etiquetado IoTD20

Flow_ID	Flow_Pkts/s	Fwd_Pkts/s	Fwd_Blk_Rate_Avg
Src_IP	Flow_IAT_Mean	Bwd_Pkts/s	Bwd_Byts/b_Avg
Src_Port	Flow_IAT_Std	Pkt_Len_Min	Bwd_Pkts/b_Avg
Dst_IP	Flow_IAT_Max	Pkt_Len_Max	Bwd_Blk_Rate_Avg
Dst_Port	Flow_IAT_Min	Pkt_Len_Mean	Subflow_Fwd_Pkts
Protocol	Fwd_IAT_Tot	Pkt_Len_Std	Subflow_Fwd_Byts
Timestamp	Fwd_IAT_Mean	Pkt_Len_Var	Subflow_Bwd_Pkts
Flow_Duration	Bwd_IAT_Mean	FIN_*Flag*_Cnt	Subflow_Bwd_Byts
Tot_Fwd_Pkts	Fwd_IAT_Max	SYN_*Flag*_Cnt	Init_Fwd_Win_Byts
Tot_Bwd_Pkts	Fwd_IAT_Min	RST_*Flag*_Cnt	Init_Bwd_Win_Byts
TotLen_Fwd_Pkts	Bwd_IAT_Tot	PSH_*Flag*_Cnt	Fwd_Act_Data_Pkts
TotLen_Bwd_Pkts	Bwd_IAT_Mean	ACK_*Flag*_Cnt	Fwd_Seg_Size_Min
Fwd_Pkt_Len_Max	Bwd_IAT_Std	URG_*Flag*_Cnt	Active_Mean
Fwd_Pkt_Len_Min	Bwd_IAT_Max	CWE_*Flag*_Count	Active_Std
Fwd_Pkt_Len_Mean	Bwd_IAT_Min	ECE_*Flag*_Cnt	Active_Max
Fwd_Pkt_Len_Std	Fwd_PSH_*Flags*	Down/Up_Ratio	Active_Min
Bwd_Pkt_Len_Max	Bwd_PSH_*Flags*	Pkt_Size_Avg	Idle_Mean
Bwd_Pkt_Len_Min	Fwd_URG_*Flags*	Fwd_Seg_Size_Avg	Idle_Std
Bwd_Pkt_Len_Mean	Bwd_URG_*Flags*	Bwd_Seg_Size_Avg	Idle_Max
Bwd_Pkt_Len_Std	Fwd_Header_Len	Fwd_Byts/b_Avg	Idle_Min
Flow_Byts/s	Bwd_Header_Len	Fwd_Pkts/b_Avg	

IoT-23: distribución de escenarios

Los escenarios seleccionados para Mirai son: CTU-IoT-*Malware* 34, CTU-IoT-*Malware* 35, CTU-IoT-*Malware* 43, CTU-IoT-*Malware* 44 CTU-IoT-*Malware* 48, CTU-IoT-*Malware* 49, CTU-IoT-*Malware* 52, CTU-IoT-*Malware* 7, y CTU-HoneyPot-4-1, CTU-HoneyPot-5-1 y CTU-HoneyPot-7-1.

#	Name of Dataset	Duration (hrs)	#Packets	#ZeekFlows	Pcap Size	Name
1	CTU-IoT-Malware-Capture-34-1	24	233,000	23,146	121 MB	Mirai
2	CTU-IoT-Malware-Capture-43-1	1	82,000,000	67,321,810	6 GB	Mirai
3	CTU-IoT-Malware-Capture-44-1	2	1,309,000	238	1.7 GB	Mirai
4	CTU-IoT-Malware-Capture-49-1	8	18,000,000	5,410,562	1.3 GB	Mirai
5	CTU-IoT-Malware-Capture-52-1	24	64,000,000	19,781,379	4.6 GB	Mirai
6	CTU-IoT-Malware-Capture-20-1	24	50,000	3,210	3.9 MB	Torii
7	CTU-IoT-Malware-Capture-21-1	24	50,000	3,287	3.9 MB	Torii
8	CTU-IoT-Malware-Capture-42-1	8	24,000	4,427	2.8 MB	Trojan
9	CTU-IoT-Malware-Capture-60-1	24	271,000,000	3,581,029	21 GB	Gagfyt
10	CTU-IoT-Malware-Capture-17-1	24	109,000,000	54,659,864	7.8 GB	Kenjiro
11	CTU-IoT-Malware-Capture-36-1	24	13,000,000	13,645,107	992 MB	Okiru
12	CTU-IoT-Malware-Capture-33-1	24	54,000,000	54,454,592	3.9 GB	Kenjiro
13	CTU-IoT-Malware-Capture-8-1	24	23,000	10,404	2.1 MB	Hakai
14	CTU-IoT-Malware-Capture-35-1	24	46,000,000	10,447,796	3.6G	Mirai
15	CTU-IoT-Malware-Capture-48-1	24	13,000,000	3,394,347	1.2G	Mirai
16	CTU-IoT-Malware-Capture-39-1	7	73,000,000	73,568,982	5.3GB	IRCBot
17	CTU-IoT-Malware-Capture-7-1	24	11,000,000	11,454,723	897 MB	Linux,Mirai
18	CTU-IoT-Malware-Capture-9-1	24	6,437,000	6,378,294	472 MB	Linux.Hajime
19	CTU-IoT-Malware-Capture-3-1	36	496,000	156,104	56 MB	Muhstik
20	CTU-IoT-Malware-Capture-1-1	112	1,686,000	1,008,749	140 MB	Hide and Seek

#	Name of Dataset	Duration(~hrs)	#Packets	#ZeekFlows	Pcap Size	Device
21	CTU-Honeypot-Capture-7-1	1.4	8,276	139	2,094 KB	Somfy Door Lock
22	CTU-Honeypot-Capture-4-1	24	21,000	461	4,594 KB	Philips HUE
23	CTU-Honeypot-Capture-5-1	5.4	398,000	1,383	381 MB	Amazon Echo

Explicación etiquetas IoT-23

1. **Attack**: Indica que ha ocurrido algún tipo de ataque desde el dispositivo infectado hacia otro host, aprovechando servicios vulnerables mediante técnicas como fuerza bruta en autenticación Telnet o inyecciones de comandos en solicitudes GET.
2. **Benign**: Se utiliza para indicar que no se encontraron actividades sospechosas o maliciosas en las conexiones analizadas.

3. **C&C (Command & Control)**: Indica que el dispositivo infectado se ha conectado a un servidor de C&C. Este comportamiento se identifica por conexiones periódicas con el servidor, descargas de binarios desde el mismo o intercambio de órdenes codificadas al estilo IRC.
4. **DDoS (Distributed Denial of Service)**: Se aplica cuando el dispositivo infectado está ejecutando un ataque de denegación de servicio distribuido, detectado por la gran cantidad de flujos dirigidos a una misma dirección IP.
5. **FileDownload**: Indica que se está descargando un archivo hacia el dispositivo infectado, identificado por conexiones con *bytes* de respuesta superiores a 3KB o 5KB, frecuentemente hacia puertos o direcciones IP conocidos como servidores de C&C.
6. **HeartBeat**: Se utiliza cuando los paquetes enviados en una conexión se utilizan para mantener un seguimiento del dispositivo infectado por parte del servidor de C&C. Esto se detecta por conexiones con *bytes* de respuesta muy bajos y conexiones periódicas, usualmente hacia puertos o direcciones IP sospechosas.
7. **Mirai**: Etiqueta que indica características típicas de un *botnet* Mirai en los flujos de conexión. Se aplica cuando los flujos muestran patrones similares a los ataques más comunes asociados con Mirai.
8. **Okiru**: Similar a Mirai, pero identifica características específicas de un *botnet* Okiru, que aunque menos común, presenta comportamientos similares en términos de patrones de conexión.
9. **PartOfAHorizontal*PortScan***: Indica que los flujos están siendo utilizados para realizar un escaneo horizontal de puertos, recopilando información para futuros ataques. Esta etiqueta se basa en patrones donde las conexiones comparten el mismo puerto, una cantidad similar de *bytes* transmitidos y múltiples direcciones IP de destino diferentes.
10. **Torii**: Se utiliza para etiquetar flujos que muestran características típicas de un *botnet* Torii, similar a Mirai y Okiru pero menos común en su detección.

IoT-23: distribución de etiquetas por fichero

En este apartado se especifica el número de instancias por clase y escenario. El número de instancias corresponde al obtenido por los autores mediante la herramienta Zeek.

CTU-IoT-*Malware*-Capture-34-1 (Mirai)

Label	Flows
Benign	1,923
C&C	6,706
DDoS	14,394
PartOfAHorizontal*PortScan*	122

CTU-IoT-*Malware*-Capture-35-1 (Mirai)

Label	Flows
Attack	3
Benign	8,262,389
C&C	81
C&C-FileDownload	12
DDoS	2,185,302

CTU-IoT-*Malware*-Capture-43-1 (Mirai)

Label	Flows
Benign	20,574,934
C&C	3,498
C&C-FileDownload	14
DDoS	65,803
FileDownload	1
Okiru	8,765,885
PartOfAHorizontal*PortScan*	37,911,674

CTU-IoT-*Malware*-Capture-44-1 (Mirai)

Label	Flows
Benign	211
C&C	14
C&C-FileDownload	11
DDoS	1

CTU-IoT-*Malware*-Capture-48-1 (Mirai)

Label	Flows
Attack	2,752
Benign	3,734
C&C-HeartBeat-Attack	834
C&C-HeartBeat-FileDownload	11
C&C-PartOfAHorizontal*PortScan*	888
PartOfAHorizontal*PortScan*	3,386,119

CTU-IoT-*Malware*-Capture-49-1 (Mirai)

Label	Flows
Benign	3,665
C&C	1,922
C&C-FileDownload	1
PartOfAHorizontal*PortScan*	5,404,959

CTU-IoT-*Malware*-Capture-52-1 (Mirai)

Label	Flows
Benign	1,794
C&C	6
C&C-FileDownload	12
PartOfAHorizontal*PortScan*	19,779,564

CTU-IoT-*Malware*-Capture-7-1 (Linux.Mirai)

Label	Flows
Benign	75,955
C&C-HeartBeat	5,778
DDoS	39,584
Okiru	11,333,397

CIC-IoT-2023: distribución de clases de ataque y número de flujos

En la tabla se muestran los ataques (correspondiente al campo label), los subataques (campo detailed-label), el número de instancias dentro del csv y la herramienta empleada para generarlos.

	Attack	Rows	Tool
DDoS	ACK Fragmentation	285,104	hping3 [49]
	UDP Flood	5,412,287	udp-flood [50]
	SlowLoris	23,426	slowloris [51]
	ICMP Flood	7,200,504	hping3 [49]
	RSTFIN Flood	4,045,285	hping3 [49]
	PSHACK Flood	4,094,755	hping3 [49]
	HTTP Flood	28,790	golang-httpflood [52]
	UDP Fragmentation	286,925	udp-flood [50]
	ICMP Fragmentation	452,489	hping3 [49]
	TCP Flood	4,497,667	hping3 [49]
	SYN Flood	4,059,190	hping3 [49]
	SynonymousIP Flood	3,598,138	hping3 [49]
DoS	TCP Flood	2,671,445	hping3 [49]
	HTTP Flood	71,864	golang-httpflood [52]
	SYN Flood	2,028,834	hping3 [49]
	UDP Flood	3,318,595	hping3 [49] and udp-flood [50]
Recon	Ping Sweep	2262	nmap [53] and fping [54]
	OS Scan	98,259	nmap [53]
	Vulnerability Scan	37,382	nmap [53] and vulscan [55]
	Port Scan	82,284	nmap [53]
	Host Discovery	134,378	nmap [53]
Web-Based	Sql Injection	5245	DVWA [56]
	Command Injection	5409	DVWA [56]
	Backdoor Malware	3218	DVWA [56] and Remot3d [57]
	Uploading Attack	1252	DVWA [56]
	XSS	3846	DVWA [56]
	Browser Hijacking	5859	Beef [58]
Brute Force	Dictionary Brute Force	13,064	nmap [53] and hydra [59]
Spoofing	Arp Spoofing	307,593	ettercap [60]
	DNS Spoofing	178,911	ettercap [60]
Mirai	GREIP Flood	751,682	Adapted Mirai Source Code [61]
	Greeth Flood	991,866	Adapted Mirai Source Code [61]
	UDPPlain	890,576	Adapted Mirai Source Code [61]

CIC-IoT-2023: características originales

Las características de este conjunto de datos son las siguientes:

#	Feature	Description
1	ts	Timestamp
2	flow duration	Duration of the packet's flow
3	Header Length	Header Length
4	Protocol Type	IP, UDP, TCP, IGMP, ICMP, Unknown (Integers)
5	Duration	Time-to-Live (ttl)
6	Rate	Rate of packet transmission in a flow
7	Srate	Rate of outbound packets transmission in a flow
8	Drate,	Rate of inbound packets transmission in a flow
9	fin flag number	Fin flag value
10	syn flag number	Syn flag value
11	rst flag number	Rst flag value
12	psh flag numbe	Psh flag value
13	ack flag number	Ack flag value
14	ece flag numbe	Ece flag value
15	cwr flag number	Cwr flag value
16	ack count	Number of packets with ack flag set in the same flow
17	syn count	Number of packets with syn flag set in the same flow
18	fin count	Number of packets with fin flag set in the same flow
19	urg coun	Number of packets with urg flag set in the same flow
20	rst count	Number of packets with rst flag set in the same flow
21	HTTP	Indicates if the application layer protocol is HTTP
22	HTTPS	Indicates if the application layer protocol is HTTPS
23	DNS	Indicates if the application layer protocol is DNS
24	Telnet	Indicates if the application layer protocol is Telnet
25	SMTP	Indicates if the application layer protocol is SMTP
26	SSH	Indicates if the application layer protocol is SSH
27	IRC	Indicates if the application layer protocol is IRC
28	TCP	Indicates if the transport layer protocol is TCP
29	UDP	Indicates if the transport layer protocol is UDP
30	DHCP	Indicates if the application layer protocol is DHCP
31	ARP	Indicates if the link layer protocol is ARP
32	ICMP	Indicates if the network layer protocol is ICMP
33	IPv	Indicates if the network layer protocol is IP
34	LLC	Indicates if the link layer protocol is LLC
35	Tot sum	Summation of packets lengths in flow
36	Min	Minimum packet length in the flow
37	Max	Maximumpacket length in the flow
38	AVG	Average packet length in the flow
39	Std	Standard deviation of packet length in the flow
40	Tot size	Packet's length
41	IAT	The time difference with the previous packet
42	Number	The number of packets in the flow
43	Magnitude	(Average of the lengths of incoming packets in the flow + average of the lengths of outgoing packets in the flow)$^{0.5}$
44	Radius	(Variance of the lengths of incoming packets in the flow + variance of the lengths of outgoing packets in the flow)$^{0.5}$
45	Covariance	Covariance of the lengths of incoming and outgoing packets
46	Variance	Variance of the lengths of incoming packets in the flow/ variance of the lengths of outgoing packets in the flow
47	Weight	Number of incoming packets × Number of outgoing packets

Anexo III: Estudio de Zeek

Archivos generados por Zeek

Fichero	Descripción
conn.log	Detalles de conexión IP, TCP, UDP, ICMP
conn_statistics.log	Detalles de conexión IP, TCP, UDP, ICMP con medidas estadísticas
dhcp.log	Actividad de los leases de DHCP
dns.log	Detalles sobre solicitudes y respuestas DNS
dpd.log	Fallos de detección de protocolo dinámico
files.log	Resultados de análisis de archivos
ftp.log	Detalles de solicitudes y respuestas FTP
http.log	Detalles de solicitudes y respuestas HTTP
irc.log	Detalles de comunicación IRC
Kerberos.log	Autenticación de kerberos
mysql.log	Comandos y respuestas del servidor
radius.log	Intentos de autenticación radius
sip.log	Análisis de SIP
smtp.log	Transacciones SMTP
software.log	Software usado en la red según host
ssh.log	Handshakes de SSH
ssl.log	Handshakes de SSL
syslog.log	Mensajes syslog
tunnel.log	Detalles sobre túneles de encapsulación
weird.log	Actividad inesperada de protocolo o red
X509.log	Información sobre el certificado X.509
dce_rpc.log	Detalles en los mensajes DCE/RPC
ntlm.log	Información sobre NT LAN Manager
rdp.log	Información sobre Remote Desktop Protocol
smb_files.log	Detalles sobre archivos smb
smb_mapping.log	Mapeo de SMB

En el capítulo 5 se muestra el proceso para obtener nuevos atributos, que posteriormente se emplearon en la clasificación de los flujos obtenidos a partir de las capturas. En este anexo se explica el significado de los atributos obtenidos a partir de Zeek.

- ts: tiempo del primer paquete en formato UTC (el timestamp en .pcap CICIDS es UTC-3)
- duration: cuánto ha durado la conexión (campo de tipo intervalo, sus unidades son segundos)
- orig_*bytes*: número de *bytes* de origen a destino

- resp_*bytes*: número de *bytes* de destino a origen
- conn_state (Posibles valores):
  - S0: Intento de conexión visto, sin respuesta.
  - S1: Conexión establecida, no terminada.
  - SF: Establecimiento normal y terminación. Mismo símbolo que para el estado S1. Para distinguirlos, en S1 no hay ningún recuento de *bytes* en el resumen.
  - REJ: Intento de conexión rechazado.
  - S2: Conexión establecida e intento de cierre por parte del origen visto. Sin respuesta del destino.
  - S3: Conexión establecida e intento de cierre por parte del destino visto. Sin respuesta del origen.
  - RSTO: Conexión establecida, el origen abortó la conexión. Envió un RST.
  - RSTR: El destino mandó un RST.
  - RSTOS0: El origen envió un SYN seguido de un RST, nunca se vio un SYN-ACK del destino.
  - RSTRH: El destino envió un SYN ACK seguido de un RST, nunca se vio un SYN del (supuesto) origen.
  - SH: El origen envió un SYN seguido de un FIN, nunca se vio un SYN ACK del destino (por lo tanto, la conexión estaba "medio" abierta).
  - SHR: El destino envió un SYN ACK seguido de un FIN, nunca se vio un SYN del autor.
  - OTH: No se ve SYN, solo tráfico intermedio (un ejemplo de esto es una "conexión parcial" que no se cerró más tarde).
- missed_*bytes*: cantidad de *bytes* perdidos en los gaps (representa los paquetes perdidos en la conexión)
- history: es una cadena de letras que representa la historia del estado de la conexión. Si el evento proviene del origen, la letra está en mayúsculas; si proviene del destino, está en minúsculas.
  - s: SYN sin el bit ACk activo
  - h: SYN+ACK (handshake)
  - a: ACK puro
  - d: paquete con payload (“datos”)
  - f: paquete con bit FIN activo
  - r: paquete con bit RST activo
  - c: paquete con checksum erróneo (se aplica a UDP también)
  - g: gap
  - t: paquete con payload retransmitido
  - w: paquete con anuncio de ventana cero
  - i: paquete inconsistente (por ejemplo, bits FIN+RST)
  - q: paquete multi-*flag* (SYN+FIN o SYN+RST)
  - ^: la dirección de la conexión fue invertida por la heurística de Zeek
- orig_pkts: paquetes de origen a destino
- resp_pkts: paquetes de destino a origen
- orig_ip_*bytes*: número de *bytes* IP enviados por origen.
- orig_*bytes*_no_cero: *bytes* de los paquetes que no tienen Payload nula.
- pkts_orig_cero: paquetes que tienen Payload nula emitidos de origen a destino.
- pkts_orig_no_cero: paquetes que no tienen Payload nula emitidos de origen a destino.
- time: medida de tiempo entre paquetes.

Además se incluyen medidas estadísticas como la media y desviación estándar, valor máximo y mínimo de ciertos atributos.

Anexo IV: *Scripts* de Zeek

A continuación se muestra un ejemplo de *script* que utiliza conn_statistics.zeek para la obtención de atributos a partir de las capturas de tráfico. Éste es llamado dentro del archivo *local.zeek* y genera un nuevo log llamado con_statistics.log, con los atributos propios del archivo conn.log además de otras medidas ya explicadas en el capítulo . Este *script* itera sobre cada archivo en una carpeta que contiene capturas de tráfico y crea otra carpeta que contiene los logs correspondientes a estas.

```
#!/bin/bash
# Directorio de origen
source_dir="/root/bbdd/iotd20/pcaps/dos/"
# Directorio de destino
dest_dir="/root/bbdd/logs-zeek/iotd20-logs/logs-dos/"
# Obtener una lista de archivos pcap en el directorio de origen
files=$(ls "$source_dir"*.pcap)
# Iterar sobre cada archivo
for file_with_extension in $files
do
   # Obtener el nombre del archivo sin la extensión
   filename=$(basename -- "$file_with_extension")
   filename_no_extension="${filename%.*}"
   # Crear el nombre de la carpeta de destino
   dest_folder="$dest_dir$filename_no_extension-logs"
   # Crear la carpeta de destino
   mkdir -p "$dest_folder"
   # Ejecutar Zeek en el archivo actual
   zeek -C -r "$file_with_extension" /usr/local/zeek/share/zeek/site/local.zeek Log::default_logdir="$dest_folder"
done
```

Cada uno de los logs generados tiene una estructura similar a la mostrada a continuación:

*{"**ts**":1558922777.824831,"**startTime**":"2019-05-27 02:06:17","**uid**":"CSsw4d1rySllO7hBS3","id.**orig**_h":"192.168.0.14","**id.orig_p**":54685,"**id.resp_h**":"192.168.0.1","**id.resp_p**":80,"**proto**":"tcp","**duration**":0.08716106414794922,"**orig_bytes**":0,"**resp_bytes**":89574,"**conn_state**":"SHR","**local_orig**":true,"**local_resp**":true,"**missed_bytes**":0,"**history**":"^hadf","**orig_pkts**":0,"**orig_ip_bytes**":0,"**resp_pkts**":68,"**resp_ip_bytes**":93118,"**tunnel_parents**":[],"**orig_bytes_mean**":0.0,"**resp_bytes_mean**":1336.9253731343283,"**resp_bytes_std**":347.3232961417521,"**orig_bytes_mean_nocero**":0.0,"**resp_bytes_mean_nocero**":1399.59375,"**orig_bytes_std_nocero**":0.0,"**resp_bytes_std_nocero**":355.3704631012208,"**orig_bytes_min**":1000,"**resp_bytes_min**":0,"**orig_bytes_max**":0,"**resp_bytes_max**":2318,"**orig_pkts_nocero**":0,"**resp_pkts_nocero**":64,"**orig_pkts_cero**":0,"**resp_pkts_cero**":4,"**time_mean**":0.0012817803551169002,"**time_std**":0.003319510976490021,"**time_min**":0.0,"**time_max**":0.01996302604675293,"**orig_time_mean":**0.0,"**orig_time_min**":10000.0,"**orig_time_max**":0.0,"**resp_time_mean":**0.0012810446999289773,"**resp_time_std**":0.0033617492476943843,"**resp_time_min**":0.0,"**resp_time_max**":0.01996302604675293}*

El archivo .log está formado por una serie de registros en formato JSON que describen conexiones de red. Cada registro representa una conexión individual y contiene múltiples campos con información sobre la conexión.

Anexo V: *Scripts* de Python para conversión a .csv

A continuación se presenta un ejemplo del código empleado para convertir a formato csv los logs que presentan formato .json. En las bases de datos CIC-IoT-2023 e IoT-23 ha sido necesario el uso de la opción *chunks* para leer de forma fragmentada el archivo debido a que el tamaño de los logs superaba al de la memoria RAM disponible.

```
import sys
import json
import pandas as pd
from datetime import datetime

# Rutas de los archivos de entrada y salida
zeek_log_path = r"/root/bbdd/iot-23/CTU-IoT-Malware-Capture-33-1/bro/conn-labeled.log"
csv_output_path = r"/root/bbdd/iot-23/CTU-IoT-Malware-Capture-33-1/bro/output.csv"

# Función para aplicar transformaciones a un chunk de datos
def apply_transformations(chunk):
    # Aplicar transformación a la columna 'ts'
    chunk['ts'] = chunk['ts'].apply(datetime.fromtimestamp)
    # Dividir la columna 'tunnel_parents label detailed-label'
    chunk[['tunnel_parents', 'label', 'detailed-label']] = chunk['tunnel_parents   label   detailed-label'].str.split('\s{3}', expand=True)
    # Eliminar la columna original
    chunk.drop(columns=['tunnel_parents   label   detailed-label'], inplace=True)
    return chunk

# Función para procesar un chunk de datos
def process_chunk(chunk):
    return apply_transformations(chunk)

# Tamaño del chunk
chunk_size = 50000

# Leer el archivo de registro de Zeek en chunks
with open(zeek_log_path, 'r') as file:
    header_line = file.readlines()[6].strip().split('\t')[1:]
chunks = pd.read_csv(zeek_log_path, sep='\t', skiprows=8, names=header_line, engine='python', chunksize=chunk_size)

# Aplicar transformaciones a cada chunk y concatenar los resultados
processed_chunks = [process_chunk(chunk) for chunk in chunks]
df = pd.concat(processed_chunks, ignore_index=True)

# Guardar el DataFrame resultante como archivo CSV
df.to_csv(csv_output_path, index=False)

# Imprimir mensaje de éxito
print("Archivo CSV guardado exitosamente.")
```

Anexo VI: *Scripts* de unión de archivos .csv

Una vez se ha obtenido el log en formato .csv, se deben unir todos aquellos que corresponden a un mismo *dataset*, por lo que obtendremos tres ficheros. En el caso de IoTD20, cuyo tamaño es reducido y es posible realizarlo sin fragmentarlo en *chunks*, se hizo de la siguiente forma:

```
import os
import pandas as pd

def concatenate_csv_files(main_directory,save_directory):
    """
    Concatenate all CSV files in subdirectories of the main directory into a single CSV file.

    Parameters:
    main_directory (str): Path to the main directory containing subdirectories with CSV files.

    The function saves the concatenated CSV file in the main directory, named as the main directory name + '_all.csv'.
    """

    # Get the main directory name for the output file
    main_directory_name = os.path.basename(os.path.normpath(main_directory))
    # Initialize an empty list to hold DataFrames
    data_frames = []
    header_saved = False
    column_order = []

    total_length = 0  # Initialize total length counter
    # Walk through each subfolder in the main directory
    for subdir, _, files in os.walk(main_directory):
        for file in files:
            # Check if the file is a CSV file
            if file.endswith('.csv'):
                file_path = os.path.join(subdir, file)
                # Read the CSV file and append the DataFrame to the list
                if not header_saved:
                    # Read the first CSV file with headers
                    df = pd.read_csv(file_path)
                    header_saved = True
                    column_order = df.columns.tolist()  # Save the column order
                    print(column_order)
                else:
                    # Read subsequent CSV files
                    df = pd.read_csv(file_path)
                    # Reorder the columns of the DataFrame to match the column order of the first DataFrame
                    df = df.reindex(column_order, axis=1)
                data_frames.append(df)
                total_length += len(df)  # Add length of current DataFrame to total length

    # Concatenate all DataFrames in the list into a single DataFrame
    concatenated_df = pd.concat(data_frames, ignore_index=True)

    # Save the concatenated DataFrame to a new CSV file in the main directory
    output_file = os.path.join(save_directory, f'{main_directory_name}_all.csv')
    concatenated_df.to_csv(output_file, index=False)

    print(f'Total length of concatenated CSV: {total_length}')
    print(f'All CSV files have been concatenated and saved to {output_file}')
```

Sin embargo, para los *datasets* IoT-23 y CIC-IoT-2023 requieren su lectura y escritura fragmentadas:

```
import os
import pandas as pd

def concatenate_csv_files(main_directory, save_directory, chunk_size=50000):
    """
    Concatenate all CSV files in subdirectories of the main directory into a single CSV file.
    Parameters:
    main_directory (str): Path to the main directory containing subdirectories with CSV files.
    save_directory (str): Path to the directory where the concatenated CSV file will be saved.
    chunk_size (int): Number of rows per chunk to read from each CSV file.
    """
    # Get the main directory name for the output file
    main_directory_name = os.path.basename(os.path.normpath(main_directory))
    output_file = os.path.join(save_directory, f'{main_directory_name}_all.csv')
    # Initialize a flag to indicate whether to write header
    header_written = False
    column_order = []
    total_length = 0  # Initialize total length counter
    # Walk through each subfolder in the main directory
    for subdir, _, files in os.walk(main_directory):
        for file in files:
            # Check if the file is a CSV file
            if file.endswith('.csv'):
                file_path = os.path.join(subdir, file)

                # Process the CSV file in chunks
                for chunk in pd.read_csv(file_path, chunksize=chunk_size):
                    if not header_written:
                        # Write the first chunk with headers and save column order
                        chunk.to_csv(output_file, mode='w', header=True, index=False)
                        header_written = True
                        column_order = chunk.columns.tolist()
                    else:
                        # Ensure chunk has same column order and write without headers
                        chunk = chunk.reindex(columns=column_order)
                        chunk.to_csv(output_file, mode='a', header=False, index=False)
                    # Add length of current chunk to total length
                    total_length += len(chunk)

    print(f'Total length of concatenated CSV: {total_length}')
    print(f'All CSV files have been concatenated and saved to {output_file}')
```

Anexo VII: *Scripts* de etiquetado

A continuación, se muestra uno de los *scripts* empleados para el etiquetado de los flujos en el capítulo 5. Para ello se ha empleado la librería *pandas* de Python. Se genera la etiqueta original *detailed-label*, la etiqueta común *label,* y la etiqueta binaria *binary-label.* Después, en función del nombre del archivo, que sirve como indicativo del tipo de ataque contenido, comprueba los flujos maliciosos mediante campos como puerto origen y destino (id.orig_p e id.resp_p), direcciones origen y destino (id.orig_h e id.rsep_h), protocolo (*proto*), estado de conexión (conn_state), y si se ha empleado algún tipo de *flag*, empleando el campo *history* para este último caso.

```
import os
import pandas as pd
def process_conn_log(folder_path):
    conn_log_path = os.path.join(folder_path, "conn_stadistics.log")
    # Check if conn_stadistics.log file exists
    if os.path.exists(conn_log_path):
        # Extract file name from folder path
        folder_name = os.path.basename(folder_path)

        # Read conn_stadistics.log into a DataFrame
        df = pd.read_json(conn_log_path, lines=True)

        # Create a column to hold binary-label with default value 'benign'
        df['binary-label'] = '0'
        df['label'] = 'benign'
        df['detailed-label'] = 'benign'

        if "mirai-udpflooding" in folder_name:
            conditions = df["id.orig_h"] == "210.89.164.90"
            df.loc[conditions, ['binary-label', 'label', 'detailed-label']] = ["1", "Mirai", "Mirai-UDPFlood"]

        elif "mirai-ackflooding" in folder_name:
            conditions = df["id.orig_h"] == "210.89.164.90"
            df.loc[conditions, ['binary-label', 'label', 'detailed-label']] = ["1", "Mirai", "Mirai-ACKFlood"]

        elif "mirai-httpflooding" in folder_name:
            conditions = df["id.orig_h"] == "210.89.164.90"
            df.loc[conditions, ['binary-label', 'label', 'detailed-label']] = ["1", "Mirai", "Mirai-HTTPFlood"]

        elif "mirai-hostbruteforce" in folder_name:
            if ("mirai-hostbruteforce-1" in folder_name or "mirai-hostbruteforce-3" in folder_name or "mirai-hostbruteforce-5" in
folder_name):
                conditions = (df["id.orig_h"] == "192.168.0.13") & (df["proto"] == "tcp") & (df["id.resp_p"] == 23)
                df.loc[conditions, 'label'] = 'Mirai'
                df.loc[conditions, 'binary-label'] = 1  # Assuming you want binary label as 1
                df.loc[conditions, 'detailed-label'] = 'Mirai-TelnetBruteforce'

            elif ("mirai-hostbruteforce-2" in folder_name or "mirai-hostbruteforce-4" in folder_name):
                print("mirando hbf2 o hbf4")
                conditions = (df["id.orig_h"] == "192.168.0.24") & (df["proto"] == "tcp") & (df["id.resp_p"] == 23)
                df.loc[conditions, 'label'] = 'Mirai'
                df.loc[conditions, 'binary-label'] = 1  # Assuming you want binary label as 1
                df.loc[conditions, 'detailed-label'] = 'Mirai-TelnetBruteforce'

        else:
            print(f"No matching condition found for folder {folder_path}")
            return
        # Save the DataFrame as conn_stadistics_labeled.csv
        df.to_csv(os.path.join(folder_path, "conn_stadistics_labeled.csv"), index=False)
        print(f"Saved conn_stadistics_labeled.csv in {folder_path}")

    else:
        print(f"conn_stadistics.log not found in {folder_path}")
```

Etiquetado DoS

```
import ipaddress
def is_ipv4(address):
    try:
        ipaddress.IPv4Address(address)
        return True
    except ipaddress.AddressValueError:
        return False
def process_conn_log(folder_path):
    conn_log_path = os.path.join(folder_path, "conn_stadistics.log")
    # Check if conn_stadistics.log file exists
    if os.path.exists(conn_log_path):
        # Extract file name from folder path
        folder_name = os.path.basename(folder_path)
        # Read conn_stadistics.log into a DataFrame
        df = pd.read_json(conn_log_path, lines=True)
        # Create a column to hold binary-label with default value 'benign'
        df['binary-label'] = '0'
        df['label'] = 'benign'
        df['detailed-label'] = 'benign'
        if "dos-synflooding-1-dec" in folder_name or "dos-synflooding-2-dec" in folder_name :
            # Apply is_ipv4 function to id.orig_h column to check if each value is an IPv4 address
            ipv4_mask = df["id.orig_h"].apply(is_ipv4)
            network_range = ipaddress.ip_network("222.0.0.0/8")
            # Apply ipaddress.ip_address() function only to IPv4 addresses in the id.orig_h column
            ipv4_addresses = df.loc[ipv4_mask, "id.orig_h"].apply(ipaddress.ip_address)
            # Check if the IPv4 addresses are in the network range
            ip_in_net = ipv4_addresses.apply(lambda x: x in network_range)
            conditions = (ip_in_net) & (df["history"].str.lower().str.contains("s")) & (df["id.resp_h"] == "192.168.0.13") & (df["id.resp_p"]
== 554) & (df["proto"] == "tcp")
            df.loc[conditions, ['binary-label', 'label', 'detailed-label']] = ["1", "DoS", "DoS-SYNFlood"]

        elif "dos-synflooding-3-dec" in folder_name:
            ipv4_mask = df["id.orig_h"].apply(is_ipv4)
            network_range = ipaddress.ip_network("111.0.0.0/8")
            # Apply ipaddress.ip_address() function only to IPv4 addresses in the id.orig_h column
            ipv4_addresses = df.loc[ipv4_mask, "id.orig_h"].apply(ipaddress.ip_address)
            # Check if the IPv4 addresses are in the network range
            ip_in_net = ipv4_addresses.apply(lambda x: x in network_range)

            conditions = (ip_in_net) & (df["history"].str.lower().str.contains("s")) & (df["id.resp_h"] == "192.168.0.13") & (df["proto"] ==
"tcp") & (df["id.resp_p"] == 554)
            df.loc[conditions, ['binary-label', 'label', 'detailed-label']] = ["1", "DoS", "DoS-SYNFlood"]
        elif "dos-synflooding-4-dec" in folder_name or "dos-synflooding-5-dec" in folder_name or "dos-synflooding-6-dec" in
folder_name :
            ipv4_mask = df["id.orig_h"].apply(is_ipv4)
            network_range = ipaddress.ip_network("111.0.0.0/8")
            # Apply ipaddress.ip_address() function only to IPv4 addresses in the id.orig_h column
            ipv4_addresses = df.loc[ipv4_mask, "id.orig_h"].apply(ipaddress.ip_address)
            # Check if the IPv4 addresses are in the network range
            ip_in_net = ipv4_addresses.apply(lambda x: x in network_range)
            conditions = (ip_in_net) & (df["history"].str.lower().str.contains("s")) & (df["id.resp_h"] == "192.168.0.24") & (df["proto"] ==
"tcp")& (df["id.resp_p"] == 19604)
            df.loc[conditions, ['binary-label', 'label', 'detailed-label']] = ["1", "DoS", "DoS-SYNFlood"]
        else:
            print(f"No matching condition found for folder {folder_path}")
            return
        # Save the DataFrame as conn_stadistics_labeled.csv
        df.to_csv(os.path.join(folder_path, "conn_stadistics_labeled.csv"), index=False)
        print(f"Saved conn_stadistics_labeled.csv in {folder_path}")
    else:
        print(f"conn_stadistics.log not found in {folder_path}")
for folder in os.listdir(main_directory):
    folder_path = os.path.join(main_directory, folder)
    if os.path.isdir(folder_path):
        process_conn_log(folder_path)
```

Etiquetado *Scan*

```
main_directory = "/root/bbdd/logs-zeek/iotd20-logs/logs-scan/"
def process_conn_log(folder_path):
    conn_log_path = os.path.join(folder_path, "conn_stadistics.log")

    # Check if conn_stadistics.log file exists
    if os.path.exists(conn_log_path):
        # Extract file name from folder path
        folder_name = os.path.basename(folder_path)

        # Read conn_stadistics.log into a DataFrame
        df = pd.read_json(conn_log_path, lines=True)

        # Create a column to hold binary-label with default value 'benign'
        df['binary-label'] = '0'
        df['label'] = 'benign'
        df['detailed-label'] = 'benign'

        if "scan-hostport-1-dec" in folder_name or "scan-hostport-2-dec" in folder_name or "scan-hostport-3-dec" in folder_name :
            conditions = (df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.13") &
((df["history"].str.lower().str.contains("s")) | (df["history"].str.lower().str.contains("r")))
            df.loc[conditions, ['binary-label', 'label', 'detailed-label']] = ["1", "Scan", "Scan-PortScan"]

        elif "scan-hostport-4-dec" in folder_name or "scan-hostport-5-dec" in folder_name or "scan-hostport-6-dec" in folder_name:
            conditions = (df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.24") &
((df["history"].str.lower().str.contains("s")) | (df["history"].str.lower().str.contains("r")))
            df.loc[conditions, ['binary-label', 'label', 'detailed-label']] = ["1", "Scan", "Scan-PortScan"]



        elif "scan-portos-1-dec" in folder_name or "scan-portos-2-dec" in folder_name or "scan-portos-3-dec" in folder_name:
            conditions = (df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.13") &
((df["history"].str.lower().str.contains("s")) | (df["history"].str.lower().str.contains("r")))
            df.loc[conditions, ['binary-label', 'label', 'detailed-label']] = ["1", "Scan", "Scan-PortScan"]

            conditions2 = (df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.13") & (df["proto"] != "icmp") &
~((df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.13") & (((df["proto"] == "tcp") &
(df["history"].str.lower().str.contains("s"))) | (df["history"].str.lower().str.contains("r"))))
            df.loc[conditions2, ['binary-label', 'label', 'detailed-label']] = ["1", "Scan", "Scan-OSDetection"]

        elif "scan-portos-4-dec" in folder_name or "scan-portos-5-dec" in folder_name or "scan-portos-6-dec" in folder_name:
            conditions = (df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.24") &
((df["history"].str.lower().str.contains("s")) | (df["history"].str.lower().str.contains("r")))
            df.loc[conditions, ['binary-label', 'label', 'detailed-label']] = ["1", "Scan", "Scan-PortScan"]

            conditions2 = (df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.24") & (df["proto"] != "icmp") &
~((df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.24") & (((df["proto"] == "tcp") &
(df["history"].str.lower().str.contains("s"))) | (df["history"].str.lower().str.contains("r"))))
            df.loc[conditions2, ['binary-label', 'label', 'detailed-label']] = ["1", "Scan", "Scan-OSDetection"]

        else:
            print(f"No matching condition found for folder {folder_path}")
            return

        # Save the DataFrame as conn_stadistics_labeled.csv
        df.to_csv(os.path.join(folder_path, "conn_stadistics_labeled.csv"), index=False)
        print(f"Saved conn_stadistics_labeled.csv in {folder_path}")

    else:
        print(f"conn_stadistics.log not found in {folder_path}")


for folder in os.listdir(main_directory):
    folder_path = os.path.join(main_directory, folder)
    if os.path.isdir(folder_path):
        process_conn_log(folder_path)
```

Etiquetado MITM

```
main_directory = "/root/bbdd/logs-zeek/iotd20-logs/logs-mitm/"


def process_conn_log(folder_path):
    conn_log_path = os.path.join(folder_path, "conn_stadistics.log")

    # Check if conn_stadistics.log file exists
    if os.path.exists(conn_log_path):
        # Extract file name from folder path
        folder_name = os.path.basename(folder_path)

        # Read conn_stadistics.log into a DataFrame
        df = pd.read_json(conn_log_path, lines=True)

        # Create a column to hold binary-label with default value 'benign'
        df['binary-label'] = '0'
        df['label'] = 'benign'
        df['detailed-label'] = 'benign'

        if "scan-hostport-1-dec" in folder_name or "scan-hostport-2-dec" in folder_name or "scan-hostport-3-dec" in folder_name :
            conditions = (df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.13") &
((df["history"].str.lower().str.contains("s")) | (df["history"].str.lower().str.contains("r")))
            df.loc[conditions, ['binary-label', 'label', 'detailed-label']] = ["1", "Scan", "Scan-HostDiscovery"]

        elif "scan-hostport-4-dec" in folder_name or "scan-hostport-5-dec" in folder_name or "scan-hostport-6-dec" in folder_name:
            conditions = (df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.24") &
((df["history"].str.lower().str.contains("s")) | (df["history"].str.lower().str.contains("r")))
            df.loc[conditions, ['binary-label', 'label', 'detailed-label']] = ["1", "Scan", "Scan-HostDiscovery"]
        elif "scan-portos-1-dec" in folder_name or "scan-portos-2-dec" in folder_name or "scan-portos-3-dec" in folder_name:
            conditions = (df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.13") &
((df["history"].str.lower().str.contains("s")) | (df["history"].str.lower().str.contains("r")))
            df.loc[conditions, ['binary-label', 'label', 'detailed-label']] = ["1", "Scan", "Scan-PortScan"]

            conditions2 = (df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.13") & (df["proto"] != "icmp") &
~((df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.13") & (((df["proto"] == "tcp") &
(df["history"].str.lower().str.contains("s"))) | (df["history"].str.lower().str.contains("r"))))
            df.loc[conditions2, ['binary-label', 'label', 'detailed-label']] = ["1", "Scan", "Scan-OSDetection"]

        elif "scan-portos-4-dec" in folder_name or "scan-portos-5-dec" in folder_name or "scan-portos-6-dec" in folder_name:
            conditions = (df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.24") &
((df["history"].str.lower().str.contains("s")) | (df["history"].str.lower().str.contains("r")))
            df.loc[conditions, ['binary-label', 'label', 'detailed-label']] = ["1", "Scan", "Scan-PortScan"]

            conditions2 = (df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.24") & (df["proto"] != "icmp") &
~((df["id.orig_h"] == "192.168.0.15") & (df["id.resp_h"] == "192.168.0.24") & (((df["proto"] == "tcp") &
(df["history"].str.lower().str.contains("s"))) | (df["history"].str.lower().str.contains("r"))))
            df.loc[conditions2, ['binary-label', 'label', 'detailed-label']] = ["1", "Scan", "Scan-OSDetection"]

        else:
            print(f"No matching condition found for folder {folder_path}")
            return
        # Save the DataFrame as conn_stadistics_labeled.csv
        df.to_csv(os.path.join(folder_path, "conn_stadistics_labeled.csv"), index=False)
        print(f"Saved conn_stadistics_labeled.csv in {folder_path}")
    else:
        print(f"conn_stadistics.log not found in {folder_path}")
```

 # Para benign

```
main_directory = "/root/bbdd/logs-zeek/iotd20-logs/logs-benign"
def process_conn_log(folder_path):
    conn_log_path = os.path.join(folder_path, "conn_stadistics.log")

    # Check if conn_stadistics.log file exists
    if os.path.exists(conn_log_path):
        # Extract file name from folder path
        folder_name = os.path.basename(folder_path)

        # Read conn_stadistics.log into a DataFrame
        with open(conn_log_path, 'r') as file:
            header_line = file.readlines()[6].strip().split('\t')[1:]
        df = pd.read_csv(conn_log_path, sep='\t', skiprows=8, names=header_line, skipfooter=1, engine='python')

        # Create a column to hold binary-label with default value 'benign'
        df['binary-label'] = '0'
        df['label'] = 'benign'
        df['detailed-label'] = 'benign'
        # Save the DataFrame as conn_stadistics_labeled.csv
        df.to_csv(os.path.join(folder_path, "conn_stadistics_labeled.csv"), index=False)
        print(f"Saved conn_stadistics_labeled.csv in {folder_path}")

    else:
        print(f"conn_stadistics.log not found in {folder_path}")
for folder in os.listdir(main_directory):
    folder_path = os.path.join(main_directory, folder)
    if os.path.isdir(folder_path):
        process_conn_log(folder_path)
```

Para el etiquetado del cojunto de datos IoT-23 se realiza de forma separada por escenarios, ya que cada uno de estos necesita un etiquetado diferente. Para este *dataset*, fue necesario emplear otra técnica de etiquetado, ya que las condiciones empleadas por los autores del *dataset* estaban formuladas de forma que pudisen superponerse etiquetas en ciertos casos.

```
import json

input_file = "/root/bbdd/logs-zeek/iot-23-logs/CTU-IoT-Malware-Capture-34-1/conn_stadistics.log"
output_file = "/root/bbdd/logs-zeek/iot-23-logs/labeled-csv/json-labeled-CTU-IoT-Malware-Capture-34-1.csv"
label_1 = "C&C"
label_2 = "PartOfAHorizontalPortscan"
label_3 = "DDoS"

# Initialize label checks
label_checked = [False] * 3

all_keys = ["ts", "startTime", "uid", "id.orig_h", "id.orig_p", "id.resp_h", "id.resp_p", "proto", "service", "duration", "orig_bytes",
        "resp_bytes", "conn_state", "local_orig", "local_resp", "missed_bytes", "history", "orig_pkts", "orig_ip_bytes", "resp_pkts",
        "resp_ip_bytes", "tunnel_parents", "orig_bytes_mean", "resp_bytes_mean", "orig_bytes_std", "resp_bytes_std",
"orig_bytes_mean_nocero",  "resp_bytes_mean_nocero", "orig_bytes_std_nocero", "resp_bytes_std_nocero", "orig_bytes_min",
"resp_bytes_min", "orig_bytes_max",  "resp_bytes_max", "orig_pkts_nocero", "resp_pkts_nocero", "orig_pkts_cero",
"resp_pkts_cero", "time_mean", "time_std", "time_min",  "time_max", "orig_time_mean", "orig_time_std", "orig_time_min",
"orig_time_max", "resp_time_mean", "resp_time_std", "resp_time_min", "resp_time_max"]

with open(input_file, "r") as f_in, open(output_file, "w") as f_out:
    f_out.write("ts,startTime,uid,id.orig_h,id.orig_p,id.resp_h,id.resp_p,proto,service,duration,orig_bytes,resp_bytes,conn_state,local
_orig,local_resp,missed_bytes,history,orig_pkts,orig_ip_bytes,resp_pkts,resp_ip_bytes,tunnel_parents,orig_bytes_mean,resp_bytes
_mean,orig_bytes_std,resp_bytes_std,orig_bytes_mean_nocero,resp_bytes_mean_nocero,orig_bytes_std_nocero,resp_bytes_std_n
ocero,orig_bytes_min,resp_bytes_min,orig_bytes_max,resp_bytes_max,orig_pkts_nocero,resp_pkts_nocero,orig_pkts_cero,resp_pk
ts_cero,time_mean,time_std,time_min,time_max,orig_time_mean,orig_time_std,orig_time_min,orig_time_max,resp_time_mean,r
esp_time_std,resp_time_min,resp_time_max,label,binary-label\n")
    for line in f_in:
        data = json.loads(line)
```

```
    for key in all_keys:
        if key not in data:
            data[key] = ""
    binary_label = 0
    labels = []  # Start empty
    if data["id.resp_p"] == "6667" and not label_checked[0]:
        labels.append(label_1)
        binary_label = 1
        label_checked[0] = True
    if data["id.resp_p"] == "63798" and not label_checked[1]:
        labels.append(label_2)
        binary_label = 1
        label_checked[1] = True
    if data["id.resp_p"] == "256" and not label_checked[1]:
        labels.append(label_2)
        binary_label = 1
        label_checked[1] = True
    if data["id.resp_h"] == "123.59.209.185" and not label_checked[2]:
        labels.append(label_3)
        binary_label = 1
        label_checked[2] = True
    if data["id.resp_h"] == "71.61.66.148" and not label_checked[2]:
        labels.append(label_3)
        binary_label = 1
        label_checked[2] = True
    if data["id.resp_h"] == "74.91.117.248" and not label_checked[2]:
        labels.append(label_3)
        binary_label = 1
        label_checked[2] = True
    if data["id.resp_p"] == "5376" and not label_checked[2]:
        labels.append(label_3)
        binary_label = 1
        label_checked[2] = True


    # Reset label checks if all labels are checked
    # If no labels are added, assign "benign"
    if not labels:
        labels.append("benign")

    values = [str(data[key]) for key in all_keys]
    labels_joined = '-'.join(labels)
    csv_line = ','.join(values) + f',{labels_joined},{binary_label}\n'
    f_out.write(csv_line)

    label_checked = [False] * 3
```

 # Escenario 35

```
import json
input_file = "/root/bbdd/logs-zeek/iot-23-logs/CTU-IoT-Malware-Capture-35-1/conn_stadistics.log"
output_file = "/root/bbdd/logs-zeek/iot-23-logs/labeled-csv/json-labeled-CTU-IoT-Malware-Capture-35-1.csv"

label_1 = "C&C"
label_2 = "FileDownload"
label_3 = "Attack"
label_4 = "DDoS"

# Initialize label checks
label_checked = [False] * 4

with open(input_file, "r") as f_in, open(output_file, "w") as
f_out:    f_out.write("ts,startTime,uid,id.orig_h,id.orig_p,id.resp_h,id.resp_p,proto,service,duration,orig_bytes,resp_bytes,conn_stat
e,local_orig,local_resp,missed_bytes,history,orig_pkts,orig_ip_bytes,resp_pkts,resp_ip_bytes,tunnel_parents,orig_bytes_mean,resp
_bytes_mean,orig_bytes_std,resp_bytes_std,orig_bytes_mean_nocero,resp_bytes_mean_nocero,orig_bytes_std_nocero,resp_bytes
_std_nocero,orig_bytes_min,resp_bytes_min,orig_bytes_max,resp_bytes_max,orig_pkts_nocero,resp_pkts_nocero,orig_pkts_cero,r
```

```
esp_pkts_cero,time_mean,time_std,time_min,time_max,orig_time_mean,orig_time_std,orig_time_min,orig_time_max,resp_time_
mean,resp_time_std,resp_time_min,resp_time_max,label,binary-label\n")

  for line in f_in:
    data = json.loads(line)
    for key in all_keys:
      if key not in data:
        data[key] = ""
    binary_label = 0
    labels = []  # Start empty
    if data["id.resp_h"] == "104.248.160.24" and not label_checked[0]:
      labels.append(label_1)
      binary_label = 1
      label_checked[0] = True
    if data["id.resp_h"] == "104.248.160.24" and data["resp_ip_bytes"] > 30000 and not label_checked[1]:
      labels.append(label_2)
      binary_label = 1
      label_checked[1] = True
    if data["id.resp_h"] == "110.183.76.177" and not label_checked[2]:
      labels.append(label_3)
      binary_label = 1
      label_checked[2] = True
    if data["id.resp_h"] == "112.27.30.87" and not label_checked[2]:
      labels.append(label_3)
      binary_label = 1
      label_checked[2] = True
    if data["id.resp_h"] == "85.217.225.181" and not label_checked[2]:
      labels.append(label_3)
      binary_label = 1
      label_checked[2] = True
    if data["id.resp_p"] == "992" and not label_checked[3]:
      labels.append(label_4)
      binary_label = 1
      label_checked[3] = True
    if data["id.resp_h"] == "209.97.190.136" and not label_checked[3]:
      labels.append(label_4)
      binary_label = 1
      label_checked[3] = True
    if data["id.resp_h"] == "173.113.172.138" and not label_checked[3]:
      labels.append(label_4)
      binary_label = 1
      label_checked[3] = True
    if data["id.resp_h"] == "216.18.168.16" and not label_checked[3]:
      labels.append(label_4)
      binary_label = 1
      label_checked[3] = True
    if data["id.resp_h"] == "24.165.115.195" and not label_checked[3]:
      labels.append(label_4)
      binary_label = 1
      label_checked[3] = True
    if data["id.resp_h"] == "54.39.87.104" and not label_checked[3]:
      labels.append(label_4)
      binary_label = 1
      label_checked[3] = True

    # Reset label checks if all labels are checked
    # If no labels are added, assign "benign"
    if not labels:
      labels.append("benign")

    values = [str(data[key]) for key in all_keys]
    labels_joined = '-'.join(labels)
    csv_line = ','.join(values) + f',{labels_joined},{binary_label}\n'
    f_out.write(csv_line)
    label_checked = [False] * 4
```

 # Escenario 43

```
import json
input_file = "/root/bbdd/logs-zeek/iot-23-logs/CTU-IoT-Malware-Capture-43-1/conn_stadistics.log"
output_file = "/root/bbdd/logs-zeek/iot-23-logs/labeled-csv/json-labeled-CTU-IoT-Malware-Capture-43-1.csv"
label_1 = "C&C"
label_2 = "DDoS"
label_3 = "Okiru"
label_4 = "PartOfAHorizontalPortScan"
label_5 = "FileDonwload"

# Initialize label checks
label_checked = [False] * 5
with open(input_file, "r") as f_in, open(output_file, "w") as
f_out:  f_out.write("ts,startTime,uid,id.orig_h,id.orig_p,id.resp_h,id.resp_p,proto,service,duration,orig_bytes,resp_bytes,conn_state
,local_orig,local_resp,missed_bytes,history,orig_pkts,orig_ip_bytes,resp_pkts,resp_ip_bytes,tunnel_parents,orig_bytes_mean,resp_
bytes_mean,orig_bytes_std,resp_bytes_std,orig_bytes_mean_nocero,resp_bytes_mean_nocero,orig_bytes_std_nocero,resp_bytes_
std_nocero,orig_bytes_min,resp_bytes_min,orig_bytes_max,resp_bytes_max,orig_pkts_nocero,resp_pkts_nocero,orig_pkts_cero,re
sp_pkts_cero,time_mean,time_std,time_min,time_max,orig_time_mean,orig_time_std,orig_time_min,orig_time_max,resp_time_m
ean,resp_time_std,resp_time_min,resp_time_max,label,binary-label\n")

  for line in f_in:
    data = json.loads(line)
    for key in all_keys:
      if key not in data:
        data[key] = ""
    binary_label = 0
    labels = []  # Start empty
    if data["id.resp_p"] == 45 and not label_checked[0]:
      labels.append(label_1)
      binary_label = 1
      label_checked[0] = True

    if data["id.resp_h"] == "142.11.219.83" and not label_checked[0]:
      labels.append(label_1)
      binary_label = 1
      label_checked[0] = True

    if data["id.resp_p"] == 27015 and not label_checked[1]:
      labels.append(label_2)
      binary_label = 1
      label_checked[1] = True

    if data["id.resp_p"] == 37215 and not label_checked[2]:
      labels.append(label_3)
      binary_label = 1
      label_checked[2] = True

    if data["id.resp_p"] == 52869 and data["conn_state"] == "S0" and not label_checked[3]:
      labels.append(label_4)
      binary_label = 1
      label_checked[3] = True

    if data["resp_ip_bytes"] > 50000 and not label_checked[4]:
      labels.append(label_5)
      binary_label = 1
      label_checked[4] = True

    # If no labels are added, assign "benign"
    if not labels:
      labels.append("benign")

    values = [str(data[key]) for key in all_keys]
    labels_joined = '-'.join(labels)
    csv_line = ','.join(values) + f',{labels_joined},{binary_label}\n'
    f_out.write(csv_line)
    label_checked = [False] * 5
```

 # Escenario 44

```
import json
input_file = "/root/bbdd/logs-zeek/iot-23-logs/CTU-IoT-Malware-Capture-44-1/conn_stadistics.log"
output_file = "/root/bbdd/logs-zeek/iot-23-logs/labeled-csv/json-labeled-CTU-IoT-Malware-Capture-44-1.csv"

label_1 = "C&C"
label_2 = "DDoS"
label_3 = "FileDonwload"

# Initialize label checks
label_checked = [False] * 3
with open(input_file, "r") as f_in, open(output_file, "w") as f_out:
    f_out.write("ts,startTime,uid,id.orig_h,id.orig_p,id.resp_h,id.resp_p,proto,service,duration,orig_bytes,resp_bytes,conn_state,local_orig,local_resp,missed_bytes,history,orig_pkts,orig_ip_bytes,resp_pkts,resp_ip_bytes,tunnel_parents,orig_bytes_mean,resp_bytes_mean,orig_bytes_std,resp_bytes_std,orig_bytes_mean_nocero,resp_bytes_mean_nocero,orig_bytes_std_nocero,resp_bytes_std_nocero,orig_bytes_min,resp_bytes_min,orig_bytes_max,resp_bytes_max,orig_pkts_nocero,resp_pkts_nocero,orig_pkts_cero,resp_pkts_cero,time_mean,time_std,time_min,time_max,orig_time_mean,orig_time_std,orig_time_min,orig_time_max,resp_time_mean,resp_time_std,resp_time_min,resp_time_max,label,binary-label\n")

    for line in f_in:
        data = json.loads(line)

        for key in all_keys:
            if key not in data:
                data[key] = ""

        binary_label = 0
        labels = []  # Start empty
        if data["id.resp_h"] == "46.101.251.172" and (data['proto'] == "tcp") and not label_checked[0]:
            labels.append(label_1)
            binary_label = 1
            label_checked[0] = True

        if data["id.resp_p"] == 80 and (data['proto'] == "udp") and not label_checked[1]:
            labels.append(label_2)
            binary_label = 1
            label_checked[1] = True

        if data["id.resp_h"] == "86.136.151.56"  and not label_checked[1]:
            labels.append(label_2)
            binary_label = 1
            label_checked[1] = True

        if data['resp_ip_bytes'] > 50000 and not label_checked[2]:
            labels.append(label_3)
            binary_label = 1
            label_checked[2] = True



        # Reset label checks if all labels are checked
        # If no labels are added, assign "benign"
        if not labels:
            labels.append("benign")

        values = [str(data[key]) for key in all_keys]
        labels_joined = '-'.join(labels)
        csv_line = ','.join(values) + f',{labels_joined},{binary_label}\n'
        f_out.write(csv_line)

        label_checked = [False] * 3
```

 # Escenario 48

```
import json
input_file = "/root/bbdd/logs-zeek/iot-23-logs/CTU-IoT-Malware-Capture-48-1/conn_stadistics.log"
output_file = "/root/bbdd/logs-zeek/iot-23-logs/labeled-csv/json-labeled-CTU-IoT-Malware-Capture-48-1.csv"
label_1 = "C&C"
label_2 = "HeartBeat"
label_3 = "FileDownload"
label_4 = "PartOfAHorizontalPortScan"
label_5 = "Attack"

# Initialize label checks
label_checked = [False] * 5
with open(input_file, "r") as f_in, open(output_file, "w") as f_out:
    f_out.write("ts,startTime,uid,id.orig_h,id.orig_p,id.resp_h,id.resp_p,proto,service,duration,orig_bytes,resp_bytes,conn_state,local
_orig,local_resp,missed_bytes,history,orig_pkts,orig_ip_bytes,resp_pkts,resp_ip_bytes,tunnel_parents,orig_bytes_mean,resp_bytes
_mean,orig_bytes_std,resp_bytes_std,orig_bytes_mean_nocero,resp_bytes_mean_nocero,orig_bytes_std_nocero,resp_bytes_std_n
ocero,orig_bytes_min,resp_bytes_min,orig_bytes_max,resp_bytes_max,orig_pkts_nocero,resp_pkts_nocero,orig_pkts_cero,resp_pk
ts_cero,time_mean,time_std,time_min,time_max,orig_time_mean,orig_time_std,orig_time_min,orig_time_max,resp_time_mean,r
esp_time_std,resp_time_min,resp_time_max,label,binary-label\n")
    for line in f_in:
        data = json.loads(line)
        for key in all_keys:
            if key not in data:
                data[key] = ""

        binary_label = 0
        labels = []  # Start empty
        if data["id.resp_h"] == "167.99.182.238" and not label_checked[0]:
            labels.append(label_1)
            binary_label = 1
            label_checked[0] = True

        if data["id.resp_h"] == "167.99.182.238" and (data['resp_ip_bytes'] > 1) and not label_checked[1]:
            labels.append(label_2)
            binary_label = 1
            label_checked[1] = True

        if data["id.resp_p"] == 80 and (data['resp_ip_bytes'] > 50000) and not label_checked[2]:
            labels.append(label_3)
            binary_label = 1
            label_checked[2] = True

        if data["id.resp_p"] == 23 and (data['conn_state'] == 'S0') and not label_checked[3]:
            labels.append(label_4)
            binary_label = 1
            label_checked[3] = True

        if data["id.resp_p"] == 23 and (data['orig_ip_bytes'] > 7) and not label_checked[4]:
            labels.append(label_5)
            binary_label = 1
            label_checked[4] = True

        # Reset label checks if all labels are checked
        # If no labels are added, assign "benign"
        if not labels:
            labels.append("benign")

        values = [str(data[key]) for key in all_keys]
        labels_joined = '-'.join(labels)
        csv_line = ','.join(values) + f',{labels_joined},{binary_label}\n'
        f_out.write(csv_line)

        label_checked = [False] * 5
```

 # Escenario 49

```
import json

input_file = "/root/bbdd/logs-zeek/iot-23-logs/CTU-IoT-Malware-Capture-49-1/conn_stadistics.log"
output_file = "/root/bbdd/logs-zeek/iot-23-logs/labeled-csv/json-labeled-CTU-IoT-Malware-Capture-49-1.csv"

label_1 = "C&C"
label_2 = "PartOfAHorizontalPortScan"
label_3 = "FileDownload"

# Initialize label checks
label_checked = [False] * 3

with open(input_file, "r") as f_in, open(output_file, "w") as f_out:

 f_out.write("ts,startTime,uid,id.orig_h,id.orig_p,id.resp_h,id.resp_p,proto,service,duration,orig_bytes,resp_bytes,conn_state,local_orig,local_resp,missed_bytes,history,orig_pkts,orig_ip_bytes,resp_pkts,resp_ip_bytes,tunnel_parents,orig_bytes_mean,resp_bytes_mean,orig_bytes_std,resp_bytes_std,orig_bytes_mean_nocero,resp_bytes_mean_nocero,orig_bytes_std_nocero,resp_bytes_std_nocero,orig_bytes_min,resp_bytes_min,orig_bytes_max,resp_bytes_max,orig_pkts_nocero,resp_pkts_nocero,orig_pkts_cero,resp_pkts_cero,time_mean,time_std,time_min,time_max,orig_time_mean,orig_time_std,orig_time_min,orig_time_max,resp_time_mean,resp_time_std,resp_time_min,resp_time_max,label,binary-label\n")
   for line in f_in:
      data = json.loads(line)
      for key in all_keys:
         if key not in data:
            data[key] = ""

      binary_label = 0
      labels = []  # Start empty
      if data["id.resp_p"] == 4554 and not label_checked[0]:
         labels.append(label_1)
         binary_label = 1
         label_checked[0] = True

      if data["id.resp_p"] == 8081 and (data['conn_state'] == 'S0') and not label_checked[1]:
         labels.append(label_2)
         binary_label = 1
         label_checked[1] = True

      if data['resp_ip_bytes'] > 30000 and not label_checked[2]:
         labels.append(label_3)
         binary_label = 1
         label_checked[2] = True

      # Reset label checks if all labels are checked
      # If no labels are added, assign "benign"
      if not labels:
         labels.append("benign")

      values = [str(data[key]) for key in all_keys]
      labels_joined = '-'.join(labels)
      csv_line = ','.join(values) + f',{labels_joined},{binary_label}\n'
      f_out.write(csv_line)

      label_checked = [False] * 3
```

 # Escenario 52

```
import json
input_file = "/root/bbdd/logs-zeek/iot-23-logs/CTU-IoT-Malware-Capture-52-1/conn_stadistics.log"
output_file = "/root/bbdd/logs-zeek/iot-23-logs/labeled-csv/json-labeled-CTU-IoT-Malware-Capture-52-1.csv"
label_1 = "C&C"
label_2 = "Mirai"
label_3 = "FileDownload"
label_4 = "PartOfAHorizontalPortscan"

# Initialize label checks
label_checked = [False] * 4
with open(input_file, "r") as f_in, open(output_file, "w") as f_out:
    f_out.write("ts,startTime,uid,id.orig_h,id.orig_p,id.resp_h,id.resp_p,proto,service,duration,orig_bytes,resp_bytes,conn_state,local_orig,local_resp,missed_bytes,history,orig_pkts,orig_ip_bytes,resp_pkts,resp_ip_bytes,tunnel_parents,orig_bytes_mean,resp_bytes_mean,orig_bytes_std,resp_bytes_std,orig_bytes_mean_nocero,resp_bytes_mean_nocero,orig_bytes_std_nocero,resp_bytes_std_nocero,orig_bytes_min,resp_bytes_min,orig_bytes_max,resp_bytes_max,orig_pkts_nocero,resp_pkts_nocero,orig_pkts_cero,resp_pkts_cero,time_mean,time_std,time_min,time_max,orig_time_mean,orig_time_std,orig_time_min,orig_time_max,resp_time_mean,resp_time_std,resp_time_min,resp_time_max,label,binary-label\n")
    for line in f_in:
        data = json.loads(line)
        for key in all_keys:
            if key not in data:
                data[key] = ""
        binary_label = 0
        labels = []  # Start empty
        if data["id.resp_h"] == "185.244.25.108" and not label_checked[0]:
            labels.append(label_1)
            binary_label = 1
            label_checked[0] = True

        if data["id.resp_p"] == 4441 and not label_checked[1]:
            labels.append(label_2)
            binary_label = 1
            label_checked[1] = True

        if data["id.resp_p"] == 80 and (data['resp_ip_bytes'] > 30000) and not label_checked[2]:
            labels.append(label_3)
            binary_label = 1
            label_checked[2] = True

        if data["id.resp_p"] == 23 and not label_checked[3]:
            labels.append(label_4)
            binary_label = 1
            label_checked[3] = True

        if data["id.resp_p"] == 2323 and not label_checked[3]:
            labels.append(label_4)
            binary_label = 1
            label_checked[3] = True

        # Reset label checks if all labels are checked
        # If no labels are added, assign "benign"
        if not labels:
            labels.append("benign")

        values = [str(data[key]) for key in all_keys]
        labels_joined = '-'.join(labels)
        csv_line = ','.join(values) + f',{labels_joined},{binary_label}\n'
        f_out.write(csv_line)

        label_checked = [False] * 4
```

 # Escenario 7

```
import json
input_file = "/root/bbdd/logs-zeek/iot-23-logs/CTU-IoT-Malware-Capture-7-1/conn_stadistics.log"
output_file = "/root/bbdd/logs-zeek/iot-23-logs/labeled-csv/json-labeled-CTU-IoT-Malware-Capture-7-1.csv"
label_1 = "C&C"
label_2 = "Okiru"
label_3 = "HeartBeat"
label_4 = "DDoS"
# Initialize label checks
label_checked = [False] * 4

with open(input_file, "r") as f_in, open(output_file, "w") as f_out:
    f_out.write("ts,startTime,uid,id.orig_h,id.orig_p,id.resp_h,id.resp_p,proto,service,duration,orig_bytes,resp_bytes,conn_state,local_orig,local_resp,missed_bytes,history,orig_pkts,orig_ip_bytes,resp_pkts,resp_ip_bytes,tunnel_parents,orig_bytes_mean,resp_bytes_mean,orig_bytes_std,resp_bytes_std,orig_bytes_mean_nocero,resp_bytes_mean_nocero,orig_bytes_std_nocero,resp_bytes_std_nocero,orig_bytes_min,resp_bytes_min,orig_bytes_max,resp_bytes_max,orig_pkts_nocero,resp_pkts_nocero,orig_pkts_cero,resp_pkts_cero,time_mean,time_std,time_min,time_max,orig_time_mean,orig_time_std,orig_time_min,orig_time_max,resp_time_mean,resp_time_std,resp_time_min,resp_time_max,label,binary-label\n")
    for line in f_in:
        data = json.loads(line)
        for key in all_keys:
            if key not in data:
                data[key] = ""
        binary_label = 0
        labels = []  # Start empty
        if data["id.resp_h"] == "185.130.215.13" and not label_checked[0]:
            labels.append(label_1)
            binary_label = 1
            label_checked[0] = True

        if data["id.resp_h"] == "102.157.125.155" and not label_checked[1]:
            labels.append(label_2)
            binary_label = 1
            label_checked[1] = True

        if data["id.resp_p"] == 37215 and not label_checked[1]:
            labels.append(label_2)
            binary_label = 1
            label_checked[1] = True

        if data["id.resp_p"] == 57722 and not label_checked[2]:
            labels.append(label_3)
            binary_label = 1
            label_checked[2] = True

        if data["id.resp_p"] == 80 and not label_checked[3]:
            labels.append(label_4)
            binary_label = 1
            label_checked[3] = True
        # Reset label checks if all labels are checked
        # If no labels are added, assign "benign"
        if not labels:
            labels.append("benign")

        values = [str(data[key]) for key in all_keys]
        labels_joined = '-'.join(labels)
        csv_line = ','.join(values) + f',{labels_joined},{binary_label}\n'
        f_out.write(csv_line)

        label_checked = [False] * 4
```

 # Etiquetado Benign

```
import pandas as pd

# Read the JSON file into a DataFrame
df = pd.read_json(input_file, lines=True)
# Create two new columns with default values
df['label'] = 'benign'
df['binary-label'] = 0
df.to_csv(output_file, index=False)
# Display the DataFrame
df.head()
```

Finalmente, para etiquetar el conjunto de datos de CIC-IoT-2023, se siguió la misma metodología que emplearon los autores, etiquetando todos los flujos de una misma captura de tráfico según el nombre del archivo, que indica el ataque contenido.

```
def process_conn_log(folder_path):
    conn_log_path = os.path.join(folder_path, "conn_stadistics.log")

    # Check if conn.log file exists
    if os.path.exists(conn_log_path):
        folder_name = os.path.basename(folder_path)
        with open(conn_log_path, 'r') as file:
            header_line = file.readlines()[6].strip().split('\t')[1:]
        df = pd.read_csv(conn_log_path, sep='\t', skiprows=8, names=header_line, skipfooter=1, engine='python')

        print(folder_name)
        # Check for the file with name folder_name + _loss_rows.csv
        loss_rows_path = os.path.join(loss_directory, f'{folder_name}_loss_rows.csv')
        if os.path.exists(loss_rows_path):
            print(f"Loss rows file found: {loss_rows_path}")
            df_loss = pd.read_csv(loss_rows_path)
            # Identify rows to be removed
            rows_to_remove = df[df['uid'].isin(df_loss['uid'])]

            # Print the rows that are going to be removed
            print("Rows to be removed:")
            print(rows_to_remove)
            # Remove rows from df where df['uid'] is in df_loss['uid']
            df = df[~df['uid'].isin(df_loss['uid'])]
            # Save concatenated data frame to CSV
            output_path = "/root/bbdd/logs-zeek/cic-iot-2023-logs/labeled-csv/"  # Change this to the desired directory path
            csv_filename = os.path.join(output_path, f"{folder_name}_labeled.csv")
            df.to_csv(csv_filename, index=False)
        else:
            print(f"Loss rows file not found for {folder_name}")
        # Once found, open that loss file as csv, look for the uids to remove them in the new df we are going to create
    else:
        print(f"conn.log not found in {folder_path}")


for folder in os.listdir(main_directory):
    folder_path = os.path.join(main_directory, folder)
    if os.path.isdir(folder_path):
        process_conn_log(folder_path)
```

Anexo VIII: Normas de etiquetado

Cada base de datos empleó un método distinto de etiquetado, por lo que, para poder recrearlo, se siguieron las reglas proporcionadas por los diferentes autores.

Para la base de datos IoTD20, las reglas se adaptaron para poder etiquetar mediante los atributos generados en Zeek. En concreto, se etiquetó el tráfico benigno y los siguientes ataques: DoS SYN-*Flood*, *PortScan, OS Scan,* UDP *Flood*, HTTP *Flood*, ACK *Flood* y Telnet *BruteForce*.

No.	File Name	Creation Date*	Category	Sub-category	Wireshark Rule to Filter Only Attack Packets
1	benign-dec.pcap	20/05/2019	Normal	Normal	-
8	dos-syn*flood*ing-1-dec.pcap	31/05/2019	Denial of Service (DoS)	SYN *Flood*ing	ip.src == 222.0.0.0/8 and tcp.*flags*.syn == 1 and ip.dst == 192.168.0.13 and tcp.dstport == 554 and tcp
9	dos-syn*flood*ing-2-dec.pcap	31/05/2019	Denial of Service (DoS)	SYN *Flood*ing	ip.src == 222.0.0.0/8 and tcp.*flags*.syn == 1 and ip.dst == 192.168.0.13 and tcp.dstport == 554 and tcp
10	dos-syn*flood*ing-3-dec.pcap	31/05/2019	Denial of Service (DoS)	SYN *Flood*ing	ip.src == 111.0.0.0/8 and tcp.*flags*.syn == 1 and ip.dst == 192.168.0.13 and tcp.dstport == 554 and tcp
11	dos-syn*flood*ing-4-dec.pcap	05/06/2019	Denial of Service (DoS)	SYN *Flood*ing	ip.dst == 192.168.0.24 and tcp.*flags*.syn == 1 and ip.src == 111.0.0.0/8 and tcp and tcp.dstport == 19604
12	dos-syn*flood*ing-5-dec.pcap	05/06/2019	Denial of Service (DoS)	SYN *Flood*ing	ip.dst == 192.168.0.24 and tcp.*flags*.syn == 1 and ip.src == 111.0.0.0/8 and tcp and tcp.dstport == 19604
13	dos-syn*flood*ing-6-dec.pcap	05/06/2019	Denial of Service (DoS)	SYN *Flood*ing	ip.dst == 192.168.0.24 and tcp.*flags*.syn == 1 and ip.src == 111.0.0.0/8 and tcp and tcp.dstport == 19604
14	scan-hostport-1-dec.pcap	11/07/2019	Scanning	*Port Scan*ning	ip.src == 192.168.0.15 and ip.dst == 192.168.0.13 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1)
15	scan-hostport-2-dec.pcap	11/07/2019	Scanning	*Port Scan*ning	ip.src == 192.168.0.15 and ip.dst == 192.168.0.13 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1)
16	scan-hostport-3-dec.pcap	11/07/2019	Scanning	*Port Scan*ning	ip.src == 192.168.0.15 and ip.dst == 192.168.0.13 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1)
17	scan-hostport-4-dec.pcap	11/07/2019	Scanning	*Port Scan*ning	ip.src == 192.168.0.15 and ip.dst == 192.168.0.24 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1)

18	scan-hostport-5-dec.pcap	11/07/2019	Scanning	*Port Scan*ning	ip.src == 192.168.0.15 and ip.dst == 192.168.0.24 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1)
19	scan-hostport-6-dec.pcap	11/07/2019	Scanning	*Port Scan*ning	ip.src == 192.168.0.15 and ip.dst == 192.168.0.24 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1)
20	scan-portos-1-dec.pcap	11/07/2019	Scanning	*Port Scan*ning	ip.src == 192.168.0.15 and ip.dst == 192.168.0.13 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1)
			Scanning	OS/Version Detection	(ip.src == 192.168.0.15 and ip.dst == 192.168.0.13) and (not icmp) and not (ip.src == 192.168.0.15 and ip.dst == 192.168.0.13 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1))
21	scan-portos-2-dec.pcap	11/07/2019	Scanning	*Port Scan*ning	ip.src == 192.168.0.15 and ip.dst == 192.168.0.13 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1)
			Scanning	OS/Version Detection	(ip.src == 192.168.0.15 and ip.dst == 192.168.0.13) and (not icmp) and not (ip.src == 192.168.0.15 and ip.dst == 192.168.0.13 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1))
22	scan-portos-3-dec.pcap	11/07/2019	Scanning	*Port Scan*ning	ip.src == 192.168.0.15 and ip.dst == 192.168.0.13 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1)
			Scanning	OS/Version Detection	(ip.src == 192.168.0.15 and ip.dst == 192.168.0.13) and (not icmp) and not (ip.src == 192.168.0.15 and ip.dst == 192.168.0.13 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1))
23	scan-portos-4-dec.pcap	11/07/2019	Scanning	*Port Scan*ning	ip.src == 192.168.0.15 and ip.dst == 192.168.0.24 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1)
			Scanning	OS/Version Detection	(ip.src == 192.168.0.15 and ip.dst == 192.168.0.24) and (not icmp) and not (ip.src == 192.168.0.15 and ip.dst == 192.168.0.24 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1))
24	scan-portos-5-dec.pcap	11/07/2019	Scanning	*Port Scan*ning	ip.src == 192.168.0.15 and ip.dst == 192.168.0.24 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1)
			Scanning	OS/Version Detection	(ip.src == 192.168.0.15 and ip.dst == 192.168.0.24) and (not icmp) and not (ip.src == 192.168.0.15 and ip.dst == 192.168.0.24 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1))
25	scan-portos-6-dec.pcap	11/07/2019	Scanning	*Port Scan*ning	ip.src == 192.168.0.15 and ip.dst == 192.168.0.24 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1)

			Scanning	OS/Version Detection	(ip.src == 192.168.0.15 and ip.dst == 192.168.0.24) and (not icmp) and not (ip.src == 192.168.0.15 and ip.dst == 192.168.0.24 and ((tcp.*flags*.syn == 1 and tcp.window_size == 1024) or tcp.*flags*.reset == 1))
26	mirai-udp*flood*ing-1-dec.pcap	01/08/2019	Mirai *Botnet*	UDP *Flood*ing	ip.dst == 210.89.164.90
27	mirai-udp*flood*ing-2-dec.pcap	01/08/2019	Mirai *Botnet*	UDP *Flood*ing	ip.dst == 210.89.164.90
28	mirai-udp*flood*ing-3-dec.pcap	01/08/2019	Mirai *Botnet*	UDP *Flood*ing	ip.dst == 210.89.164.90
29	mirai-udp*flood*ing-4-dec.pcap	01/08/2019	Mirai *Botnet*	UDP *Flood*ing	ip.dst == 210.89.164.90
30	mirai-ack*flood*ing-1-dec.pcap	01/08/2019	Mirai *Botnet*	ACK *Flood*ing	ip.dst == 210.89.164.90
31	mirai-ack*flood*ing-2-dec.pcap	01/08/2019	Mirai *Botnet*	ACK *Flood*ing	ip.dst == 210.89.164.90
32	mirai-ack*flood*ing-3-dec.pcap	01/08/2019	Mirai *Botnet*	ACK *Flood*ing	ip.dst == 210.89.164.90
33	mirai-ack*flood*ing-4-dec.pcap	01/08/2019	Mirai *Botnet*	ACK *Flood*ing	ip.dst == 210.89.164.90
34	mirai-http*flood*ing-1-dec.pcap	01/08/2019	Mirai *Botnet*	HTTP *Flood*ing	ip.dst == 210.89.164.90
35	mirai-http*flood*ing-2-dec.pcap	01/08/2019	Mirai *Botnet*	HTTP *Flood*ing	ip.dst == 210.89.164.90
36	mirai-http*flood*ing-3-dec.pcap	01/08/2019	Mirai *Botnet*	HTTP *Flood*ing	ip.dst == 210.89.164.90
37	mirai-http*flood*ing-4-dec.pcap	01/08/2019	Mirai *Botnet*	HTTP *Flood*ing	ip.dst == 210.89.164.90
38	mirai-host*bruteforce*-1-dec.pcap	05/09/2019	Mirai *Botnet*	Telnet *Bruteforce*	tcp.dstport==23 and ip.src==192.168.0.13
39	mirai-host*bruteforce*-2-dec.pcap	05/09/2019	Mirai *Botnet*	Telnet *Bruteforce*	tcp.dstport==23 and ip.src==192.168.0.24
40	mirai-host*bruteforce*-3-dec.pcap	10/09/2019	Mirai *Botnet*	Telnet *Bruteforce*	tcp.dstport==23 and ip.src==192.168.0.13
41	mirai-host*bruteforce*-4-dec.pcap	10/09/2019	Mirai *Botnet*	Telnet *Bruteforce*	tcp.dstport==23 and ip.src==192.168.0.24
42	mirai-host*bruteforce*-5-dec.pcap	10/09/2019	Mirai *Botnet*	Telnet *Bruteforce*	tcp.dstport==23 and ip.src==192.168.0.13

Para el *dataset* IoT-23, se siguieron las siguientes reglas:

CTU-IoT-*Malware*-Capture-7-1 (Linux, Mirai)

Id	Field	bro field number	Data	Comparator	Label	type	connector
1	id.resp_h	5	185.130.215.13	eq	C&C	Malicious	-
2	id.resp_h	5	102.157.125.155	eq	Okiru	Malicious	-
3	id.resp_p	6	37215	eq	Okiru	Malicious	-
4	id.resp_p	6	57722	eq	HeartBeat	Malicious	-
5	id.resp_p	6	80	eq	DDoS	Malicious	-

CTU-IoT-*Malware*-Capture-34-1 (Mirai)

Id	Field	bro field number	Data	Comparator	Label	type	connector
1	id.resp_p	6	6667	eq	C&C	Malicious	-
2	id.resp_p	6	63798	eq	PartOfAHorizontal*Portscan*	Malicious	-
3	id.resp_p	6	256	eq	PartOfAHorizontal*Portscan*	Malicious	-
4	id.resp_h	5	123.59.209.185	eq	DDoS	Malicious	-
5	id.resp_h	5	71.61.66.148	eq	DDoS	Malicious	-
6	id.resp_h	5	74.91.117.248	eq	DDoS	Malicious	-
7	id.resp_p	6	5376	eq	DDoS	Malicious	-

CTU-IoT-*Malware*-Capture-35-1 (Mirai)

Id	Field	bro field number	Data	Comparator	Label	type	connector
1	id.resp_h	5	104.248.160.24	eq	C&C	Malicious	-
2	id.resp_h	5	104.248.160.24	eq	FileDownload	Malicious	and 3
3	resp_ip_*bytes*	19	30000	gt	FileDownload	Malicious	and 2
4	id.resp_h	5	110.183.76.177	eq	Attack	Malicious	-
5	id.resp_h	5	112.27.30.87	eq	Attack	Malicious	-

6	id.resp_p	6	85.217.225.181	eq	Attack	Malicious	-
7	id.resp_p	6	992	eq	DDoS	Malicious	-
8	id.resp_h	5	209.97.190.136	eq	DDoS	Malicious	-
9	id.resp_h	5	173.113.172.138	eq	DDoS	Malicious	-
10	id.resp_h	5	216.18.168.16	eq	DDoS	Malicious	-
11	id.resp_h	5	24.165.115.195	eq	DDoS	Malicious	-
12	id.resp_h	5	54.39.87.104	eq	DDoS	Malicious	-

CTU-IoT-*Malware*-Capture-43-1 (Mirai)

Id	Field	bro field number	Data	Comparator	Label	type	connector
1	id.resp_p	6	45	eq	C&C	Malicious	-
2	id.resp_h	5	142.11.219.83	eq	C&C	Malicious	-
3	id.resp_p	6	27015	eq	DDoS	Malicious	-
4	id.resp_p	6	37215	eq	Okiru	Malicious	-
5	id.resp_p	6	52869	eq	PartOfAHorizontal*Portscan*	Malicious	and 6
6	conn_state	12	S0	eq	PartOfAHorizontal*PortScan*	Malicious	and 5
7	resp_ip_*bytes*	19	50000	gt	FileDonwload	Malicious	-

CTU-IoT-*Malware*-Capture-44-1 (Mirai)

Id	Field	bro field number	Data	Comparator	Label	type	connector
1	id.resp_h	5	46.101.251.172	eq	C&C	Malicious	and 2
2	proto	7	tcp	eq	C&C	Malicious	and 1
3	id.resp_p	6	80	eq	DDoS	Malicious	and 4
4	proto	7	udp	eq	DDoS	Malicious	and 3
5	id.resp_h	5	86.136.151.56	eq	DDoS	Malicious	-
6	resp_ip_*bytes*	19	50000	gt	FileDownload	Malicious	-

CTU-IoT-*Malware*-Capture-48-1 (Mirai)

Id	Field	bro field number	Data	Comparator	Label	type	connector
1	id.resp_h	5	167.99.182.238	eq	C&C	Malicious	-
2	id.resp_h	5	167.99.182.238	eq	HeartBeat	Malicious	and 3
3	resp_ip_*bytes*	19	1	gt	HeartBeat	Malicious	and 2
4	id.resp_p	6	80	eq	FileDownload	Malicious	and 5
5	resp_ip_*bytes*	19	50000	gt	FileDownload	Malicious	and 4
6	id.resp_p	6	23	eq	PartOfAHorizontal*Portscan*	Malicious	and 7
7	conn_state	12	S0	eq	PartOfAHorizontal*PortScan*	Malicious	and 6
8	id.resp_p	6	23	eq	Attack	Malicious	and 9
9	orig_ip_*bytes*	17	7	gt	Attack	Malicious	and 8

CTU-IoT-*Malware*-Capture-49-1 (Mirai)

Id	Field	bro field number	Data	Comparator	Label	type	connector
1	id.resp_p	6	4554	eq	C&C	Malicious	-
2	id.resp_p	6	8081	eq	PartOfAHorizontal*Portscan*	Malicious	and 3
3	conn_state	12	S0	eq	PartOfAHorizontal*PortScan*	Malicious	and 2
4	resp_ip_*bytes*	19	30000	gt	FileDownload	Malicious	-

CTU-IoT-*Malware*-Capture-52-1 (Mirai)

Id	Field	bro field number	Data	Comparator	Label	type	connector
1	id.resp_h	5	185.244.25.108	eq	C&C	Malicious	-
2	id.resp_p	6	4441	eq	Mirai	Malicious	-
3	id.resp_p	6	80	eq	FileDownload	Malicious	and 4
4	resp_ip_*bytes*	19	30000	gt	FileDownload	Malicious	and 3
5	id.resp_p	6	23	eq	PartOfAHorizontal*Portscan*	Malicious	-
6	id.resp_p	6	2323	eq	PartOfAHorizontal*Portscan*	Malicious	-

Anexo IX: *Scripts* de capture-loss

Como ya se ha desarrollado en el capítulo , ha sido necesario el estudio de pérdida de información en las capturas de tráfico. Esto se pudo observar mediante la herramienta Zui, y por ello se decidió analizar en profundidad la cantidad de flujos que presentaban pérdida de información, y la magnitud de esta pérdida.

Los *scripts* que se desarrollaron se muestran a continuación. Primero, se llevó a cabo un análisis en menor profundidad mediante estadísticas, para ello se empleó:

```
import os
import pandas as pd
# Define the source directory containing capture loss logs
source_dir = "/root/capture_loss_iotd20_logs"
# Define the output file
output_file = "/root/capture_loss_iotd20_logs/concatenated_logs.csv"
# Create an empty list to store DataFrames
dfs = []
# Iterate through each JSON file in the source directory
for file_name in os.listdir(source_dir):
    if file_name.endswith(".log"):
        file_path = os.path.join(source_dir, file_name)
        # Load the JSON file into a pandas DataFrame
        df = pd.read_json(file_path, lines=True)
        # Add a new column with the file name
        df['file_name'] = file_name
        # Append the DataFrame to the list
        dfs.append(df)
# Concatenate all DataFrames into a single DataFrame
concatenated_df = pd.concat(dfs, ignore_index=True)
concatenated_df.to_csv(output_file, index=False)
print("Concatenation completed. Output file:", output_file)
print("Número de filas en el DataFrame:", df.shape[0])
csv_file = output_file
df = pd.read_csv(csv_file)
# List the rows with the highest values in the percent_lost column
top_percent_lost = df.nlargest(34, 'percent_lost')  # Change 10 to the desired number of rows
selected_columns = ['ts_delta', 'gaps', 'acks', 'file_name', 'percent_lost']
top_percent_lost_selected = top_percent_lost[selected_columns]

print(top_percent_lost_selected)
# Ordenar primero por el nombre del archivo y luego por el porcentaje perdido
top_percent_lost_sorted = top_percent_lost_selected.sort_values(by=['file_name', 'percent_lost'], ascending=[True, False])

print(top_percent_lost_sorted)
import os
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
main_directory = "/root/bbdd/logs-zeek/iotd20-logs/logs-original/"
# Function to process a conn.log file
```

```
def process_conn_log(folder_path):
    conn_log_path = os.path.join(folder_path, "conn.log")

    # Check if conn.log file exists
    if os.path.exists(conn_log_path):
        # Extract file name from folder path
        folder_name = os.path.basename(folder_path)

        # Read conn.log into a DataFrame
        df = pd.read_json(conn_log_path, lines=True)

        # Convert timestamp to datetime if needed
        # df["timestamp"] = pd.to_datetime(df["timestamp"])

        # Create a dot plot of missed_bytes evolution
        plt.scatter(df["ts"], df["missed_bytes"], s=10, marker='o')
        plt.xlabel("Timestamp")
        plt.ylabel("Missed Bytes")
        plt.title(f"Evolution of Missed Bytes - {folder_name}")
        plt.show()

        # Calculate statistics
        stats = df["missed_bytes"].describe()
        print("Statistics:")
        print(stats)

        # Print top 15 highest values
        top_15 = df.nlargest(15, "missed_bytes")
        print("\nTop 15 highest missed_bytes:")
        print(top_15[["uid", "missed_bytes", "id.orig_h", "id.orig_p", "id.resp_h", "id.resp_p", "proto"]])

    else:
        print(f"conn.log not found in {folder_path}")
def process_conn_log(folder_path, thresholds=[(0, 0.05), (0.05, 0.1), (0.1, 0.2), (0.2, 1)]):
    conn_log_path = os.path.join(folder_path, "conn_stadistics.log")

    # Check if conn.log file exists
    if os.path.exists(conn_log_path):
        # Extract file name from folder path
        folder_name = os.path.basename(folder_path)

        # Read conn.log into a DataFrame
        df = pd.read_json(conn_log_path, lines=True)

        # Sample 5 flows with NaN values in either orig_bytes or resp_bytes before dropping
        nan_flows = df[df['orig_bytes'].isnull() | df['resp_bytes'].isnull()].head(5)
        if not nan_flows.empty:
            print("Sample of 5 flows with NaN values in either orig_bytes or resp_bytes:")
            print(nan_flows[["uid","missed_bytes","orig_bytes","resp_bytes", "id.orig_h", "id.orig_p", "id.resp_h", "id.resp_p", "proto"]])

        # Notify and count rows with NaN values in both orig_bytes and resp_bytes columns
        nan_count = df[df['orig_bytes'].isnull() & df['resp_bytes'].isnull()].shape[0]
        if nan_count > 0:
```

```
            print(f"{nan_count} rows with NaN values in both orig_bytes and resp_bytes columns.")

        # Calculate missed_bytes ratio
        df['missed_ratio'] = np.where(df['missed_bytes'] == 0, 0, df['missed_bytes'] / (df['orig_bytes'] + df['resp_bytes']))

        # Sort DataFrame by missed_ratio in descending order
        df_sorted = df.sort_values(by='missed_ratio', ascending=False)

        # Create a dot plot of missed_bytes ratio evolution
        plt.scatter(range(1, len(df_sorted) + 1), df_sorted["missed_ratio"], s=10, marker='o')
        plt.xlabel("Rank Position (sorted by missed ratio)")
        plt.ylabel("Missed Bytes Ratio")
        plt.title(f"Evolution of Missed Bytes Ratio - {folder_name}")
        plt.show()

        # Calculate statistics
        stats = df_sorted["missed_ratio"].describe()
        print("Statistics:")
        print(stats)

        # Print top 15 highest values
        top_15 = df_sorted.head(15)
        print("\nTop 15 highest missed_bytes ratios:")
        print(top_15[["uid", "missed_ratio", "id.orig_h", "id.orig_p", "id.resp_h", "id.resp_p", "proto"]])

        # Print the least 15 flows based on missed bytes ratio
        print("\nLeast 15 flows based on missed bytes ratio:")
        least_15 = df_sorted.tail(15)
        print(least_15[["uid", "missed_ratio", "orig_bytes","resp_bytes", "id.orig_h", "id.orig_p", "id.resp_h", "id.resp_p", "proto"]])
        # Initialize lists to store counts for each threshold range
        threshold_counts = [0] * len(thresholds)
            # Count number of flows in each threshold range
        for i, (lower, upper) in enumerate(thresholds):
            if upper is None:
                num_flows = (df_sorted['missed_ratio'] >= lower).sum()
            else:
                num_flows = ((df_sorted['missed_ratio'] >= lower) & (df_sorted['missed_ratio'] <= upper)).sum()
            threshold_counts[i] = num_flows

        # Create grouped bar plot for the number of flows exceeding each threshold range
        threshold_ranges = [f"{int(lower * 100)}% - {int(upper * 100)}%" if upper is not None else f"> {int(lower * 100)}%" for lower, upper
in thresholds]
        plt.bar(threshold_ranges, threshold_counts)
        plt.xlabel("Missed Bytes Ratio Threshold Range")
        plt.ylabel("Number of Flows")
        plt.title(f"Number of Flows Exceeding Missed Bytes Ratio Thresholds - {folder_name}")
        plt.show()
    else:
        print(f"conn.log not found in {folder_path}")
for folder in os.listdir(main_directory):
    folder_path = os.path.join(main_directory, folder)
    if os.path.isdir(folder_path):
        process_conn_log(folder_path)
```

Posteriormente, al ver que había flujos con una cantidad de pérdida significante, se diseñó otro *script* para detectarlos dentro de los conjuntos de datos, y eliminar todos aquellos que superasen la cantidad de 1% de pérdida de información.

Para IoTD20 se realizó sobre el *dataset* entero, sin diferenciar por archivos, ya que su tamaño era reducido.

```
import pandas as pd
import numpy as np
import os

def process_conn_log(folder_path,columns = ['uid','missed_bytes','orig_bytes','resp_bytes']):

    df = pd.read_csv(folder_path,usecols=columns)
    df.loc[df['missed_bytes'] == '-', 'missed_bytes'] = np.nan
    df.loc[df['orig_bytes'] == '-', 'missed_bytes'] = np.nan
    df.loc[df['resp_bytes'] == '-', 'missed_bytes'] = np.nan

    # Convert remaining NaNs to 0 after substitution
    df['missed_bytes'] = pd.to_numeric(df['missed_bytes'], errors='coerce').fillna(0)
    df['orig_bytes'] = pd.to_numeric(df['orig_bytes'], errors='coerce').fillna(0)
    df['resp_bytes'] = pd.to_numeric(df['resp_bytes'], errors='coerce').fillna(0)

    df['missed_ratio'] = np.where((df['missed_bytes'].isna()) | (df['missed_bytes'] == 0), 0, pd.to_numeric(df['missed_bytes']) /
(pd.to_numeric(df['orig_bytes']) + pd.to_numeric(df['resp_bytes'])))

    # Filter rows with loss > 0.01 and append to list
    filtered_chunk = df[df['missed_ratio'] > 0.01]

    # Filter rows with loss > 0.01 and append to list
    filtered_chunk = df[df['missed_ratio'] > 0.01].copy()  # Make a copy to avoid the warning
    filtered_chunk.loc[:, 'missed_ratio'] = df['missed_ratio']  # Assign values using .loc[]

    print("all_labeled_loss_rows.csv created")
    # Save concatenated data frame to CSV
    output_path = "/root/bbdd/logs-zeek/iotd20-logs/loss-rows/"  # Change this to the desired directory path
    csv_filename = os.path.join(output_path, "all_labeled_loss_rows.csv")
    filtered_chunk.to_csv(csv_filename, index=False)
main_directory = "/root/bbdd/logs-zeek/iotd20-logs/all-labeled_all.csv"
process_conn_log(main_directory)
loss_rows_path = "/root/bbdd/logs-zeek/iotd20-logs/loss-rows/all_labeled_loss_rows.csv"
df = pd.read_csv(main_directory)
# Check for the file with name folder_name + _loss_rows.csv
df_loss = pd.read_csv(loss_rows_path)
# Identify rows to be removed
rows_to_remove = df[df['uid'].isin(df_loss['uid'])]

# Print the rows that are going to be removed
print("Rows to be removed:")
print(rows_to_remove)
# Remove rows from df where df['uid'] is in df_loss['uid']
df = df[~df['uid'].isin(df_loss['uid'])]
# Save concatenated data frame to CSV
output_path = "/root/bbdd/logs-zeek/iotd20-logs/all-labeled-final.csv"  # Change this to the desired directory path
df.to_csv(output_path, index=False)
```

En el caso del conjunto de datos IoT-23, se diseñó este *script*:

```
import os
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
main_directory = "/root/bbdd/logs-zeek/iot-23-logs/labeled-csv/"
import pandas as pd
def process_conn_log(folder_name,columns = ['uid','missed_bytes','orig_bytes','resp_bytes']):
    conn_log_path = os.path.join(main_directory, folder_name)
    loss_rows_df = []  # Initialize a list to store data frames for chunks with loss > 0.01

    # Check if conn.log file exists
    if os.path.exists(conn_log_path):
        for chunk in pd.read_csv(conn_log_path,usecols=columns, chunksize=50000):
            # Calculate missed_bytes ratio
            chunk['missed_ratio'] = np.where((chunk['missed_bytes'].isna()) | (chunk['missed_bytes'] == 0), 0,
pd.to_numeric(chunk['missed_bytes']) / (pd.to_numeric(chunk['orig_bytes']) + pd.to_numeric(chunk['resp_bytes'])))
            # Filter rows with loss > 0.01 and append to list
            filtered_chunk = chunk[chunk['missed_ratio'] > 0.01].copy()
            filtered_chunk.loc[:, 'missed_ratio'] = chunk['missed_ratio']
            loss_rows_df.append(filtered_chunk)

    # Concatenate data frames in the list
    loss_rows_df = pd.concat(loss_rows_df)
    # Save concatenated data frame to CSV
    output_path = "/root/bbdd/logs-zeek/iot-23-logs/loss-rows/"  # Change this to the desired directory path
    csv_filename = os.path.join(output_path, f"{folder_name}_loss_rows.csv")
    loss_rows_df.to_csv(csv_filename, index=False)
json_files = [f for f in os.listdir(main_directory) if f.startswith("json")]
for json_file in json_files:
        process_conn_log(json_file)
```

En el caso de CIC-IoT-2023, además de sustituir ciertos valores vacíos, se extraen los flujos con más de 1% de pérdidas, almacenándolos en otro fichero para su posterior análisis además de eliminarlos del conjunto de datos, identificándolos con el campo “uid”.

```
import pandas as pd
import numpy as np
import os
header = [
    'ts', 'startTime', 'uid', 'id.orig_h', 'id.orig_p', 'id.resp_h', 'id.resp_p', 'proto', 'service', 'duration',
    'orig_bytes', 'resp_bytes', 'conn_state', 'local_orig', 'local_resp', 'missed_bytes', 'history', 'orig_pkts',
    'orig_ip_bytes', 'resp_pkts', 'resp_ip_bytes', 'tunnel_parents', 'orig_bytes_mean', 'resp_bytes_mean',
    'orig_bytes_std', 'resp_bytes_std', 'orig_bytes_mean_nocero', 'resp_bytes_mean_nocero', 'orig_bytes_std_nocero',
    'resp_bytes_std_nocero', 'orig_bytes_min', 'resp_bytes_min', 'orig_bytes_max', 'resp_bytes_max', 'orig_pkts_nocero',
    'resp_pkts_nocero', 'orig_pkts_cero', 'resp_pkts_cero', 'time_mean', 'time_std', 'time_min', 'time_max',
    'orig_time_mean', 'orig_time_std', 'orig_time_min', 'orig_time_max', 'resp_time_mean', 'resp_time_std',
    'resp_time_min', 'resp_time_max' ]
def process_conn_log(folder_path,columns = ['uid','missed_bytes','orig_bytes','resp_bytes']):
    conn_log_path = os.path.join(folder_path, "conn_stadistics.log")

    # Check if conn.log file exists
    if os.path.exists(conn_log_path):
        folder_name = os.path.basename(folder_path)
        #with open(conn_log_path, 'r') as file:
            #header_line = file.readlines()[6].strip().split('\t')[1:]
        df = pd.read_csv(conn_log_path, sep='\t', skiprows=8, names=header, skipfooter=1, engine='python',usecols=columns)
        df.loc[df['missed_bytes'] == '-', 'missed_bytes'] = np.nan
        df.loc[df['orig_bytes'] == '-', 'missed_bytes'] = np.nan
        df.loc[df['resp_bytes'] == '-', 'missed_bytes'] = np.nan

        # Convert remaining NaNs to 0 after substitution
        df['missed_bytes'] = pd.to_numeric(df['missed_bytes'], errors='coerce').fillna(0)
        df['orig_bytes'] = pd.to_numeric(df['orig_bytes'], errors='coerce').fillna(0)
        df['resp_bytes'] = pd.to_numeric(df['resp_bytes'], errors='coerce').fillna(0)
```

```
    df['missed_ratio'] = np.where((df['missed_bytes'].isna()) | (df['missed_bytes'] == 0), 0, pd.to_numeric(df['missed_bytes']) /
(pd.to_numeric(df['orig_bytes']) + pd.to_numeric(df['resp_bytes'])))

    # Filter rows with loss > 0.01 and append to list
    filtered_chunk = df[df['missed_ratio'] > 0.01]

    # Filter rows with loss > 0.01 and append to list
    filtered_chunk = df[df['missed_ratio'] > 0.01].copy()  # Make a copy to avoid the warning
    filtered_chunk.loc[:, 'missed_ratio'] = df['missed_ratio']  # Assign values using .loc[]

  print(f"{folder_name}_loss_rows.csv created")
  # Save concatenated data frame to CSV
  output_path = "/root/bbdd/logs-zeek/cic-iot-2023-logs/loss-rows/"  # Change this to the desired directory path
  csv_filename = os.path.join(output_path, f"{folder_name}_loss_rows.csv")
  filtered_chunk.to_csv(csv_filename, index=False)
```

Anexo X: *Scripts* de representación de capture-loss

```
# Create an empty list to store DataFrames
dfs = []
# Iterate through each JSON file in the source directory
for file_name in os.listdir(source_dir):
    if file_name.endswith(".log"):
        file_path = os.path.join(source_dir, file_name)

        # Load the JSON file into a pandas DataFrame
        df = pd.read_json(file_path, lines=True)

        # Add a new column with the file name
        df['file_name'] = file_name

        # Append the DataFrame to the list
        dfs.append(df)
# Concatenate all DataFrames into a single DataFrame
concatenated_df = pd.concat(dfs, ignore_index=True)
concatenated_df.to_csv(output_file, index=False)
print("Concatenation completed. Output file:", output_file)
csv_file = output_file
df = pd.read_csv(csv_file)

# List the rows with the highest values in the percent_lost column
top_percent_lost = df.nlargest(34, 'percent_lost')  # Change 10 to the desired number of rows
selected_columns = ['ts_delta', 'gaps', 'acks', 'file_name', 'percent_lost']
top_percent_lost_selected = top_percent_lost[selected_columns]

print(top_percent_lost_selected)
# Ordenar primero por el nombre del archivo y luego por el porcentaje perdido
top_percent_lost_sorted = top_percent_lost_selected.sort_values(by=['file_name', 'percent_lost'], ascending=[True, False])

print(top_percent_lost_sorted)
# Function to process a conn.log file
def process_conn_log(folder_path, thresholds=[(0, 0.05), (0.05, 0.1), (0.1, 0.2), (0.2, 1)]):
    conn_log_path = os.path.join(folder_path, "conn_stadistics.log")

    # Check if conn.log file exists
    if os.path.exists(conn_log_path):
        # Extract file name from folder path
        folder_name = os.path.basename(folder_path)

        # Read conn.log into a DataFrame
        df = pd.read_json(conn_log_path, lines=True)

        # Sample 5 flows with NaN values in either orig_bytes or resp_bytes before dropping
        nan_flows = df[df['orig_bytes'].isnull() | df['resp_bytes'].isnull()].head(5)
        if not nan_flows.empty:
            print("Sample of 5 flows with NaN values in either orig_bytes or resp_bytes:")
            print(nan_flows[["uid","missed_bytes","orig_bytes","resp_bytes", "id.orig_h", "id.orig_p", "id.resp_h", "id.resp_p", "proto"]])

        # Notify and count rows with NaN values in both orig_bytes and resp_bytes columns
        nan_count = df[df['orig_bytes'].isnull() & df['resp_bytes'].isnull()].shape[0]
        if nan_count > 0:
            print(f"{nan_count} rows with NaN values in both orig_bytes and resp_bytes columns.")

        # Calculate missed_bytes ratio
        df['missed_ratio'] = np.where(df['missed_bytes'] == 0, 0, df['missed_bytes'] / (df['orig_bytes'] + df['resp_bytes']))

        # Sort DataFrame by missed_ratio in descending order
        df_sorted = df.sort_values(by='missed_ratio', ascending=False)

        # Create a dot plot of missed_bytes ratio evolution
        plt.scatter(range(1, len(df_sorted) + 1), df_sorted["missed_ratio"], s=10, marker='o')
```

```
    plt.xlabel("Rank Position (sorted by missed ratio)")
    plt.ylabel("Missed Bytes Ratio")
    plt.title(f"Evolution of Missed Bytes Ratio - {folder_name}")
    plt.show()

    # Calculate statistics
    stats = df_sorted["missed_ratio"].describe()
    print("Statistics:")
    print(stats)

    # Print top 15 highest values
    top_15 = df_sorted.head(15)
    print("\nTop 15 highest missed_bytes ratios:")
    print(top_15[["uid", "missed_ratio", "id.orig_h", "id.orig_p", "id.resp_h", "id.resp_p", "proto"]])

    # Print the least 15 flows based on missed bytes ratio
    print("\nLeast 15 flows based on missed bytes ratio:")
    least_15 = df_sorted.tail(15)
    print(least_15[["uid", "missed_ratio", "orig_bytes","resp_bytes", "id.orig_h", "id.orig_p", "id.resp_h", "id.resp_p", "proto"]])

    # Initialize lists to store counts for each threshold range
    threshold_counts = [0] * len(thresholds)

    # Count number of flows in each threshold range
    for i, (lower, upper) in enumerate(thresholds):
      if upper is None:
        num_flows = (df_sorted['missed_ratio'] >= lower).sum()
      else:
        num_flows = ((df_sorted['missed_ratio'] >= lower) & (df_sorted['missed_ratio'] <= upper)).sum()
      threshold_counts[i] = num_flows

    # Create grouped bar plot for the number of flows exceeding each threshold range
    threshold_ranges = [f"{int(lower * 100)}% - {int(upper * 100)}%" if upper is not None else f"> {int(lower * 100)}%" for lower,
upper in thresholds]
    plt.bar(threshold_ranges, threshold_counts)
    plt.xlabel("Missed Bytes Ratio Threshold Range")
    plt.ylabel("Number of Flows")
    plt.title(f"Number of Flows Exceeding Missed Bytes Ratio Thresholds - {folder_name}")
    plt.show()

  else:
    print(f"conn.log not found in {folder_path}")


for folder in os.listdir(main_directory):
  folder_path = os.path.join(main_directory, folder)
  if os.path.isdir(folder_path):
    process_conn_log(folder_path)
```

Para Iot-23

```
import os
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from bokeh.plotting import figure, show, output_file
from bokeh.models import HoverTool
from bokeh.io import export_png

main_directory = "/root/bbdd/logs-zeek/iot-23-logs/labeled-csv/"

def process_conn_log(folder_name, thresholds=[(0, 0.05), (0.05, 0.1), (0.1, 0.2), (0.2, 1)], columns=["uid", "missed_bytes",
"orig_bytes","resp_bytes", "id.orig_h", "id.orig_p", "id.resp_h", "id.resp_p", "proto", 'binary-label']):
  conn_log_path = os.path.join(main_directory, folder_name)

  # Check if conn.log file exists
  if os.path.exists(conn_log_path)
```

```
        # Create Bokeh plot
        p = figure(title=f"Evolution of Missed Bytes Ratio - {folder_name}", x_axis_label="Rank Position (sorted by missed ratio)",
y_axis_label="Missed Bytes Ratio")

        # Add hover tool
        hover = HoverTool()
        hover.tooltips = [("Index", "$index"), ("Missed Bytes Ratio", "@missed_ratio")]
        p.add_tools(hover)
        for chunk in pd.read_csv(conn_log_path, usecols=columns, chunksize=50000):
            # Calculate missed_bytes ratio
            chunk['missed_ratio'] = np.where((chunk['missed_bytes'].isna()) | (chunk['missed_bytes'] == 0), 0,
pd.to_numeric(chunk['missed_bytes']) / (pd.to_numeric(chunk['orig_bytes']) + pd.to_numeric(chunk['resp_bytes'])))
            if chunk['missed_ratio'].isnull().values.any():
                print("Warning: NaN value detected in missed_ratio column!")
            # Define color based on binary-label
            colors = ['green' if label == 0 else 'red' for label in chunk['binary-label']]
            # Add scatter plot for the chunk
            p.scatter(list(range(1, len(chunk) + 1)), chunk["missed_ratio"], size=10, color=colors, alpha=0.5)

        # Show plot for the chunk
        export_png(p,filename=f"{folder_name}.png")
        show(p)
        print(f"Saved in: {folder_name}")

    else:
        print(f"conn.log not found in {folder_path}")

json_files = [f for f in os.listdir(main_directory) if f.startswith("json")]
for json_file in json_files:
        process_conn_log(json_file)
```

Anexo XI: *Scripts* de obtención de resultados

***Train-test* para IoT-23**

```
import dask.dataframe as dd
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay, roc_curve, auc, RocCurveDisplay
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB, BernoulliNB
from sklearn.linear_model import SGDClassifier
from sklearn.ensemble import BaggingClassifier, AdaBoostClassifier, RandomForestClassifier
from sklearn.neighbors import NearestCentroid
from sklearn.neural_network import MLPClassifier
from fpdf import FPDF
import matplotlib.pyplot as plt
import os
from sklearn import tree
import time
import pandas as pd
from concurrent.futures import ThreadPoolExecutor, as_completed
from sklearn.preprocessing import StandardScaler, LabelBinarizer
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import roc_curve, auc, accuracy_score
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import (classification_report, accuracy_score, confusion_matrix,
                             ConfusionMatrixDisplay, roc_curve, RocCurveDisplay, precision_score, recall_score, f1_score)
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
import numpy as np

# Create PDF with fpdf
class PDF(FPDF):
    def header(self):
        self.set_font('Arial', 'B', 12)
        self.cell(0, 10, 'Classification Report with Metrics, Training, and Testing Time', 0, 1, 'C')

    def chapter_title(self, title):
        self.set_font('Arial', 'B', 12)
        self.cell(0, 10, title, 0, 1, 'L')
        self.ln(10)

    def chapter_body(self, body):
        self.set_font('Arial', '', 10)
        self.multi_cell(0, 5, body)
        self.ln()

    def add_image(self, image_path, title=''):
        if title:
            self.chapter_title(title)
        self.image(image_path, x=10, y=None, w=180)
        self.ln(10)

    def add_classification_report(self, report):
        self.chapter_title("Classification Report:")
        self.chapter_body(report)

# Function to train and evaluate a single model
def train_and_evaluate_model(name, model, X_train, X_test, y_train, y_test):
    pdf = PDF()
    output_folder = f"/root/resultados-ml/iot-23/{name}-60test"
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)
```

```
print(f"Start training {name}")
start_time = time.time()
if name in ["SVM", "KNN", "SGD", "MLP", "Nearest_Centroid"]:
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    print(f"Scaling done for {name}")
    if name == "SVM":
        # Define parameter grid for grid search
        param_grid = {
            'C': [100, 1000],
            'gamma': [1e-4, 1e-5],
            'kernel': ['rbf', 'sigmoid']
        }
        # Perform grid search
        print(f"Start grid search for {name}")
        grid_search = GridSearchCV(SVC(probability=True), param_grid, refit=True, verbose=2, cv=5, n_jobs=-1)
        grid_search.fit(X_train, y_train)
        print(f"End grid search for {name}")
        # Use the best model with the best parameters
        model = grid_search.best_estimator_
        params = grid_search.best_params_
        single_start_time = time.time()
        model.fit(X_train, y_train)
        single_train_time = time.time() - single_start_time
        pdf.chapter_body(f"Best model: {model} Best params: {params} \n")
        pdf.chapter_body(f"Best model Training time: {single_train_time:.4f} seconds\n")
    else:
        model.fit(X_train, y_train)
else:
    model.fit(X_train, y_train)
train_time = time.time() - start_time
print(f"End training {name}")
start_time = time.time()
print(f"Start prediction for {name}")
y_pred = model.predict(X_test)
test_time = time.time() - start_time

pdf.add_page()
pdf.chapter_title('Training and Testing Time')
pdf.chapter_body(f"Training time: {train_time:.4f} seconds\nTesting time: {test_time:.4f} seconds\n")
print(f"Creating reports for {name}")

report = classification_report(y_test, y_pred)
pdf.add_classification_report(report)
print(f"Getting scores for {name}")
precision_scores = precision_score(y_test, y_pred, average=None)
recall_scores = recall_score(y_test, y_pred, average=None)

pdf.chapter_body("Precision and Recall Scores by Class with 8 decimals:\n")
unique_classes = np.unique(np.concatenate([y_test, y_pred]))
for i, class_name in enumerate(unique_classes):
    precision = precision_scores[i]
    recall = recall_scores[i]
    pdf.chapter_body(f"Class '{class_name}':\n")
    pdf.chapter_body(f"  Precision: {precision:.8f}\n")
    pdf.chapter_body(f"  Recall: {recall:.8f}\n")

# Define the classes of interest
classes_of_interest = ["Scan", "benign", "DoS"]
print(f"Getting confusion matrix for {name}")
# Confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=classes_of_interest)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=classes_of_interest)
# Save the confusion matrix plot as an image file
```

```
cm_plot_path = os.path.join(output_folder, "confusion_matrix.png")
disp.plot()
plt.savefig(cm_plot_path)

# Add the confusion matrix plot to the PDF
pdf.add_image(cm_plot_path, title="Confusion Matrix Plot")
plt.show()
plt.close()

cm1 = confusion_matrix(y_test, y_pred, labels=classes_of_interest, normalize = 'true')
disp1 = ConfusionMatrixDisplay(confusion_matrix=cm1, display_labels=classes_of_interest)
# Save the confusion matrix plot as an image file
cm1_plot_path = os.path.join(output_folder, "confusion_matrix_normalized.png")
disp1.plot()
plt.savefig(cm1_plot_path)

# Add the confusion matrix plot to the PDF
pdf.add_image(cm1_plot_path, title="Normalized Confusion Matrix Plot")
plt.show()
plt.close()

if isinstance(model, DecisionTreeClassifier):
    print(f"Plotting tree for {name}")
    plt.figure(figsize=(25, 15))  # Adjust the size as needed
    # Plot the decision tree
    tree.plot_tree(model, feature_names=X.columns, filled=True, fontsize=8, proportion=True)

    # Save the decision tree plot as an image file
    tree_plot_path = os.path.join(output_folder, "decision_tree_default.png")
    plt.savefig(tree_plot_path)
    plt.close()
    # Add the decision tree plot to the PDF
    pdf.add_page()
    pdf.chapter_title('Decision Tree')
    pdf.add_image(tree_plot_path, title="Decision Tree Plot")

print(f"Start ROC plotting for {name}")
if name != "Nearest_Centroid":
    y_prob = model.predict_proba(X_test)
    label_binarizer = LabelBinarizer().fit(y_train)
    y_onehot_test = label_binarizer.transform(y_test)
    for class_of_interest in classes_of_interest:
        class_id = np.flatnonzero(label_binarizer.classes_ == class_of_interest)[0]
        fpr, tpr, _ = roc_curve(y_onehot_test[:, class_id], y_prob[:, class_id])
        display = RocCurveDisplay(fpr=fpr, tpr=tpr, estimator_name=f"{class_of_interest} vs the rest")
        display.plot(color="darkorange")
        plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title(f"One-vs-Rest ROC curve: {class_of_interest} vs (all other classes)")
        plt.legend(loc="lower right")
        plot_file = os.path.join(output_folder, f"roc_plot_{class_of_interest}.png")
        plt.savefig(plot_file)
        plt.show()
        plt.close()
        # Add the ROC curve plot to the PDF
        pdf.add_image(plot_file, title=f"ROC Curve: {class_of_interest} vs (all other classes)")
else:
    print(f"ROC for Nearest Centroid for {name}")
    centroids = model.centroids_
    distances = np.linalg.norm(X_test[:, np.newaxis] - centroids, axis=2)
    label_binarizer = LabelBinarizer().fit(y_train)
    y_onehot_test = label_binarizer.transform(y_test)
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
```

```
        for i, class_of_interest in enumerate(classes_of_interest):
            fpr[i], tpr[i], _ = roc_curve(y_onehot_test[:, i], -distances[:, i])
            roc_auc[i] = auc(fpr[i], tpr[i])
            plt.figure()
            display = RocCurveDisplay(fpr=fpr[i], tpr=tpr[i], estimator_name=f"{class_of_interest} vs the rest")
            display.plot(color="darkorange")
            plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
            plt.xlabel('False Positive Rate')
            plt.ylabel('True Positive Rate')
            plt.title(f"One-vs-Rest ROC curve: {class_of_interest} vs (all other classes)")
            plt.legend(loc="lower right")
            plot_file = os.path.join(output_folder, f"roc_plot_{class_of_interest}.png")
            plt.savefig(plot_file)
            plt.show()
            plt.close()
            # Add the ROC curve plot to the PDF
            pdf.add_image(plot_file, title=f"ROC Curve: {class_of_interest} vs (all other classes)")

    # Save PDF
    pdf_output_path = f"/root/resultados-ml/iot-23/iot23-{name}-60test-classification_report.pdf"
    pdf.output(pdf_output_path)
    print(f"PDF saved for {name}")
    return name, train_time, test_time, pdf_output_path

# Read CSV using Dask
ddf = dd.read_csv('/root/bbdd/logs-zeek/iot23-processed.csv')
# Split data into training and testing sets
X = ddf.drop(columns=['label', 'binary_label'])
y = ddf['label']
X_train, X_test, y_train, y_test = train_test_split(X.compute(), y.compute(), test_size=0.4, random_state=42)

# Define models
models = {
    "Decision_Tree": DecisionTreeClassifier(),
    "Nearest_Centroid": NearestCentroid(),
    "Random_Forest": RandomForestClassifier(n_estimators=100, random_state=0),
    "Gaussian_NB": GaussianNB(),
    "Bernoulli_NB": BernoulliNB(),
    "SGD": SGDClassifier(loss='log_loss', max_iter=1000, tol=1e-3),
    "Bagging_Tree": BaggingClassifier(estimator=DecisionTreeClassifier(), n_estimators=100, random_state=0),
    "AdaBoost_Tree": AdaBoostClassifier(estimator=DecisionTreeClassifier(), n_estimators=100, random_state=0),
    "MLP": MLPClassifier(max_iter=1000, random_state=42),
    "KNN": KNeighborsClassifier(),
    "SVM": SVC(probability=True)
}

# Train models and generate reports in parallel
with ThreadPoolExecutor(max_workers=1) as executor:
    futures = {executor.submit(train_and_evaluate_model, name, model, X_train, X_test, y_train, y_test): name for name, model in
models.items()}
    for future in as_completed(futures):
        name = futures[future]
        try:
            name, train_time, test_time, pdf_output_path = future.result()
            print(f"Completed {name}: Training time {train_time:.4f} seconds, Testing time {test_time:.4f} seconds, PDF saved at
{pdf_output_path}")
        except Exception as exc:
            print(f"Error occurred for model {name}: {exc}")
```

Anexo XII: *Script* para muestreo de clase DoS

Se presenta un *script* que realiza un muestreo previo de la clase DoS, para facilitar la clasificación y sea escalable.

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
import time
import pandas as pd

# Define the CSV file path
input_csv_path = "/root/bbdd/logs-zeek/cic-iot-2023-logs/labeled-csv_all.csv"
output_csv_path = '/root/bbdd/logs-zeek/cic-iot-2023-encoded-common-12gb.csv'

# Columns to drop
columns_to_drop = ['tunnel_parents', 'ts', 'uid', 'id.orig_h', 'id.resp_h', 'id.orig_p', 'id.resp_p', 'startTime']

# Clean the DataFrame
def clean_dataframe(df):
    # Replace commas in 'service' column
    df['service'] = df['service'].str.replace(',', '-')

    # List of numeric and string columns
    cols_num = ['duration', 'orig_bytes', 'resp_bytes', 'missed_bytes', 'orig_pkts', 'orig_ip_bytes', 'resp_pkts', 'resp_ip_bytes',
'orig_bytes_mean', 'resp_bytes_mean', 'orig_bytes_std','resp_bytes_std', 'orig_bytes_mean_nocero',
'resp_bytes_mean_nocero','orig_bytes_std_nocero', 'resp_bytes_std_nocero', 'orig_bytes_min','resp_bytes_min', 'orig_bytes_max',
'resp_bytes_max','orig_pkts_nocero', 'resp_pkts_nocero', 'orig_pkts_cero','resp_pkts_cero', 'time_mean', 'time_std', 'time_min',
'time_max','orig_time_mean', 'orig_time_std', 'orig_time_min', 'orig_time_max','resp_time_mean', 'resp_time_std',
'resp_time_min', 'resp_time_max']

    cols_str = ['proto', 'service']
    cols_dash = ['conn_state', 'local_orig', 'local_resp', 'history']

    # Clean numeric columns
    for col in cols_num:
        df[col] = df[col].fillna('0').replace(['-', '', '[]', '<NA>'], '0').astype('float64')
    # Clean string columns
    for col in cols_str:
        df[col] = df[col].fillna('unknown').replace(['-', '', '[]', '<NA>'], 'unknown').astype('object')
    for col in cols_dash:
        df[col] = df[col].fillna('-').replace(['', '[]', '<NA>'], '-').astype('object')
    # Replace label values
    df['label'] = df['label'].str.replace('Mirai', 'DoS').str.replace('Recon', 'Scan').str.replace('Scanning', 'Scan') \
        .str.replace('DDoS', 'DoS').str.replace('DictionaryBruteForce', 'BruteForce')

    return df

# Normalize local values
def normalize_local(value):
    if value in [True, 'True', 'T']:
        return 'True'
    elif value in [False, 'False', 'F']:
        return 'False'
```

```
    else:
        return value

# Load values for encoding
history_values = []
with open('history_values.txt', 'r') as f:
    for line in f:
        history_values.append(line.strip())

service_values = ['unknown', 'dns', 'http', 'ssl', 'ntp', 'gssapi-smb', 'dhcp', 'krb_tcp', 'xmpp', 'ldap_udp', 'geneve', 'radius', 'ssh', 'syslog',
'vxlan', 'mqtt', 'ayiya', 'ssl-quic', 'quic-ssl', 'ssl-http', 'irc']

conn_state_values = ["S0", "S1", "SF", "REJ", "S2", "S3", "RSTO", "RSTR",
                     "RSTOS0", "RSTRH", "SH", "SHR", "OTH", "-"]

local_values = ["True", "False"]
proto_values = ["tcp", "udp", "icmp", "unknown"]

# Fit encoders for known unique value columns
le_history = LabelEncoder()
le_history.fit(history_values)
le_service = LabelEncoder()
le_service.fit(service_values)
le_conn_state = LabelEncoder()
le_conn_state.fit(conn_state_values)
le_local_resp = LabelEncoder()
le_local_resp.fit(local_values)
le_local_orig = LabelEncoder()
le_local_orig.fit(local_values)
le_proto = LabelEncoder()
le_proto.fit(proto_values)

# Create a dictionary of encoders
encoders = {
    'conn_state': le_conn_state,
    'local_resp': le_local_resp,
    'local_orig': le_local_orig,
    'proto': le_proto,
    'service': le_service,
    'history': le_history
}

columns_to_encode = ['proto', 'service', 'history', 'conn_state', 'local_orig', 'local_resp']

# Encode columns
def encode_columns(df, columns_to_encode, encoders):
    for col in columns_to_encode:
        le = encoders[col]
        df[col] = le.transform(df[col])
    return df

# Read and process the CSV file in chunks
chunk_size = 50000
```

```
chunks = pd.read_csv(input_csv_path, dtype=str, chunksize=chunk_size)
# Write header to the output CSV
first_chunk = next(chunks)
first_chunk = first_chunk.drop(columns_to_drop, axis=1)
first_chunk = clean_dataframe(first_chunk)
first_chunk['local_orig'] = first_chunk['local_orig'].apply(normalize_local)
first_chunk['local_resp'] = first_chunk['local_resp'].apply(normalize_local)
first_chunk = encode_columns(first_chunk, columns_to_encode, encoders)
first_chunk.to_csv(output_csv_path, mode='w', index=False, header=True)

# Function to sample 30% of the DoS labeled rows
def sample_majority_class(df, label_col, majority_class, frac, random_state=None):
    majority_df = df[df[label_col] == majority_class]
    minority_df = df[df[label_col] != majority_class]
    sampled_majority_df = majority_df.sample(frac=frac, random_state=random_state)
    return pd.concat([sampled_majority_df, minority_df], ignore_index=True)

# Process and append remaining chunks
for chunk in chunks:
    chunk = chunk.drop(columns_to_drop, axis=1)
    chunk = clean_dataframe(chunk)
    chunk['local_orig'] = chunk['local_orig'].apply(normalize_local)
    chunk['local_resp'] = chunk['local_resp'].apply(normalize_local)
    chunk = encode_columns(chunk, columns_to_encode, encoders)
    sampled_chunk = sample_majority_class(chunk, label_col='label', majority_class='DoS', frac=0.3, random_state=42)
    sampled_chunk.to_csv(output_csv_path, mode='a', index=False, header=False)
    print(f"Chunk appended.")

print(f'DataFrame saved to {output_csv_path}')
```

Anexo XIII: *Script* de entrenamiento y evaluación con selección de atributos

```
# Create PDF with fpdf
class PDF(FPDF):
    def header(self):
        self.set_font('Arial', 'B', 12)
        self.cell(0, 10, 'Classification Report with Metrics, Training, and Testing Time', 0, 1, 'C')

    def chapter_title(self, title):
        self.set_font('Arial', 'B', 12)
        self.cell(0, 10, title, 0, 1, 'L')
        self.ln(10)

    def chapter_body(self, body):
        self.set_font('Arial', '', 10)
        self.multi_cell(0, 5, body)
        self.ln()

    def add_image(self, image_path, title=''):
        if title:
            self.chapter_title(title)
        self.image(image_path, x=10, y=None, w=180)
        self.ln(10)

    def add_classification_report(self, report):
        self.chapter_title("Classification Report:")
        self.chapter_body(report)

# Define the chunk size
chunk_size = 10000  # You can adjust this based on your system's memory capacity

# Initialize an empty list to store the sampled chunks
sampled_chunks = []
csv_path = '/root/bbdd/logs-zeek/cic-iot-2023-encoded-common-12gb.csv'
# Iterate over the chunks in the CSV file
# Function to sample the DoS labeled rows
def sample_majority_class(df, label_col, majority_class, frac, random_state=None):
    majority_df = df[df[label_col] == majority_class]
    minority_df = df[df[label_col] != majority_class]

    sampled_majority_df = majority_df.sample(frac=frac, random_state=random_state)

    return pd.concat([sampled_majority_df, minority_df], ignore_index=True)

for chunk in pd.read_csv(csv_path, chunksize=chunk_size):
    # Sample 50% of the chunk
    sampled_chunk = sample_majority_class(chunk, label_col='label', majority_class='DoS', frac=0.5, random_state=42)
    # Append the sampled chunk to the list
    sampled_chunks.append(sampled_chunk)
```

```
# Concatenate the sampled chunks into a single DataFrame
sampled_cic_df = pd.concat(sampled_chunks)

# Calculate the count of each label value
label_counts = sampled_cic_df['label'].value_counts()

# Print the label counts
print(label_counts)

# Define the chunk size
chunk_size = 10000  # You can adjust this based on your system's memory capacity

# Initialize an empty list to store the sampled chunks
sampled_chunks = []
csv_path = '/root/bbdd/logs-zeek/iot23-encoded-joint.csv'
# Iterate over the chunks in the CSV file
for chunk in pd.read_csv(csv_path, chunksize=chunk_size):
    # Sample 50% of the chunk
    sampled_chunk = chunk.sample(frac=0.5, random_state=42)
    # Append the sampled chunk to the list
    sampled_chunks.append(sampled_chunk)

# Concatenate the sampled chunks into a single DataFrame
sampled_iot23_df = pd.concat(sampled_chunks)

# Calculate the count of each label value
label_counts = sampled_iot23_df['label'].value_counts()

# Print the label counts
print(label_counts)

csv_path = '/root/bbdd/logs-zeek/encoded_iotd20_v2.csv'
iotd20_df = pd.read_csv(csv_path)
label_counts = iotd20_df['label'].value_counts()
# Print the label counts
print(label_counts)

# Make sure all DataFrames have the same columns, irrespective of order
columns = list(sampled_iot23_df.columns)  # assuming iotd20_df has all the columns you need

# Reorder columns of each DataFrame to match the order in 'columns'
sampled_iot23_df = sampled_iot23_df[columns]
iotd20_df = iotd20_df[columns]
sampled_cic_df = sampled_cic_df[columns]


# List of DataFrames to concatenate
dataframes = [iotd20_df, sampled_iot23_df, sampled_cic_df]
# Concatenate the DataFrames
# Lists to hold the train and test sets
X_train_list = []
X_test_list = []
y_train_list = []
```

```
y_test_list = []

# Split each dataframe individually
for df in dataframes:
    y = df['label'].values
    X = df.drop(columns=['label', 'binary-label']).values

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.6, random_state=42)

    X_train_list.append(X_train)
    X_test_list.append(X_test)
    y_train_list.append(y_train)
    y_test_list.append(y_test)
columns = sampled_iot23_df.drop(columns=['label', 'binary-label']).columns
# Concatenate the training and test sets from each dataframe
X_train = np.concatenate(X_train_list, axis=0)
X_test = np.concatenate(X_test_list, axis=0)
y_train = np.concatenate(y_train_list, axis=0)
y_test = np.concatenate(y_test_list, axis=0)
print("train and test sets ready")

del sampled_iot23_df, sampled_cic_df,X,y
del dataframes, X_train_list, X_test_list, y_train_list, y_test_list

import numpy as np
import pandas as pd
from sklearn.feature_selection import mutual_info_classif
from sklearn.feature_selection import SelectKBest
from sklearn.datasets import load_iris
# Compute the information gain for each feature
info_gain = mutual_info_classif(X_train, y_train)

# Create a DataFrame to display the information gain for each feature
feature_info_gain = pd.DataFrame({'Feature': columns, 'Information Gain': info_gain})
feature_info_gain = feature_info_gain.sort_values(by='Information Gain', ascending=False)

# Display the information gain for each feature
print("Information Gain for each feature:")
print(feature_info_gain)

# Select the top k features based on information gain
k = 15  # Number of top features to select
selector = SelectKBest(mutual_info_classif, k=k)
X_train_fs = selector.fit_transform(X_train, y_train)

# Get the selected feature names
selected_features = columns[selector.get_support()]

print(f"\nTop {k} features selected based on information gain:")
print(selected_features)

# Display the selected features
print("\nSelected features dataset:")
```

```
print(X_train[selected_features])

# Transform the test set using the same selector
X_test_fs = selector.transform(X_test)


# Function to train and evaluate a single model
def train_and_evaluate_model(name, model, X_train, X_test, y_train, y_test):
    pdf = PDF()
    output_folder = f"/root/resultados-ml/conjunto/{name}-60test-mix-train-multiclass-fs-presplit"
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    print(f"Start training {name}")
    start_time = time.time()
    if name in ["SGD", "MLP", "Nearest_Centroid"]:
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        print(f"Scaling done for {name}")
        model.fit(X_train, y_train) #quitar si descomento svc

    else:
        model.fit(X_train, y_train)
    train_time = time.time() - start_time
    print(f"End training {name}")
    start_time = time.time()
    print(f"Start prediction for {name}")
    y_pred = model.predict(X_test)
    test_time = time.time() - start_time

    pdf.add_page()
    pdf.chapter_title('Training and Testing Time')
    pdf.chapter_body(f"Training time: {train_time:.4f} seconds\nTesting time: {test_time:.4f} seconds\n")
    print(f"Creating reports for {name}")
    try:
        report = classification_report(y_test, y_pred)
        pdf.add_classification_report(report)
        print(f"Getting scores for {name}")
        precision_scores = precision_score(y_test, y_pred, average=None)
        recall_scores = recall_score(y_test, y_pred, average=None)

        pdf.chapter_body("Precision and Recall Scores by Class with 8 decimals:\n")
        unique_classes = np.unique(np.concatenate([y_test, y_pred]))
        for i, class_name in enumerate(unique_classes):
            precision = precision_scores[i]
            recall = recall_scores[i]
            pdf.chapter_body(f"Class '{class_name}':\n")
            pdf.chapter_body(f"  Precision: {precision:.8f}\n")
            pdf.chapter_body(f"  Recall: {recall:.8f}\n")
```

```
# Define the classes of interest
classes_of_interest = ["Scan", "benign", "DoS","BruteForce"]
print(f"Getting confusion matrix for {name}")
# Confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=classes_of_interest)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=classes_of_interest)
# Save the confusion matrix plot as an image file
cm_plot_path = os.path.join(output_folder, "confusion_matrix.png")
disp.plot()
plt.savefig(cm_plot_path)

# Add the confusion matrix plot to the PDF
pdf.add_image(cm_plot_path, title="Confusion Matrix Plot")
plt.show()
plt.close()

cm1 = confusion_matrix(y_test, y_pred, labels=classes_of_interest, normalize = 'true')
disp1 = ConfusionMatrixDisplay(confusion_matrix=cm1, display_labels=classes_of_interest)
# Save the confusion matrix plot as an image file
cm1_plot_path = os.path.join(output_folder, "confusion_matrix_normalized.png")
disp1.plot()
plt.savefig(cm1_plot_path)

# Add the confusion matrix plot to the PDF
pdf.add_image(cm1_plot_path, title="Normalized Confusion Matrix Plot")
plt.show()
plt.close()

if isinstance(model, DecisionTreeClassifier):
    print(f"Plotting tree for {name}")
    plt.figure(figsize=(25, 15))  # Adjust the size as needed
    # Plot the decision tree
    tree.plot_tree(model, feature_names=columns, filled=True, fontsize=8, proportion=True)

    # Save the decision tree plot as an image file
    tree_plot_path = os.path.join(output_folder, "decision_tree_default.png")
    plt.savefig(tree_plot_path)
    plt.close()
    # Add the decision tree plot to the PDF
    pdf.add_page()
    pdf.chapter_title('Decision Tree')
    pdf.add_image(tree_plot_path, title="Decision Tree Plot")

print(f"Start ROC plotting for {name}")
if name != "Nearest_Centroid":
    y_prob = model.predict_proba(X_test)

    label_binarizer = LabelBinarizer().fit(y_train)
    y_onehot_test = label_binarizer.transform(y_test)
    for class_of_interest in classes_of_interest:
        class_id = np.flatnonzero(label_binarizer.classes_ == class_of_interest)[0]
        fpr, tpr, _ = roc_curve(y_onehot_test[:, class_id], y_prob[:, class_id])
        display = RocCurveDisplay(fpr=fpr, tpr=tpr, estimator_name=f"{class_of_interest} vs the rest")
```

```
            display.plot(color="darkorange")
            plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
            plt.xlabel('False Positive Rate')
            plt.ylabel('True Positive Rate')
            plt.title(f"One-vs-Rest ROC curve: {class_of_interest} vs (all other classes)")
            plt.legend(loc="lower right")
            plot_file = os.path.join(output_folder, f"roc_plot_{class_of_interest}.png")
            plt.savefig(plot_file)
            plt.show()
            plt.close()
            # Add the ROC curve plot to the PDF
            pdf.add_image(plot_file, title=f"ROC Curve: {class_of_interest} vs (all other classes)")
        else:
            print(f"ROC for Nearest Centroid for {name}")
            centroids = model.centroids_
            del X_train # OJO BORRAR
            distances = np.linalg.norm(X_test[:, np.newaxis] - centroids, axis=2)
            label_binarizer = LabelBinarizer().fit(y_train)
            y_onehot_test = label_binarizer.transform(y_test)
            fpr = dict()
            tpr = dict()
            roc_auc = dict()

            for i, class_of_interest in enumerate(classes_of_interest):
                fpr[i], tpr[i], _ = roc_curve(y_onehot_test[:, i], -distances[:, i])
                roc_auc[i] = auc(fpr[i], tpr[i])
                plt.figure()
                display = RocCurveDisplay(fpr=fpr[i], tpr=tpr[i], estimator_name=f"{class_of_interest} vs the rest")
                display.plot(color="darkorange")
                plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
                plt.xlabel('False Positive Rate')
                plt.ylabel('True Positive Rate')
                plt.title(f"One-vs-Rest ROC curve: {class_of_interest} vs (all other classes)")
                plt.legend(loc="lower right")
                plot_file = os.path.join(output_folder, f"roc_plot_{class_of_interest}.png")
                plt.savefig(plot_file)
                plt.show()
                plt.close()
                # Add the ROC curve plot to the PDF
                pdf.add_image(plot_file, title=f"ROC Curve: {class_of_interest} vs (all other classes)")
    except Exception as e:
        print(f"Error occurred for model {name}: {e}")
        pdf.add_page()
        pdf.chapter_title('Error')
        pdf.chapter_body(f"An error occurred during the training or evaluation of the model {name}:\n{str(e)}")

    finally:
        # Save PDF
        pdf_output_path = f"/root/resultados-ml/conjunto/conjunto-{name}-60test-mix-train-multiclass-fs-presplit-
classification_report.pdf"
        pdf.output(pdf_output_path)
        print(f"PDF saved for {name}")
        return name, train_time, test_time, pdf_output_path
```

```
# Define models
models = {
    "Decision_Tree": DecisionTreeClassifier(),
    "Random_Forest": RandomForestClassifier(n_estimators=100, random_state=0),
    "Gaussian_NB": GaussianNB(),
    "Bernoulli_NB": BernoulliNB(),
    "SGD": SGDClassifier(loss='log_loss', max_iter=1000, tol=1e-3),
    "Bagging_Tree": BaggingClassifier(estimator=DecisionTreeClassifier(), n_estimators=100, random_state=0),
    "AdaBoost_Tree": AdaBoostClassifier(estimator=DecisionTreeClassifier(), n_estimators=100, random_state=0),
    "MLP": MLPClassifier(max_iter=1000, random_state=42),
    "Nearest_Centroid": NearestCentroid()
    #"KNN": KNeighborsClassifier(),
    #"SVM": SVC(probability=True)
}

# Train models and generate reports in parallel
with ThreadPoolExecutor(max_workers=1) as executor:
    futures = {executor.submit(train_and_evaluate_model, name, model, X_train, X_test, y_train, y_test): name for name, model in
models.items()}
    for future in as_completed(futures):
        name = futures[future]
        try:
            name, train_time, test_time, pdf_output_path = future.result()
            print(f"Completed {name}: Training time {train_time:.4f} seconds, Testing time {test_time:.4f} seconds, PDF saved at
{pdf_output_path}")
        except Exception as exc:
            print(f"Error occurred for model {name}: {exc}")
```

Anexo XIV: Tablas de Precision y Recall y Matrices de Confusión

Precision										
Dataset name	Label	*DecissionTree*	Gaussian NB	BernouilliNB	SGD	RandomForest	BaggingTree	BoostingTree	NearestCentroid	MLP
IoTD20	*BruteForce*	0,99	0,00	0,08	1	1	0,99	0,98	0,03	1
	DoS	0,99	0,72	0,95	0,96	0,99	0,99	0,99	0,94	0,99
	Scan	1	0,3	1	0,96	1	1	1	0,5	1
	Benign	0,98	0,93	0,2	0,92	0,98	0,98	0,98	0,59	0,96
	Average	0.99	0.68	0.89	0.96	0.99	0.99	0.99	0.86	0.99
IoT-23	DoS	1	0,91	0,84	0,99	1	1	1	0,98	1
	Scan	1	1	1	1	1	1	1	0,96	1
	Benign	1	1	1	1	1	1	1	1	1
	Average	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	1.00
CIC-IoT-2023	*BruteForce*	0,83	0,00	0,00	0,00	0,96	0,91	0,85	0,00	0,66
	DoS	1	1	1	1	1	1	1	1	1
	Scan	0,98	0,00	0,03	0,69	0,99	0,99	0,99	0,51	0,96
	Benign	0,97	0,28	0,01	0,55	0,97	0,98	0,97	0,28	0,85
	Average	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

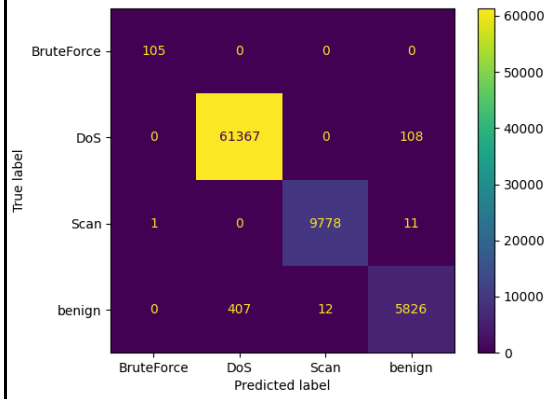
Precision										
Dataset name	Label	*DecissionTree*	Gaussian NB	BernouilliNB	SGD	RandomForest	BaggingTree	BoostingTree	NearestCentroid	MLP
Escenario 1 Binary	Benign	1	0,25	0,8	0,98	1	1	1	0,47	0,99
	Malign	1	1	0,95	1	1	1	1	0,99	1
	Average	1.00	0.82	0.91	1.00	1.00	1.00	1.00	0.87	1.00
Escenario 1 Multiclass	*BruteForce*	0.44	0.00	0.00	0.00	0.55	0.55	0.46	0.01	0.82
	DoS	1	1.00	0.99	0.99	1	1	1	1.00	1.00
	Scan	1	0.00	0.85	0.99	1	1	1	0.85	1.00
	Benign	1	0.05	0.82	0.98	1	1	1	0.99	0.99
	Average	1.00	0.51	0.91	0.99	1.00	1.00	1.00	0.96	1.00
Escenario 1 Multiclass & FS	*BruteForce*	0.42	0.00	0.00	0.00	0.53	0.51	0.43	0.01	0.04
	DoS	1	1.00	0.64	0.99	1	1	1	0.99	1
	Scan	1	0.00	0.95	0.88	1	1	1	0.88	0.99
	Benign	1	0.02	0.78	0.96	1	1	1	0.98	0.99
	Average	1.00	0.50	0.76	0.96	1.00	1.00	1.00	0.96	0.99
Escenario 2 Binary	Benign	0.96	0.47	0.21	0.11	1.00	0.99	0.98	0.73	0.11
	Malign	0.53	0.28	0.53	0.01	0.53	0.53	0.53	0.53	0.01
	Average	0.68	0.37	0.38	0.06	0.75	0.75	0.74	0.58	0.06
Escenario 2 Multiclass	*BruteForce*	0,00	0,00	0,03	0,00	0,00	0,00	0,02	0,02	0,07
	DoS	0,00	0,02	0,00	0,01	0,00	0,00	0,00	0,00	0,56
	Scan	0,06	0,07	0,11	0,17	0,14	0,04	0,19	0,88	1
	Benign	1	0,96	0,35	0,11	1	1	1	1	0,99
	Average	0.50	0.49	0.22	0.14	1.00	0.49	0.57	0.93	0.99

Recall										
Dataset name	Label	*DecissionTree*	Gaussian NB	BernouilliNB	SGD	RandomForest	BaggingTree	BoostingTree	NearestCentroid	MLP
IoTD20	*BruteForce*	1	1	1	0,27	1	1	1	1	0,99
	DoS	1	0,48	0,86	1	1	1	1	0,83	1
	Scan	1	0,00	0,95	0,99	1	1	1	0,89	1
	Benign	0,93	0,05	0,36	0,57	0,94	0,93	0,93	0,29	0,89
	Average	0.99	0.38	0.83	0.96	0.99	0.99	0.99	0.79	0.99
IoT-23	DoS	1	0,77	0,77	0,99	1	1	1	0,77	1
	Scan	1	1	1	1	1	1	1	1	1
	Benign	1	1	1	1	1	1	1	0,96	1
	Average	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	1.00
CIC-IoT-2023	*BruteForce*	0,86	0,99	0,14	0,00	0,73	0,86	0,85	0,44	0,36
	DoS	1	0,12	0,89	1	1	1	1	0,99	1
	Scan	0,98	0,00	0,47	0,52	0,98	0,99	0,99	0,81	0,96
	Benign	0,98	0,01	0,74	0,33	0,98	0,99	0,98	0,53	0,90
	Average	1.00	0.12	0.89	1.00	1.00	1.00	1.00	0.99	1.00

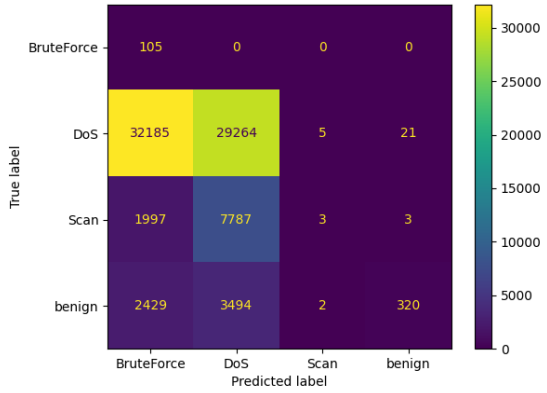
Recall										
Dataset name	Label	*DecissionTree*	Gaussian NB	BernouilliNB	SGD	RandomForest	BaggingTree	BoostingTree	NearestCentroid	MLP
Escenario 1 Binario	Benign	1	1	0,84	0,99	1	1	1.00	0,98	0.99
	Malign	1	0,08	0,93	0,99	1	1	1.00	0,65	1
	Average	1.00	0.30	0.91	0.99	1.00	1.00	1.00	0.73	1.00
Escenario 1 Multiclass	*BruteForce*	0.38	0.99	0.79	0.00	0.36	0.37	0.38	0.67	0.18
	DoS	1.00	0.12	0.76	1.00	1.00	1.00	1.00	0.97	1.00
	Scan	1.00	0.00	0.99	0.99	1.00	1.00	1.00	0.99	0.99
	Benign	1.00	0.00	0.83	0.99	1.00	1.00	1.00	0.84	1.00
	Average	1.00	0.06	0.84	0.99	1.00	1.00	1.00	0.94	1.00
Escenario 1 Multiclass & FS	*BruteForce*	0.36	1.00	0.66	0.00	0.35	0.35	0.35	0.09	0.00
	DoS	1.00	0.12	0.85	0.99	1.00	1.00	1.00	0.99	1.00
	Scan	1.00	0.00	0.99	0.98	1.00	1.00	1.00	0.99	0.99
	Benign	0.99	0.00	0.00	0.85	0.99	0.99	0.99	0.85	0.99
	Average	1.00	0.06	0.68	1.00	1.00	1.00	1.00	0.96	0.99
Escenario 2 DoS sampled Binary	Benign	0.00	1.00	0.00	0.14	0.00	0.00	0.00	0.00	0.14
	Malign	1	0.00	1	0.01	1	1	1	1	0.01
	Average	0.68	0.47	0.53	0.07	0.53	0.53	0.74	0.53	0.99
Escenario 2 DoS sampled Rest for *test*ing	*BruteForce*	0,00	1	0,99	0,00	0,00	0,01	0,00	0,99	0,48
	DoS	0,5	0,00	0,12	0,71	0,67	0,49	0,00	0,49	0,95
	Scan	0,00	0,00	0,00	0,00	0,00	0,00	0,04	1	1
	Benign	0,14	0,00	0,00	0,14	0,14	0,14	0,14	0,00	0,98
	Average	0.07	0.00	0.00	0.07	0.07	0.07	0.07	0.53	0.99

Matrices de Confusión IoTD20

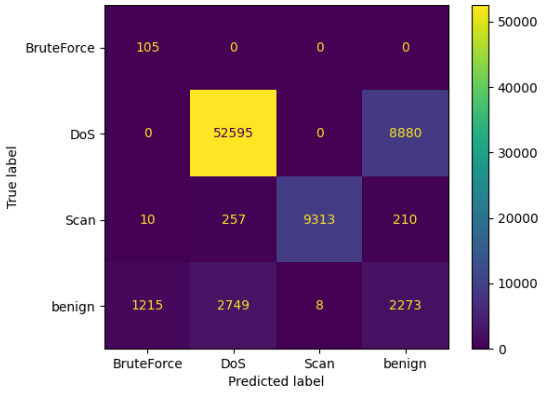
DecisionTree



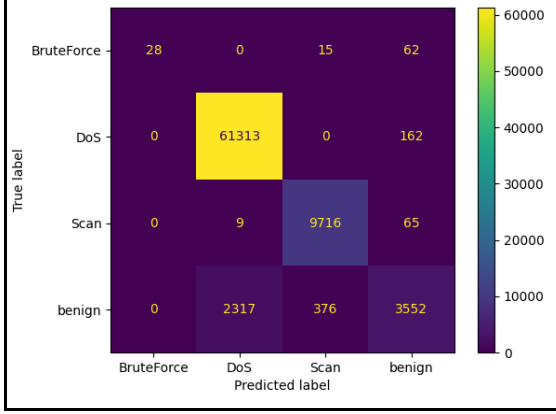
Gaussian NB



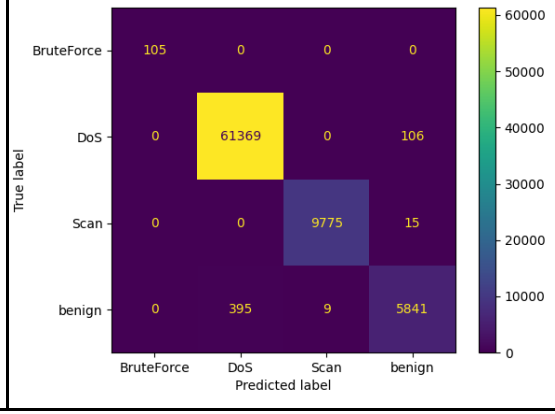
BernouilliNB

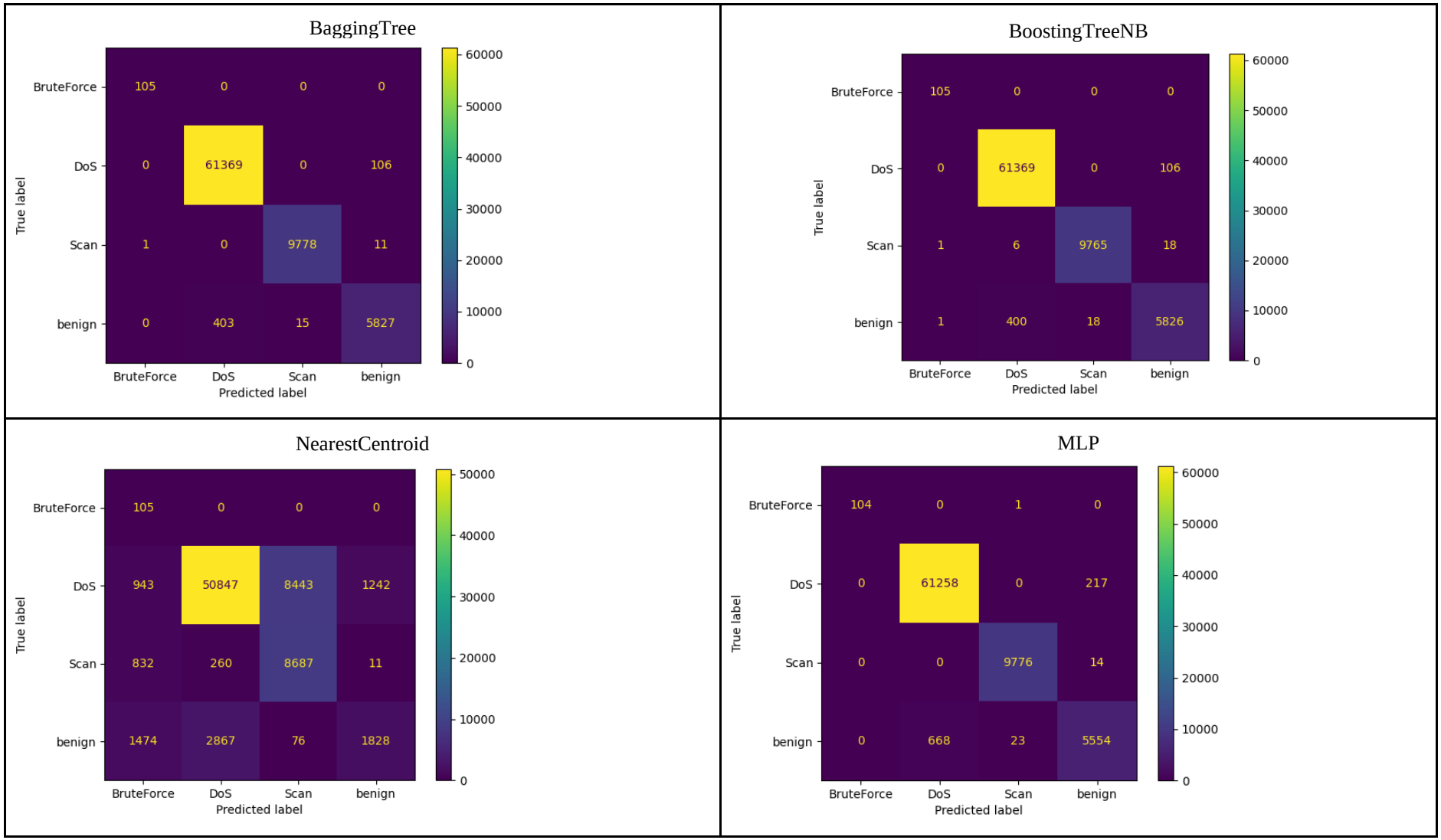


SGD



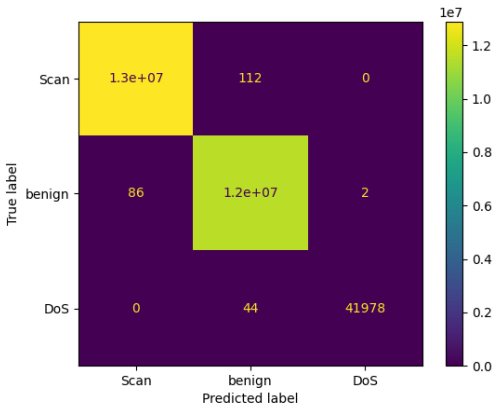
RandomForest



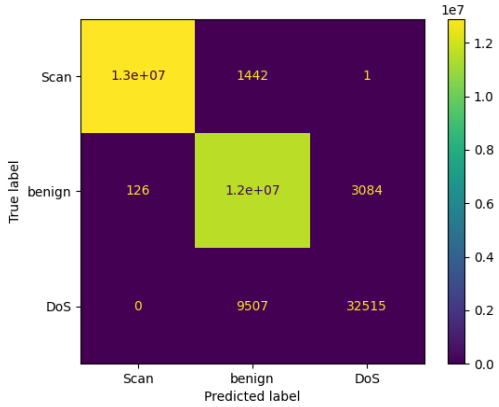

BaggingTree
BruteForce
DoS
Scan
benign
True label
Predicted label
105 0 0 0
0 61369 0 106
1 0 9778 11
0 403 15 5827
60000
50000
40000
30000
20000
10000
0
BoostingTreeNB
105 0 0 0
0 61369 0 106
1 6 9765 18
1 400 18 5826
NearestCentroid
105 0 0 0
943 50847 8443 1242
832 260 8687 11
1474 2867 76 1828
50000
40000
30000
20000
10000
0
MLP
104 0 1 0
0 61258 0 217
0 0 9776 14
0 668 23 5554

Matrices de Confusión IoT-23

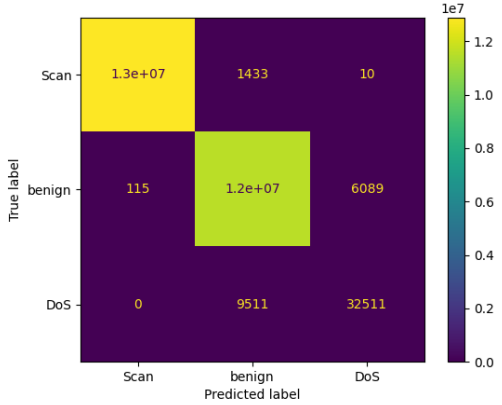
DecisionTree



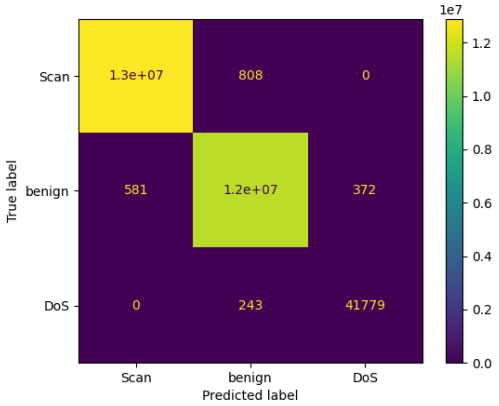
Gaussian NB



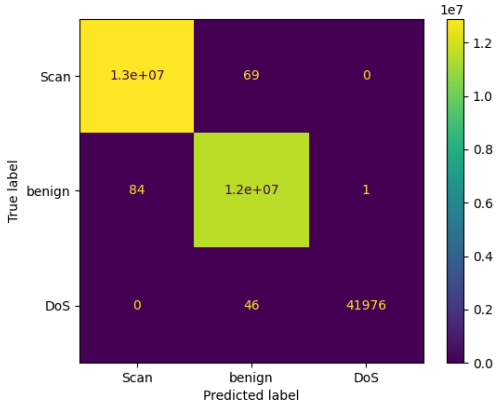
BernouilliNB



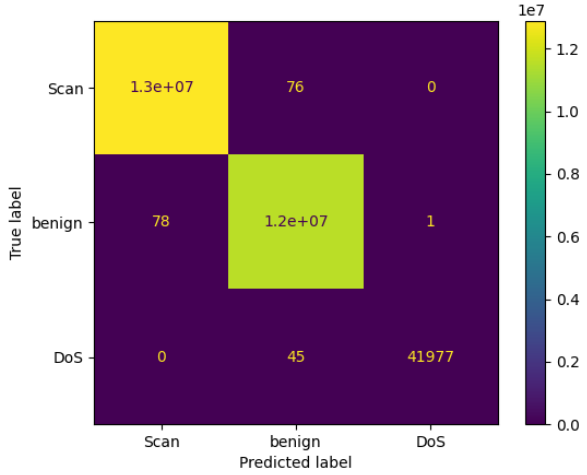
SGD



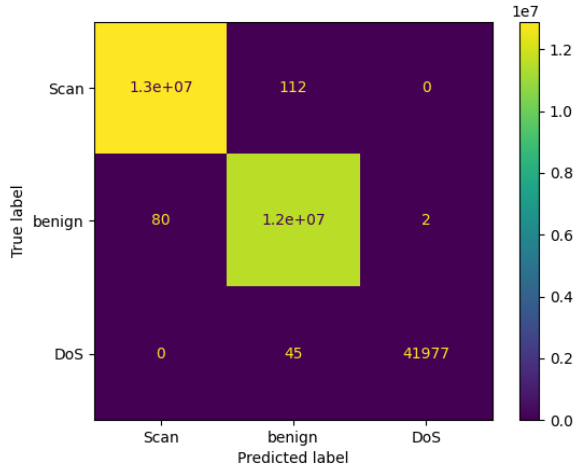
RandomForest



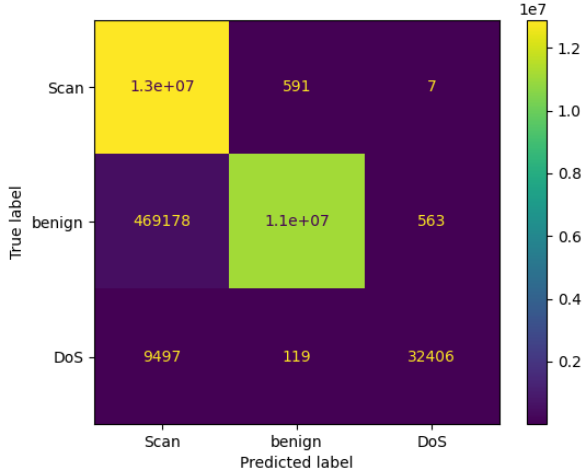
BaggingTree



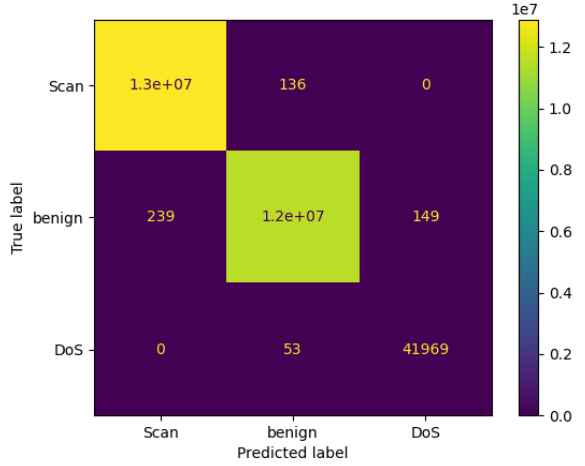
BoostingTreeNB



NearestCentroid

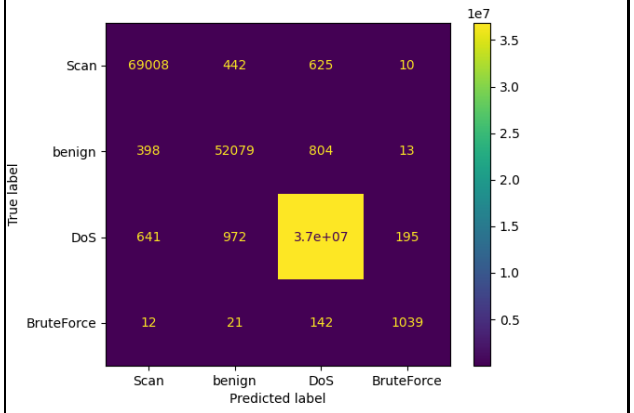


MLP



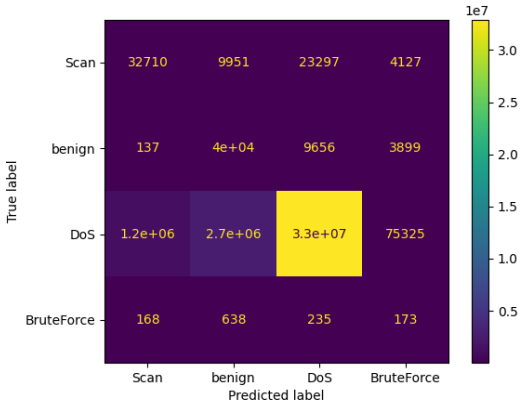
Matrices de Confusión CIC-IoT-2023

DecisionTree

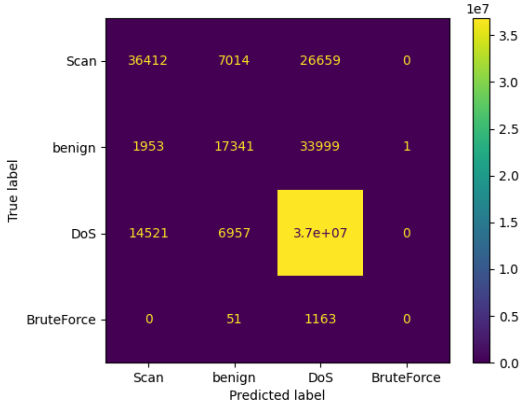


Gaussian NB

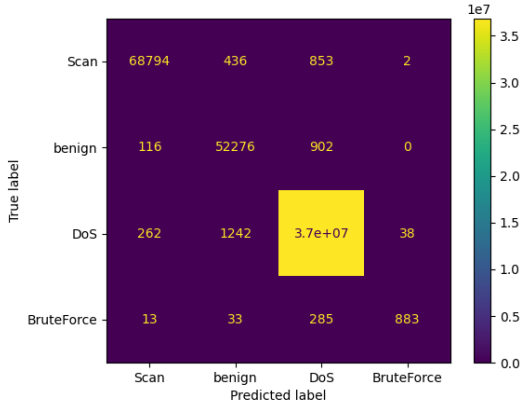
BernouilliNB



SGD

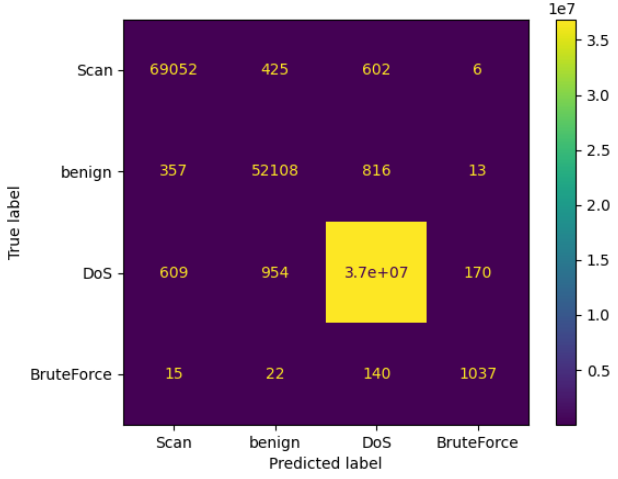


RandomForest

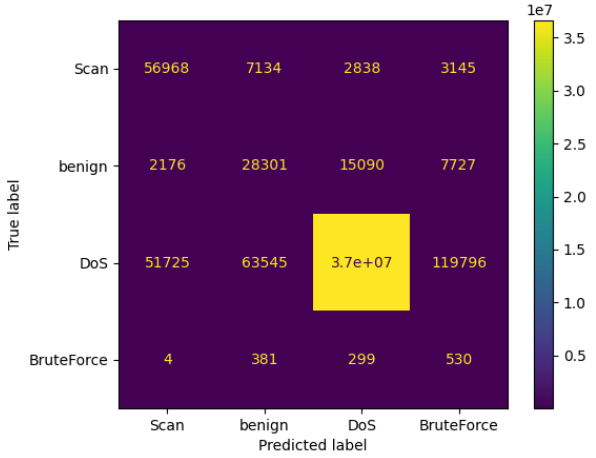


BaggingTree

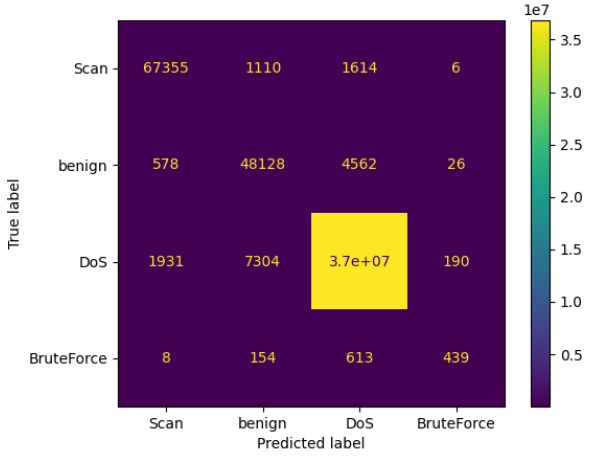
BoostingTreeNB



NearestCentroid

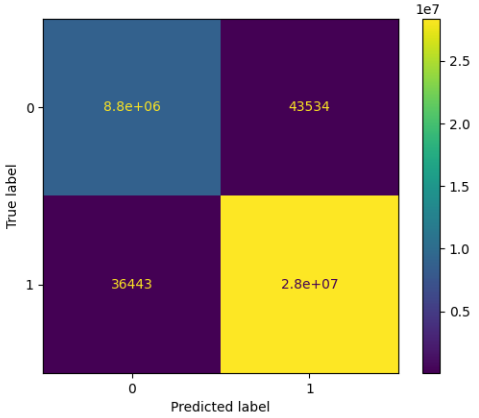


MLP

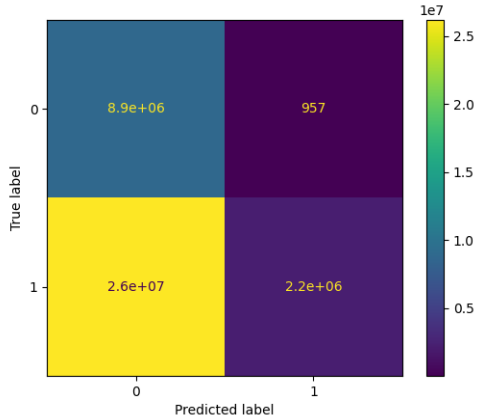


Matrices de Confusión Escenario 1, binary

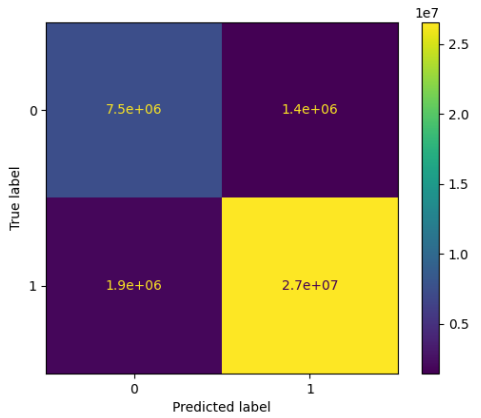
DecisionTree



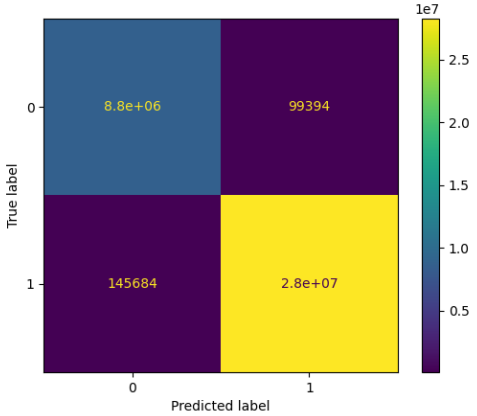
Gaussian NB



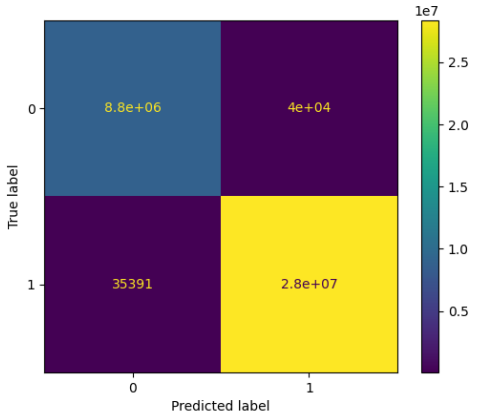
BernouilliNB



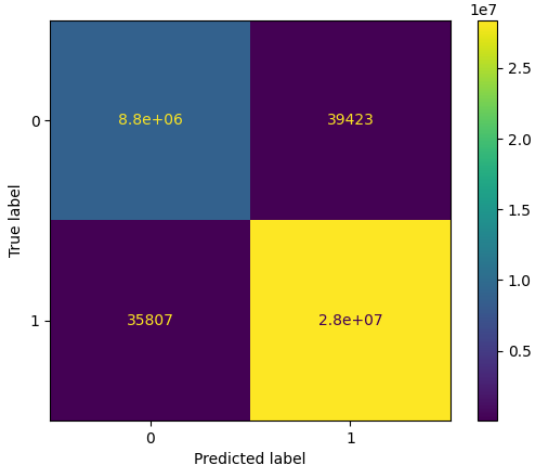
SGD



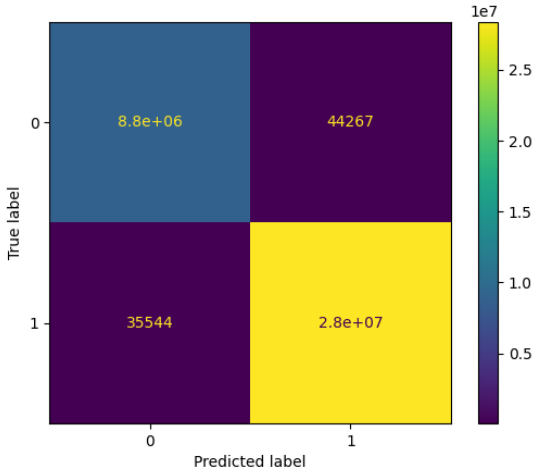
RandomForest



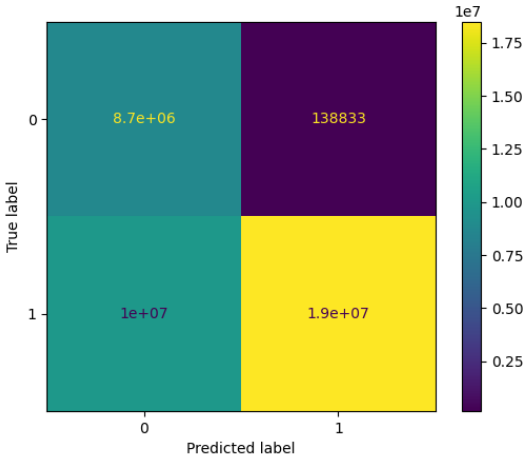
BaggingTree



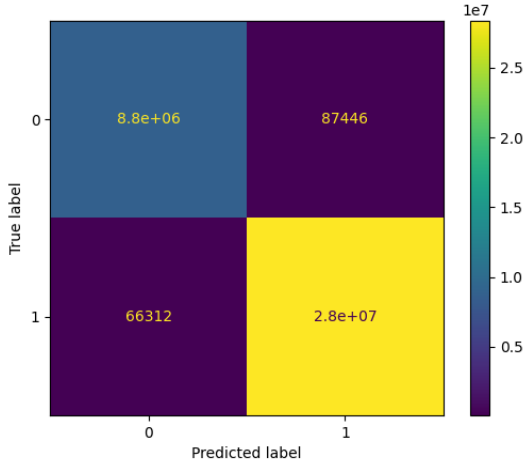
BoostingTreeNB



NearestCentroid

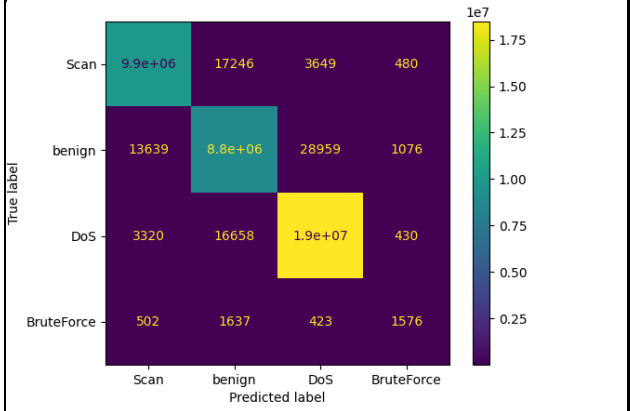


MLP

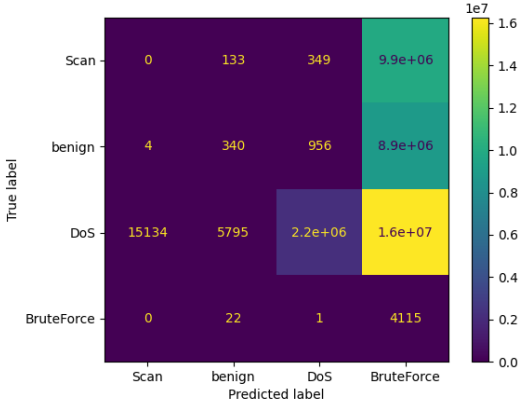


Matrices de Confusión Escenario 1, multiclass

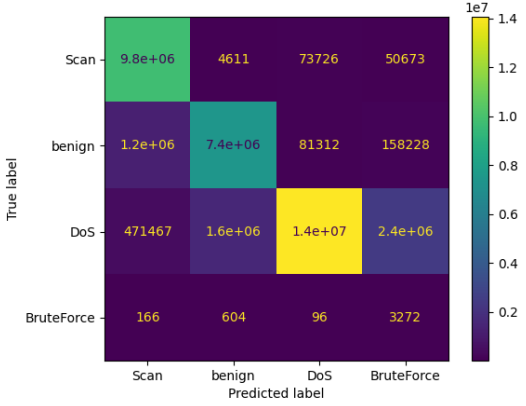
DecisionTree



Gaussian NB

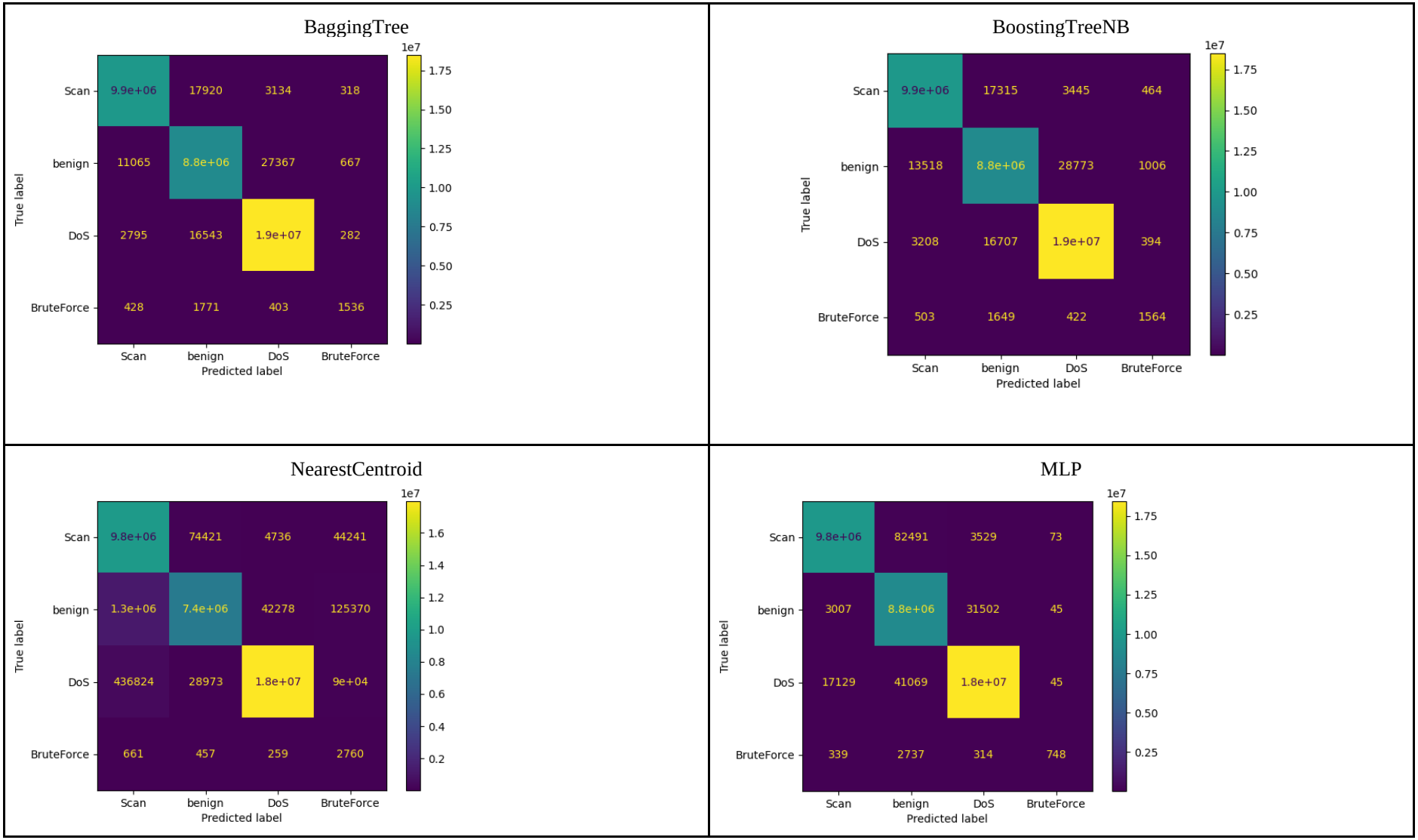


BernouilliNB



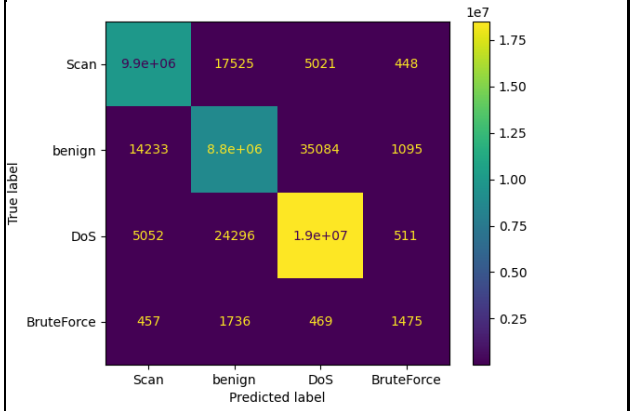
SGD

RandomForest

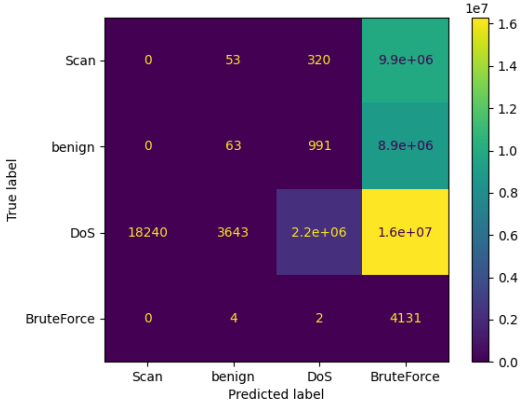

BaggingTree
1e7
Scan
benign
DoS
BruteForce
True label
9.9e+06
17920
3134
318
11065
8.8e+06
27367
667
2795
16543
1.9e+07
282
428
1771
403
1536
Scan
benign
DoS
BruteForce
Predicted label
1.75
1.50
1.25
1.00
0.75
0.50
0.25
BoostingTreeNB
1e7
Scan
benign
DoS
BruteForce
True label
9.9e+06
17315
3445
464
13518
8.8e+06
28773
1006
3208
16707
1.9e+07
394
503
1649
422
1564
Scan
benign
DoS
BruteForce
Predicted label
1.75
1.50
1.25
1.00
0.75
0.50
0.25
NearestCentroid
1e7
Scan
benign
DoS
BruteForce
True label
9.8e+06
74421
4736
44241
1.3e+06
7.4e+06
42278
125370
436824
28973
1.8e+07
9e+04
661
457
259
2760
Scan
benign
DoS
BruteForce
Predicted label
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
MLP
1e7
Scan
benign
DoS
BruteForce
True label
9.8e+06
82491
3529
73
3007
8.8e+06
31502
45
17129
41069
1.8e+07
45
339
2737
314
748
Scan
benign
DoS
BruteForce
Predicted label
1.75
1.50
1.25
1.00
0.75
0.50
0.25

Matrices de Confusión Escenario 1, multiclass feature selection

DecisionTree

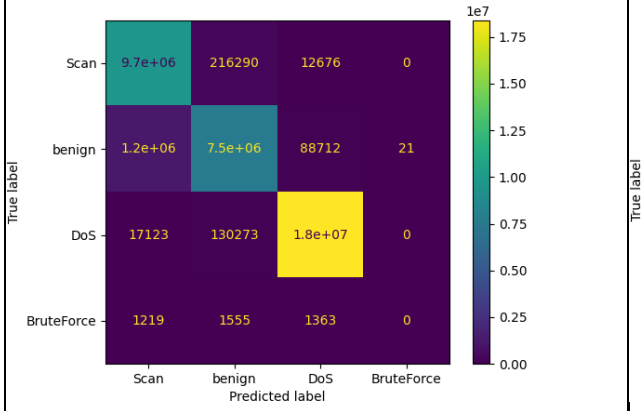


Gaussian NB

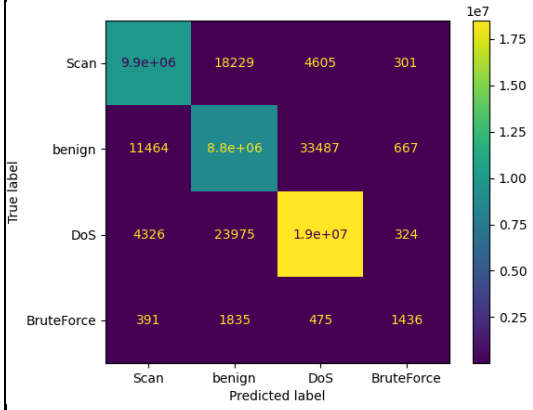


BernouilliNB

SGD



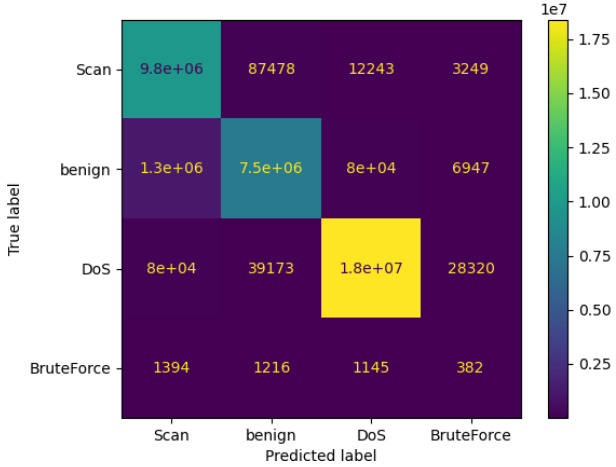
RandomForest



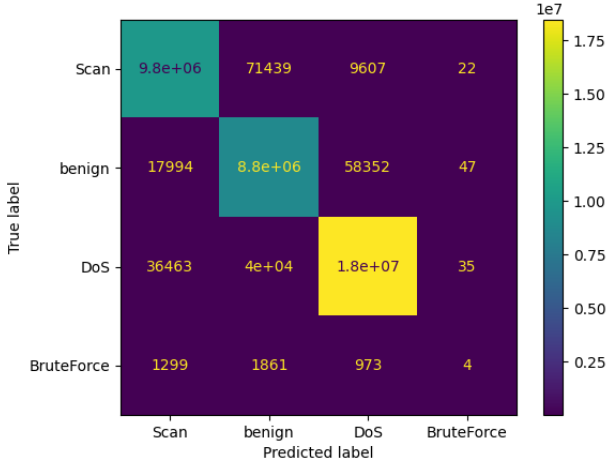
BaggingTree

BoostingTreeNB

NearestCentroid

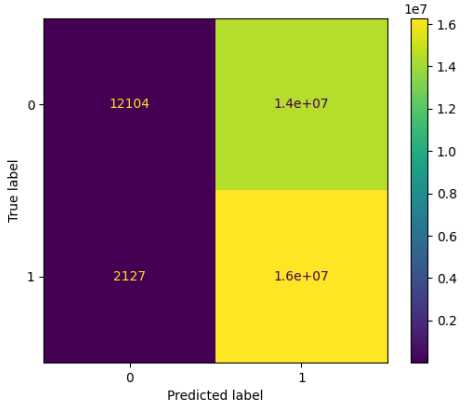


MLP

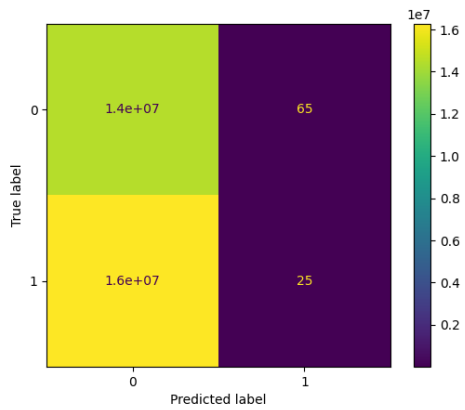


Matrices de Confusión Escenario 2 Binary

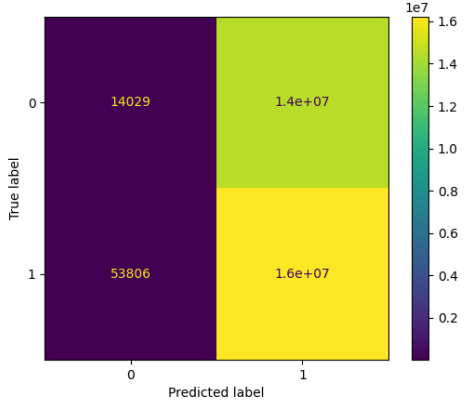
DecisionTree



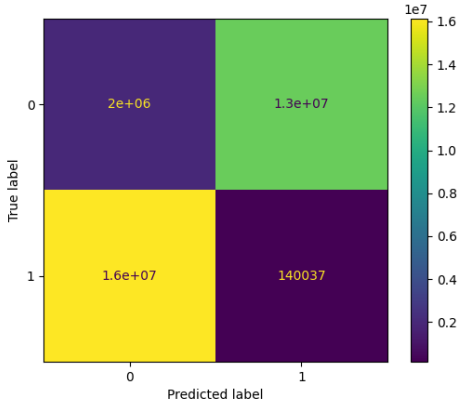
Gaussian NB



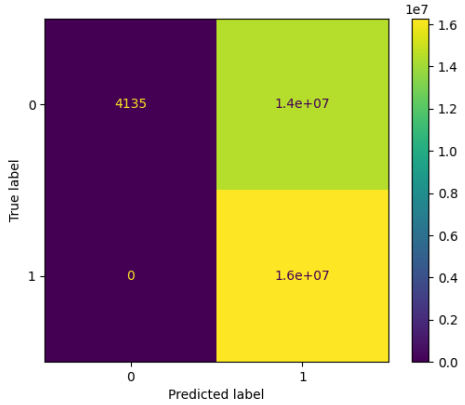
BernouilliNB

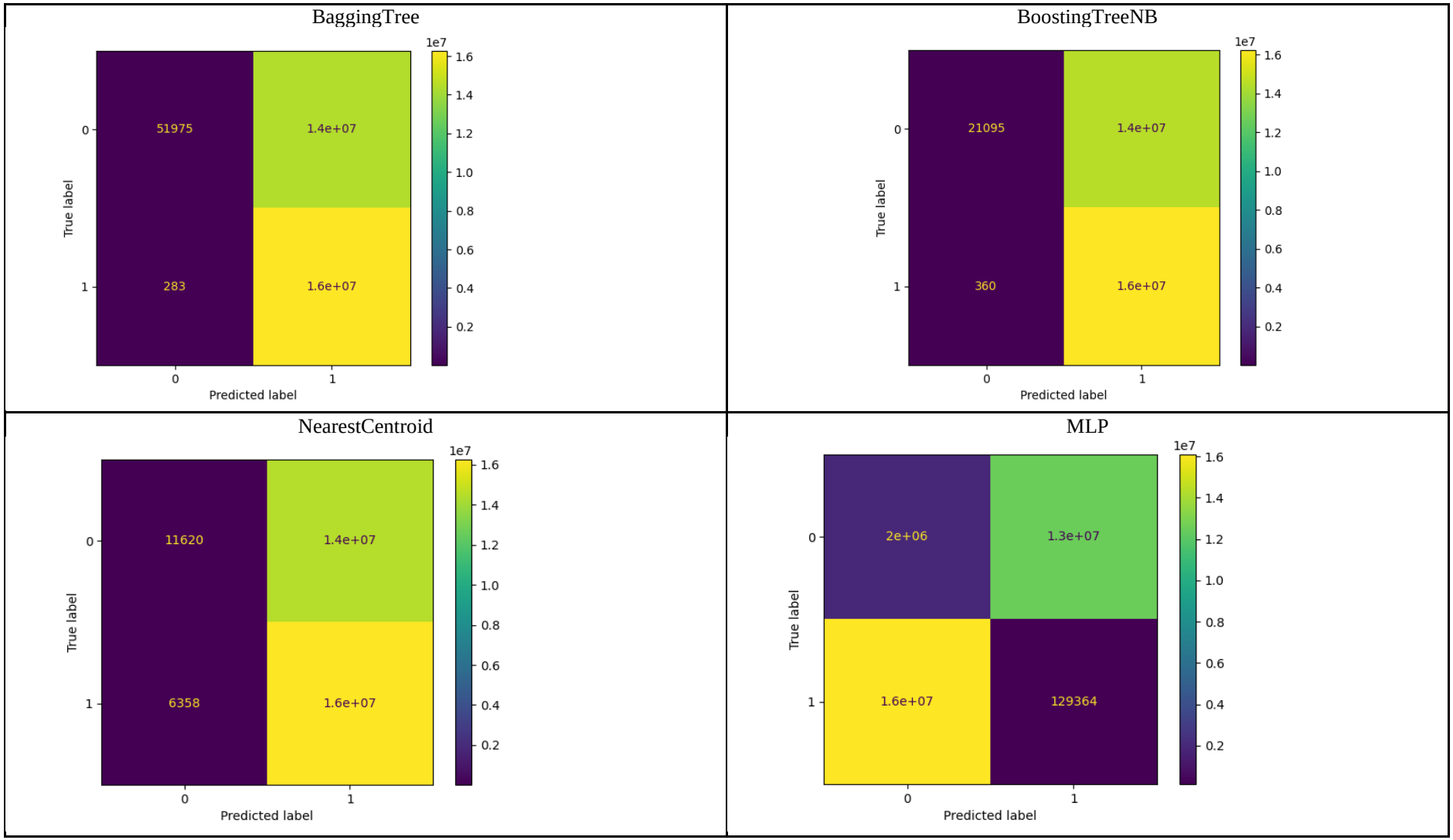


SGD



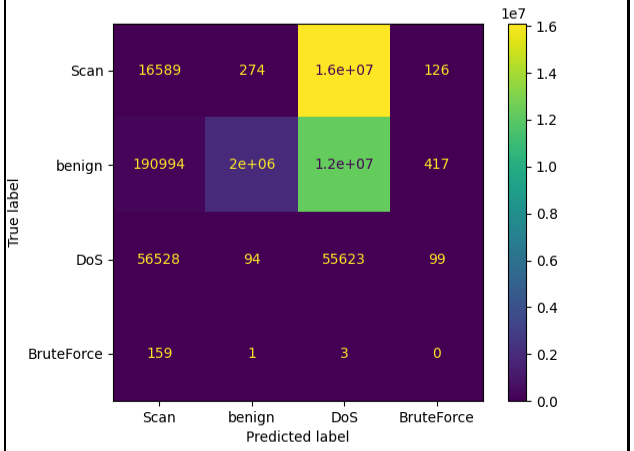
RandomForest



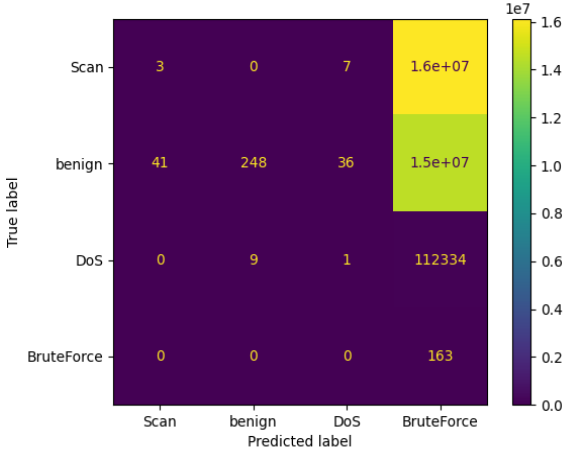

BaggingTree
51975
1.4e+07
283
1.6e+07
True label
Predicted label
1e7
BoostingTreeNB
21095
1.4e+07
360
1.6e+07
NearestCentroid
11620
1.4e+07
6358
1.6e+07
MLP
2e+06
1.3e+07
1.6e+07
129364

Matrices de Confusión Escenario 2 Multiclass

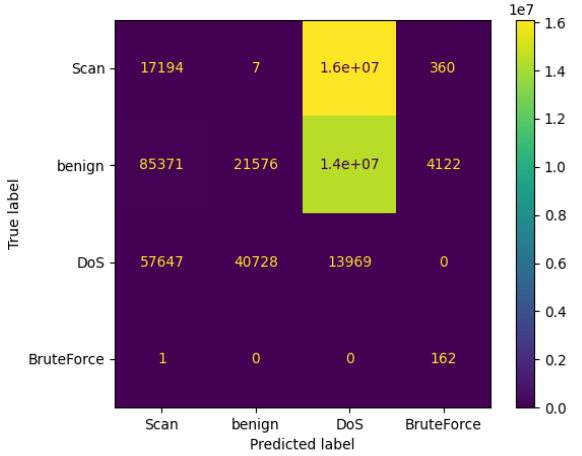
DecisionTree



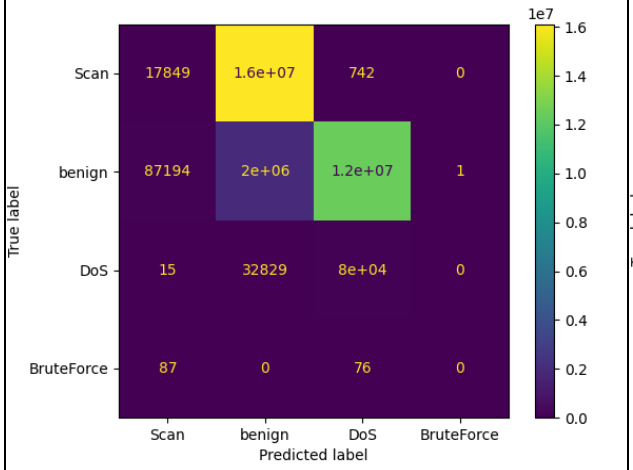
Gaussian NB



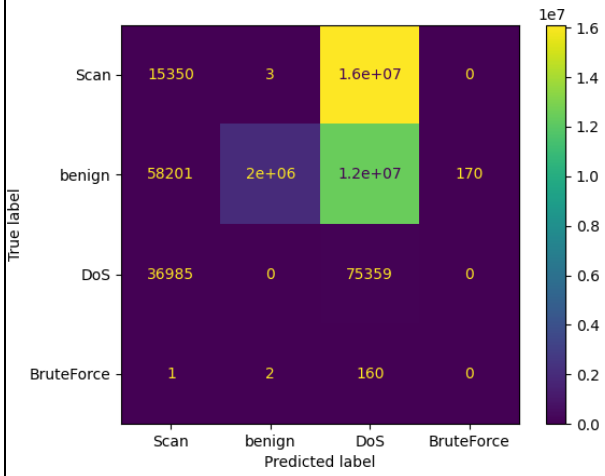
BernouilliNB

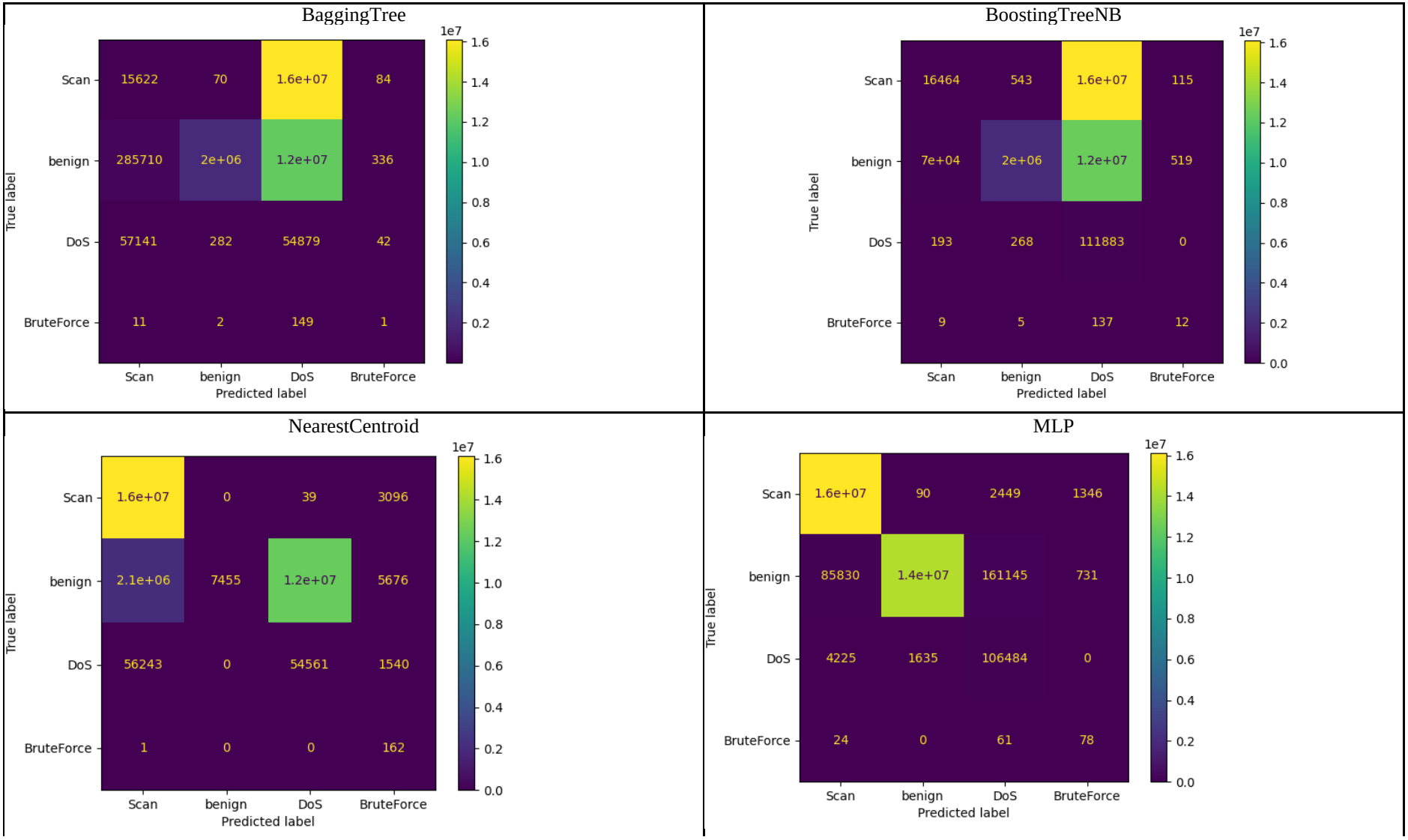


SGD

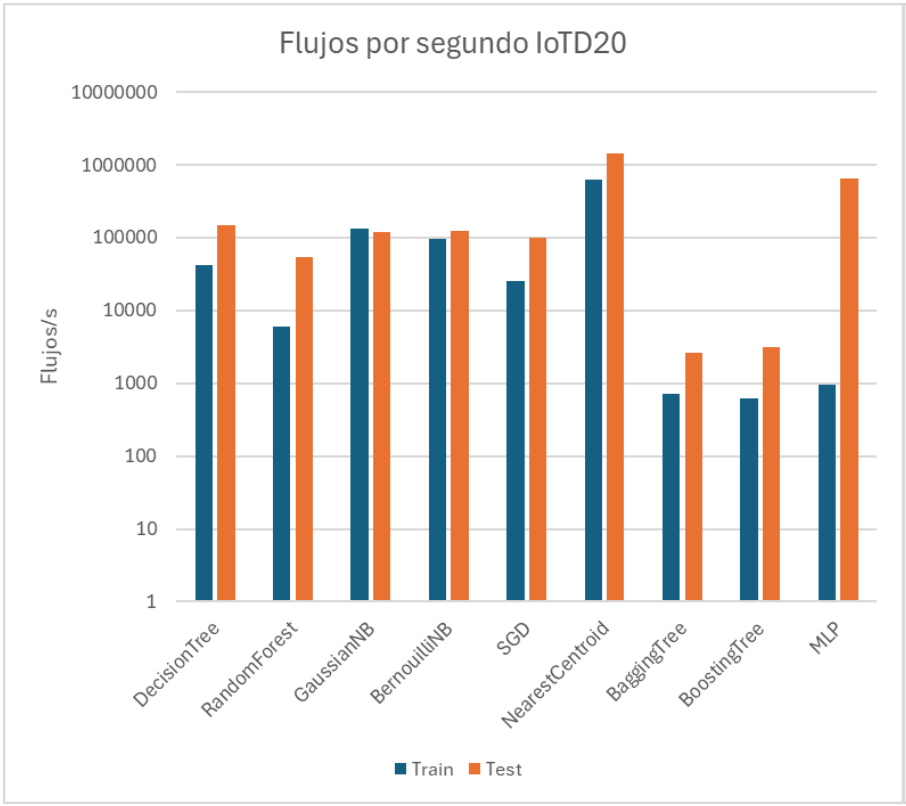


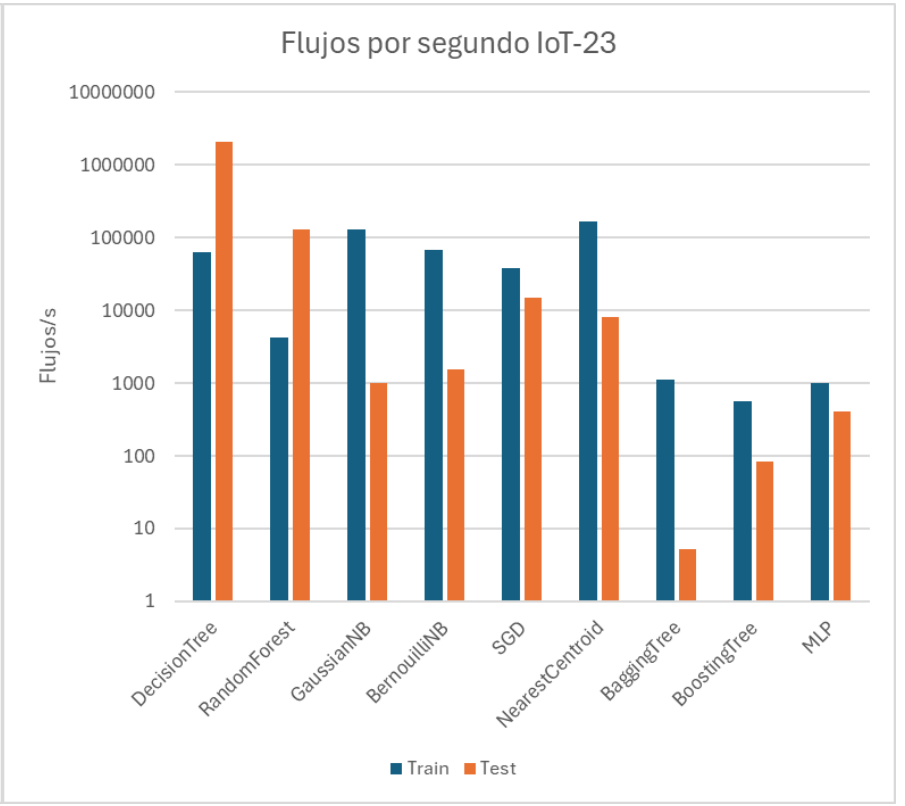
RandomForest

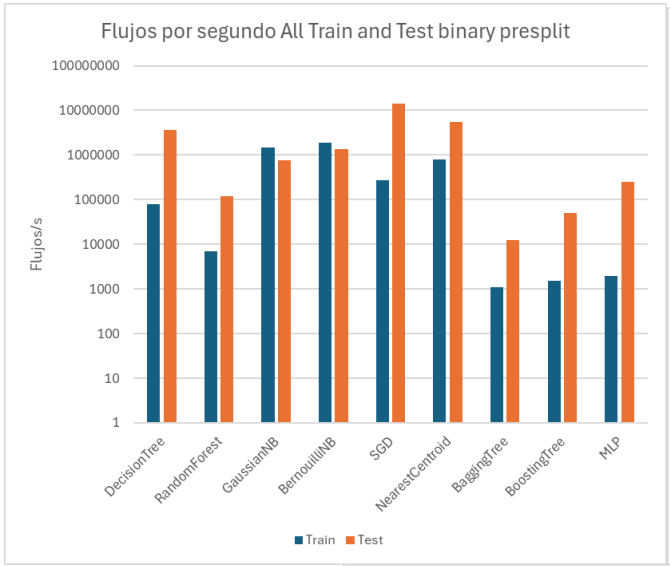


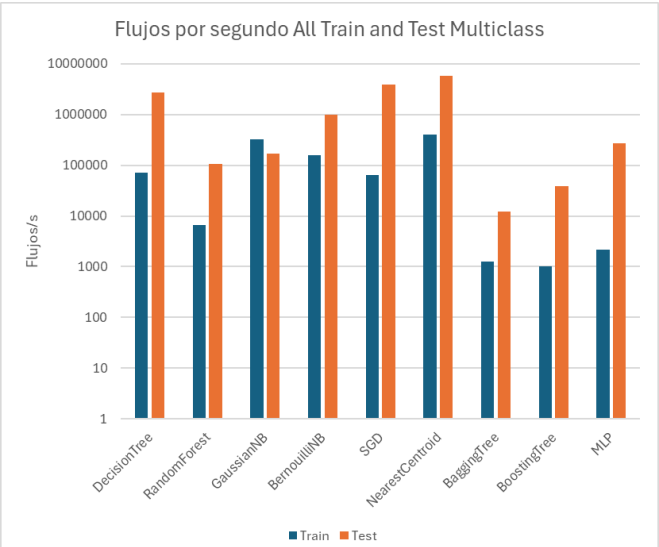

BaggingTree
Scan
benign
DoS
BruteForce
True label
Predicted label
15622
70
1.6e+07
84
285710
2e+06
1.2e+07
336
57141
282
54879
42
11
2
149
1
1e7
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
BoostingTreeNB
16464
543
1.6e+07
115
7e+04
2e+06
1.2e+07
519
193
268
111883
0
9
5
137
12
0.0
NearestCentroid
1.6e+07
0
39
3096
2.1e+06
7455
1.2e+07
5676
56243
0
54561
1540
1
0
0
162
MLP
1.6e+07
90
2449
1346
85830
1.4e+07
161145
731
4225
1635
106484
0
24
0
61
78

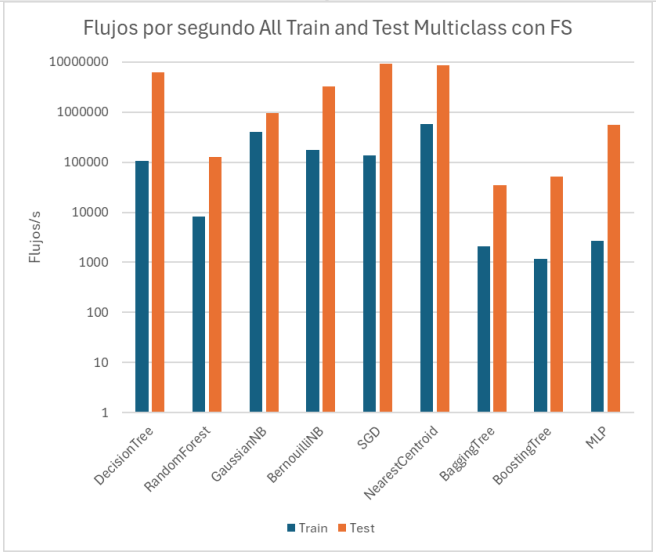
Anexo XV: Cálculo Tiempos Selección de atributos

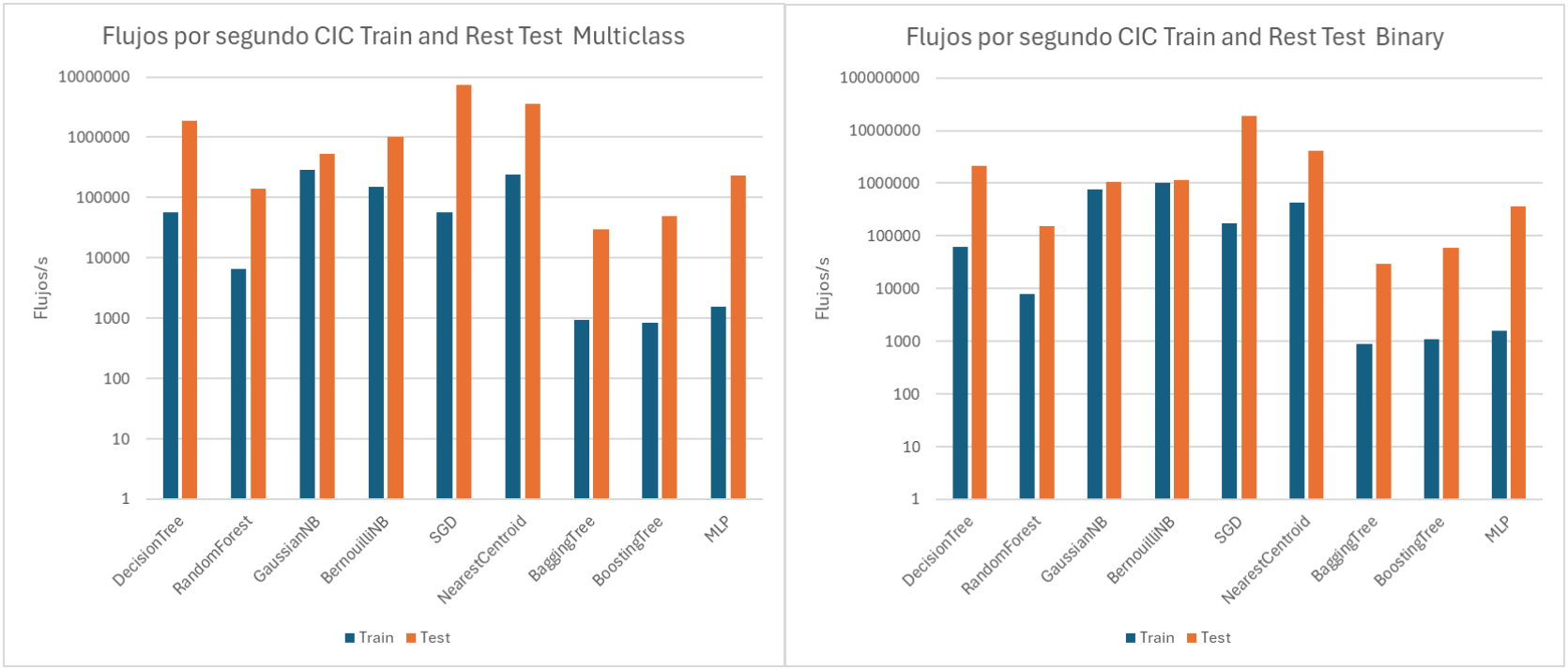





Flujos por segundo All Train and Test binary presplit
Flujos/s
100000000
10000000
1000000
100000
10000
1000
100
10
1
DecisionTree
RandomForest
GaussianNB
BernoulliNB
SGD
NearestCentroid
BaggingTree
BoostingTree
MLP
Train
Test


Flujos por segundo All Train and Test Multiclass
Flujos/s
10000000
1000000
100000
10000
1000
100
10
1
DecisionTree
RandomForest
GaussianNB
BernoulliNB
SGD
NearestCentroid
BaggingTree
BoostingTree
MLP
Train
Test


Flujos por segundo All Train and Test Multiclass con FS
Flujos/s
10000000
1000000
100000
10000
1000
100
10
1
DecisionTree
RandomForest
GaussianNB
BernoulliNB
SGD
NearestCentroid
BaggingTree
BoostingTree
MLP
Train
Test


Flujos por segundo CIC Train and Rest Test Multiclass
Flujos/s
10000000
1000000
100000
10000
1000
100
10
1
DecisionTree
RandomForest
GaussianNB
BernoulliNB
SGD
NearestCentroid
BaggingTree
BoostingTree
MLP
Train
Test
Flujos por segundo CIC Train and Rest Test Binary
Flujos/s
100000000
10000000
1000000
100000
10000
1000
100
10
1
DecisionTree
RandomForest
GaussianNB
BernoulliNB
SGD
NearestCentroid
BaggingTree
BoostingTree
MLP
Train
Test