UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Implementación Eficiente de Clasificadores Sencillos para Paquetes de Red

Autor: Alberto Cañas Gutiérrez de Cabiedes
Tutor: Daniel Perdices Burrero

enero 2026

Alberto Cañas Gutiérrez de Cabiedes
Implementación Eficiente de Clasificadores Sencillos para Paquetes de Red

Alberto Cañas Gutiérrez de Cabiedes

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A mis padres, a mis hermanos, a mis amigos y a nuestra Señora la Virgen María madre de Las Tablas y Schoenstatt.

“No puedes defender. No puedes prevenir. Lo único que puedes hacer es detectar y responder” – Bruce Schneier

Agradecimientos

En primer lugar, me gustaría agradecer a mis padres su apoyo y cariño a lo largo de toda mi etapa de estudiante así como por financiar mis estudios universitarios. También me gustaría agradecer a mis hermanos su cariño y apoyo. Sin el apoyo de mi familia no habría podido llegar tan lejos.

Quiero agradecer a mis compañeros Aitana Ayala, Diego Grande, Laura López, Cecilia Jiménez, Antonio Quintana, Jesús Quintana y en especial a Luis Arranz quien ha sido mi compañero en la mayoría de las asignaturas de prácticas, su apoyo, su cariño y los ratos de estudio juntos en la biblioteca de la EPS. Sin ellos mi estancia en la universidad habría sido mucho más difícil. También quiero agradecer a mis amigos que no forman parte de la universidad por su apoyo y su tiempo dedicado a mí durante todos estos años.

Por último me gustaría agradecer a todos mis profesores de la Escuela Politécnica Superior, de quienes he aprendido una gran cantidad de conocimientos que me han nutrido como persona y como profesional. Me gustaría agradecer especialmente a mis profesores de Programación I, Programación II, Sistemas Operativos, Inteligencia Artificial, Redes I, Redes II y Ciberseguridad por transmitirme los conocimientos que he necesitado para poder plantear, emprender y ejecutar este Trabajo de Fin de Grado. Me gustaría agradecer también de manera especial a mi tutor de TFG Daniel Perdices Burrero por el interés dedicado a este trabajo y por su guía sin la cual no habría podido desarrollarlo.

Resumen

El incremento constante de ataques de ransomware ha impulsado el desarrollo de mecanismos de detección y filtrado cada vez más eficientes, especialmente en entornos donde la latencia y el rendimiento son críticos. En este trabajo se evalúa el impacto en el rendimiento de red que supone aplicar un filtro preliminar basado en un árbol de decisión para la detección de comunicaciones potencialmente asociadas a ransomware.

Para ello, se entrena un clasificador mediante *scikit-learn* y se integra su lógica en el plano de datos mediante eBPF/XDP, generando automáticamente código C restringido a partir del modelo entrenado. La experimentación se realiza en un entorno controlado utilizando Mininet y herramientas de generación y reproducción de tráfico, incluyendo un script propio que emula escenarios mixtos de tráfico benigno y malicioso. Los resultados indican que el filtrado basado en XDP permite mantener un rendimiento útil comparable con y sin filtro, con un coste reducido en recursos del sistema, lo que respalda la viabilidad de utilizar árboles de decisión como filtros preliminares en arquitecturas *inline* y en dispositivos como SmartNICs.

Palabras clave

Ransomware, eBPF, XDP, árboles de decisión, filtrado de paquetes, rendimiento de red, SmartNIC, Mininet

Abstract

The continuous increase in ransomware attacks has motivated the development of efficient detection and filtering mechanisms, particularly in scenarios where low latency and high throughput are critical. This work evaluates the network performance impact of deploying a preliminary filter based on a decision tree classifier to identify communications potentially related to ransomware.

A model is trained using *scikit-learn* and its logic is integrated into the data plane through eBPF/XDP, automatically generating restricted C code from the trained decision tree. Experiments are conducted in a controlled environment using Mininet and traffic generation and replay tools, including a custom script designed to emulate mixed benign and malicious traffic scenarios. Results show that XDP-based filtering can sustain comparable effective throughput with and without the filter, while incurring a low system resource overhead, supporting the feasibility of decision-tree filters as lightweight preliminary defenses in *inline* architectures and SmartNIC-like deployments.

Keywords

Ransomware, eBPF, XDP, decision trees, packet filtering, network performance, SmartNIC, Mininet

Índice

Listas

Lista de algoritmos

Lista de códigos

Lista de cuadros

Lista de ecuaciones

Lista de figuras

Lista de tablas

Lista de cuadros

1

Introducción

1.1. Motivación

El ransomware es una forma específica de malware cuyo objetivo principal es bloquear el acceso a los datos de una organización o individuo, generalmente mediante técnicas de cifrado. Una vez que los archivos han sido cifrados, el atacante demanda el pago de un rescate (habitualmente en criptomonedas) a cambio de proporcionar la clave necesaria para restaurar la información secuestrada [1].

Esta clase de ataques se ha convertido en una amenaza persistente y cada vez más frecuente en el panorama actual de la ciberseguridad. Según el informe anual de la empresa especializada en ciberseguridad Cyberint [2], el pasado año 2024 se reportaron, a nivel global, 5.414 ataques de ransomware alrededor del mundo lo que supone un incremento del 11 % respecto al año 2023. Es también destacable el hecho de que un tercio del total de los ataques se realizaron en el último trimestre del año, lo que puede indicar una tendencia de aumento de ataques de esta clase en el presente año 2025.

Dado que este tipo de ataques no solo persisten, sino que van en aumento, resulta imprescindible desarrollar mecanismos capaces de detectarlos y prevenirlos sin que ello afecte negativamente al rendimiento de los sistemas ni se convierta en un cuello de botella. Para poder desplegar un sistema de detección y respuesta ante ransomware sin afectar al rendimiento del sistema, es fundamental analizar dónde y cómo implementar dicho sistema, tanto a nivel lógico como físico.

Un aspecto fundamental a considerar es en qué espacio del sistema operativo debe desplegarse la medida de seguridad. En el artículo de Parola et al.(2023) [3], se compara el rendimiento de la gestión de paquetes en tres escenarios: desde el espacio de usuario mediante el flujo tradicional, desde el espacio de usuario utilizando la herramienta DPDK —que opera mediante técnicas de *polling* para evitar la intervención del kernel— y, finalmente, desde el propio espacio del kernel.

Aunque DPDK presenta la menor latencia, el estudio concluye que, en términos de equilibrio entre rendimiento y consumo de recursos, el enfoque más eficiente es el que se implementa directamente

en el espacio del kernel. Este equilibrio sugiere que podría resultar especialmente interesante explorar el desarrollo de mecanismos de detección de ransomware a nivel de kernel, ya que permitiría una supervisión más cercana al sistema sin comprometer significativamente el rendimiento.

Otro aspecto relevante a considerar es el nodo físico de la red en el que debe implementarse el sistema de detección. La práctica más común consiste en desplegar el software de detección directamente en el servidor que se desea proteger —por ejemplo, el servidor A—. Sin embargo, teniendo en cuenta el crecimiento sostenido de las redes, con una tasa de crecimiento compuesta del 24 % según un informe de Cisco (2021) [4], este enfoque puede resultar cada vez menos eficiente.

El aumento del tráfico puede derivar en una mayor exposición a ataques, lo que incrementa la carga computacional que el servidor A debe asumir para protegerse a sí mismo. Por ello, puede resultar conveniente delegar parte de esta responsabilidad a otros componentes de la infraestructura de red, como las Tarjetas de Interfaz de Red (NIC, por sus siglas en inglés). Dentro de esta categoría, destacan especialmente las SmartNICs, que incorporan unidades de procesamiento dedicadas (DPUs) capaces de ejecutar tareas de seguridad de forma autónoma, descargando así al servidor principal.

1.2. Objetivos

El objetivo principal de este trabajo es evaluar el impacto que la aplicación de un sistema de filtrado de tráfico basado en clasificadores sencillos puede tener sobre el rendimiento de la red. Dicho sistema está diseñado para identificar patrones de comunicación asociados a posibles ataques de tipo ransomware, y para ello se implementará el árbol de decisión en lenguaje C utilizando tecnologías eBPF/XDP, lo que permitirá su integración directa en el plano de datos de red, reduciendo la latencia y mejorando la eficiencia del procesamiento. Como objetivos secundarios, se analizará la viabilidad de desplegar este mecanismo de filtrado en dispositivos distintos a los servidores convencionales, como las SmartNICs.

1.3. Estructura del documento

El presente documento se divide en cinco capítulos. El primer capítulo explica la importancia de investigar métodos de filtrado de ransomware y su impacto en el rendimiento de la red. El segundo capítulo aborda el conjunto de datos utilizado para entrenar el árbol de decisión, así como el estado del arte de técnicas, herramientas y artículos que ofrecen soluciones similares o enfoques relacionados. El tercer capítulo desarrolla la metodología empleada en el trabajo, describiendo la implementación realizada con eBPF/XDP, scikit-learn, Mininet y Jinja. El cuarto capítulo presenta los experimentos y métricas utilizadas para el análisis. Finalmente, el quinto capítulo expone las conclusiones y comentarios.

2

Estado del Arte

2.1. Conjunto de Datos

En este apartado se describe el conjunto de datos empleado para el entrenamiento del modelo de árbol de decisión, diseñado con el objetivo de detectar posibles comunicaciones asociadas a ransomware.

El conjunto de datos utilizado proviene del trabajo realizado por Eduardo Berrueta et al. entre los años 2015 y 2022 [5]. Este *dataset* contiene capturas de tráfico en formato .pcap correspondientes a setenta familias distintas de ransomware, con un volumen total superior a 60 GB con los datos comprimidos.

Se ha optado por trabajar únicamente con dos de las muestras incluidas en el conjunto de datos original, que serán los que trataremos en profundidad en este apartado, porque no es necesario entrenara con la totalidad de las clases para evaluar la viabilidad y el rendimiento.

Pero antes de tratar estos dos subconjuntos de datos en concreto, cabe hacer una mención al artículo de Eduardo Berrueta y su equipo [6]. En él explican que la motivación para crear el *dataset* proviene de la necesidad de estandarizar recursos a la hora de evaluar la fiabilidad de las herramientas de detección de ransomware, ya que habitualmente cada fabricante o investigador utiliza su propio conjunto de datos. Al ser estos conjuntos muy específicos y reducidos, las comparativas resultantes tienden a ser poco realistas. Con su trabajo, los actores pretenden aportar un *dataset* completo para pruebas de detección de ransomware que contribuya a resolver estas problemáticas.

Para este trabajo he usado muestras de dos ransomwares tomadas en el año 2021. El primero de los casos analizados es Hive, un conocido Ransomware-as-a-Service (RaaS) que emergió en junio de 2021. Este ransomware se caracterizaba por emplear un esquema de cifrado híbrido basado en los algoritmos RSA y AES, y por implementar una estrategia de doble extorsión. En este modelo, no solo se cifraban los datos de la víctima, sino que también se exfiltraban, exigiendo posteriormente un pago tanto por la clave de descifrado como por evitar la publicación de la información sustraída [7]. El segundo de los casos analizados es CryLock, una variante evolucionada del ransomware Cryakl.

CryLock se propagaba principalmente a través de correos electrónicos de phishing o campañas de spam, y al igual que otros ransomware modernos, empleaba un esquema de cifrado híbrido basado en los algoritmos RSA y AES. Además, una de sus características distintivas era la eliminación de las copias de seguridad presentes en los dispositivos comprometidos, con el objetivo de dificultar la recuperación de los datos sin realizar el pago del rescate. [8]

2.2. Métodos de Procesado de Paquetes a altas velocidades

El filtrado de paquetes constituye una de las técnicas fundamentales para el control del tráfico en redes informáticas. Su propósito principal es inspeccionar, permitir o bloquear paquetes de datos en función de criterios previamente definidos, como direcciones IP, puertos, protocolos o patrones de comportamiento.

A lo largo de la evolución de las redes de comunicaciones, el problema del filtrado de paquetes ha sido abordado mediante una amplia variedad de enfoques, que van desde soluciones sencillas y de rápida implementación hasta mecanismos con un alto grado de complejidad técnica. El mecanismo más sencillo de implementar para el filtrado de paquetes es el uso de un firewall. En los sistemas operativos GNU/Linux, este puede configurarse mediante el comando iptables, que opera a través de Netfilter, el subsistema encargado del filtrado de paquetes a nivel de kernel. El hecho de que el firewall funcione en espacio de kernel permite que su impacto sobre el rendimiento del sistema sea reducido. Su función principal consiste en controlar el tratamiento de los paquetes de red entrantes, salientes y reenviados, permitiendo aceptarlos, denegarlos, redirigirlos o registrarlos en función de criterios como dirección IP, puertos, protocolos, entre otros. [9]

Otro enfoque relevante para el filtrado de paquetes, especialmente orientado a la seguridad de los sistemas, es el uso de sistemas de detección de intrusiones (Intrusion Detection Systems, IDS por sus siglas en inglés). Uno de los más reconocidos en este ámbito es Snort, ampliamente utilizado desde su lanzamiento en 1998 y caracterizado por ser de código abierto [10]. Para llevar a cabo el filtrado, Snort emplea la biblioteca libpcap para capturar paquetes de red, que posteriormente son decodificados para identificar su estructura y los protocolos involucrados (Ethernet, IP, TCP/UDP, entre otros). A partir de esta información, Snort monitoriza el tráfico y genera alertas en función de un conjunto de reglas predefinidas. Además, este sistema puede operar en diferentes modos: *sniffer*, *logger* o como IDS propiamente dicho, y permite su ampliación mediante un sistema de plugins.

Esta evolución desde mecanismos de filtrado simples hacia un análisis más profundo del tráfico se ha visto respaldada por tecnologías como libpcap y BPF (Berkeley Packet Filter), que permiten la captura eficiente de paquetes sin comprometer significativamente el rendimiento del sistema. BPF es una interfaz que habilita la captura de paquetes a través de programas escritos en lenguaje C, los

cuales deben ajustarse a un conjunto de restricciones más estricto que el lenguaje C estándar. Un aspecto clave de BPF es su modelo de ejecución: aunque los filtros se definen desde el espacio de usuario, su ejecución se realiza en el espacio del kernel, lo que permite un procesamiento más rápido y eficiente al evitar cambios de contexto innecesarios [11].

Por otro lado, libpcap es una biblioteca inicialmente desarrollada en C, que actualmente cuenta con una versión en Python, y que se utiliza para capturar paquetes ofreciendo una API uniforme para acceder a datos de red a bajo nivel desde el espacio de usuario. Cuando el sistema lo permite, libpcap hace uso de BPF para realizar filtrado a nivel de kernel [12]. Esta biblioteca (y, por extensión, BPF) ha sido utilizada en multitud de herramientas, entre ellas Wireshark [13].

BPF ha ido evolucionando a lo largo de los años, llegando su desarrollo hasta eBPF (Extended Berkeley Packet Filter). eBPF, como su propio nombre indica, comenzó siendo una extensión de BPF, pero con el paso del tiempo se ha convertido en una plataforma más general capaz de interactuar con distintas partes del sistema operativo, tales como redes, seguridad, trazado y observabilidad. A diferencia del BPF original —que únicamente filtraba paquetes en el espacio del socket—, eBPF permite engancharse (*hook*) en múltiples puntos del kernel, ejecutar programas verificados para garantizar la seguridad, utilizar mapas compartidos entre el kernel y el espacio de usuario, y beneficiarse de la compilación *JIT* (Just In Time) para un alto rendimiento [14]. En base a eBPF, se ha desarrollado XDP (eXpress Data Path), una tecnología para el kernel de Linux que permite ejecutar programas eBPF en una etapa más temprana del procesamiento de paquetes, lo que se traduce en una menor latencia y un mayor rendimiento. Para ello, XDP establece como *hook* el controlador (driver) de la tarjeta de red, o bien opera en modos alternativos como *generic* (pila de red) u *offload* (hardware si la NIC lo soporta). De este modo, el filtrado de paquetes se sitúa en la capa más baja de la pila de red. Es importante señalar que XDP requiere especificar la tarjeta de red sobre la que se ejecutará el programa eBPF [15].

Para entender lo que resta del presente trabajo es necesario entender el flujo de funcionamiento de XDP y por lo tanto es necesario conocer el flujo de eBPF, para ello nos basaremos en 2.1 obtenido de [14]. El primer paso consiste en escribir el programa en un subconjunto restringido del lenguaje C. Posteriormente, mediante herramientas como Clang y LLVM, el código se compila a bytecode. A continuación, desde el espacio de usuario, el programa se carga en el espacio del kernel y se prepara el espacio de usuario para recibir información mediante los mapas definidos en el programa, en caso de ser necesario. Estos dos últimos pasos pueden realizarse mediante un programa en C en el espacio de usuario utilizando la biblioteca libbpf. Una vez que el programa llega al kernel, este pasa por el verificador, cuyo objetivo es impedir la ejecución de acciones no permitidas, como bucles infinitos o accesos ilegales a memoria. A continuación, el compilador en tiempo real (*Just-In-Time* compiler, *JIT*) transforma el bytecode en código máquina. Tras este proceso, el programa se “engancha” (*hook*) a diferentes eventos de la capa de red, como, por ejemplo, el controlador (driver) de la tarjeta de red en el caso de XDP. Durante la ejecución, y siempre que sea necesario, el programa interactúa con los mapas definidos. Este flujo garantiza que el código eBPF se ejecute de forma segura y eficiente,

integrándose de manera controlada en el sistema operativo.

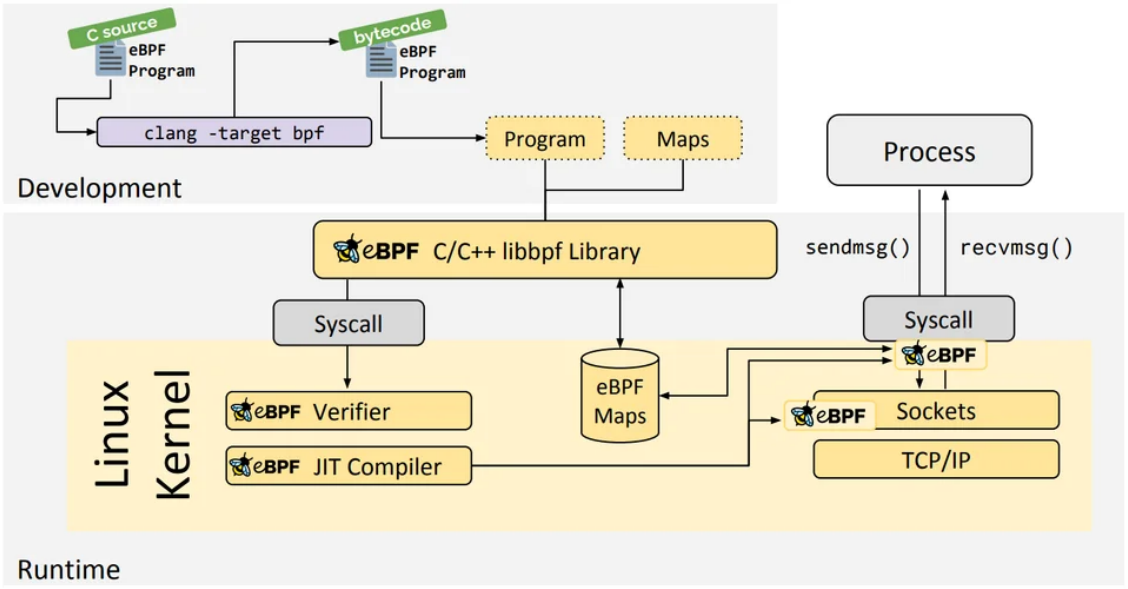

Figura 2.1: Flujo de ejecución de un programa eBPF. Fuente [14]

2.3. Trabajos Relacionados

Como se ha expuesto en la motivación y en las subsecciones anteriores, la detección temprana de ransomware y la necesidad de contar con sistemas de filtrado de paquetes que ofrezcan un alto rendimiento y una baja latencia constituyen problemáticas que han captado la atención y el interés de la comunidad académica desde hace tiempo. Dado que el presente trabajo se centra en analizar el impacto en el rendimiento de red de sistemas de detección y filtrado de ransomware basados en árboles de decisión, resulta pertinente revisar los trabajos académicos que exploran tanto el uso de tecnologías como eBPF y XDP en el filtrado de tráfico malicioso, como aquellos que estudian la aplicación de árboles de decisión para la detección de ransomware.

La idea de utilizar eBPF o XDP para filtrar tráfico malicioso ha sido explorada en múltiples ocasiones. Por ejemplo, el trabajo de Farasat et al. (2024) [16] propone un *framework* de seguridad denominado SmartX Intelligent Sec, que emplea eBPF/XDP como método de captura continua de paquetes. Una vez capturados, los paquetes son analizados por una red BiLSTM en el espacio de usuario, y, en función del análisis de riesgos realizado por el modelo, eBPF/XDP filtra el paquete o lo deja pasar. El estudio destaca las ventajas de emplear eBPF/XDP por su ligereza y rapidez, lo que evita problemas de saturación por consumo de recursos, como los que pueden producirse en sistemas de detección virtualizados que corren en el espacio de usuario. Este *framework* se describe como generalista y, en el artículo, se muestra su eficacia principalmente frente a ataques DDoS.

En cuanto al uso de eBPF/XDP para la detección de ransomware, sobresale el trabajo de Zhu-

ravchak et al. (2023) [17], en el que se analiza la viabilidad de desarrollar sistemas de detección de ransomware basados en eBPF/XDP. El artículo resalta su eficiencia y seguridad, permitiendo una detección rápida, precisa, flexible y escalable mediante el análisis de parámetros de red y la ejecución de código dentro del kernel de Linux sin necesidad de modificarlo. Esta característica facilita la adaptación del sistema al tráfico de red sin reinicios, promoviendo la automatización de la detección y la respuesta, y, por ende, mejorando la seguridad. Como ejemplo, se menciona Cilium, una plataforma de software de código abierto diseñada para proporcionar conectividad de red eficiente entre aplicaciones y servicios en entornos de contenedores como Docker y Kubernetes. Cilium se basa en la tecnología eBPF, lo que le permite integrar de manera dinámica controles de seguridad y gestión directamente en el sistema Linux, aplicando y actualizando políticas de seguridad sin necesidad de modificar el código de la aplicación ni la configuración del contenedor.

Otro artículo relevante, que se aproxima tangencialmente al presente trabajo, es el de Higuchi et al. (2023) [18]. En lugar de utilizar eBPF para filtrar paquetes de red entrantes, este estudio explota la capacidad de eBPF de engancharse (*hook*) en distintas partes del kernel para monitorizar otros procesos del sistema, tales como llamadas al sistema operativo que permiten detectar modificaciones en ficheros, incluyendo operaciones de lectura, escritura, eliminación, renombrado y apertura. Los datos capturados por eBPF son posteriormente analizados en el espacio de usuario mediante un modelo de Machine Learning; el artículo menciona tres modelos distintos, entre los cuales destaca *random forest*, una evolución de los árboles de decisión empleados en el presente trabajo. Este modelo evalúa si la acción corresponde al inicio de un proceso de cifrado de ficheros por parte de un ransomware. Durante el análisis preventivo, el fichero se marca como inalterable, y, en función del resultado del modelo, el fichero permanecerá protegido o permitirá su modificación. El estudio subraya que el uso de eBPF se justifica por su seguridad y por la posibilidad de ejecutar código dentro del kernel sin necesidad de modificarlo. Además, debido a su ligereza, eBPF resulta especialmente adecuado para entornos virtualizados, como contenedores.

Aunque uno de los artículos más relevantes sobre el uso de eBPF para la detección de malware es el de Brodzik et al. (2024) [19], cuyo enfoque guarda ciertas similitudes con el trabajo de Higuchi et al., presenta una aproximación diferente. En lugar de utilizar eBPF para el filtrado de paquetes de red, lo emplea para monitorizar procesos del sistema operativo.

Las principales diferencias entre ambos trabajos radican, en primer lugar, en el ámbito de monitorización y, en segundo lugar, en el lugar donde se implementa el modelo de Machine Learning. Mientras que Higuchi et al. se centran en la protección del sistema de archivos, Brodzik et al. orientan su investigación hacia la detección y eliminación de procesos maliciosos en entornos de nube basados en Linux. Una característica destacable de este enfoque es que no requiere un sistema específico de protección de datos, gracias a la baja latencia que se obtiene al implementar directamente los modelos de Machine Learning en el espacio del kernel mediante eBPF.

Este último aspecto resulta particularmente relevante para el presente trabajo, ya que el estudio de Brodzik et al. incluye una comparativa entre la ejecución de un árbol de decisión en el espacio de usuario y en el espacio de kernel. Los resultados ponen de manifiesto que la segunda opción reduce de manera significativa la latencia, lo que incrementa la eficacia del sistema para detener ataques de ransomware en tiempo real.

Siguiendo con metodologías basadas en el uso de eBPF para la detección de ransomware, resulta relevante el trabajo de Sekar et al. (2024) [20]. En este estudio, eBPF se emplea para monitorizar más de 325 llamadas al sistema, lo que pone de manifiesto su elevada eficiencia para la recolección de datos a nivel de kernel. Posteriormente, en el espacio de usuario, los autores aplican modelos de Inteligencia Artificial con el fin de clasificar los procesos en benignos o maliciosos (ransomware). Como paso adicional, incorporan técnicas de procesamiento de lenguaje natural (NLP) para identificar posibles mensajes de rescate desplegados por el ransomware, complementando así el proceso de detección y respuesta.

El uso de modelos de Machine Learning e Inteligencia Artificial para la detección de ransomware, así como el análisis de su eficiencia, ha sido objeto de numerosos estudios en los últimos años. Uno de los trabajos más relevantes en esta línea es la revisión realizada por Bello et al. (2021) [21], en la que se examina el desempeño de tres enfoques principales: *deep learning*, *random forest* y árboles de decisión. En lo que respecta a los árboles de decisión, el estudio subraya que estos han mostrado resultados positivos tanto en entornos Windows como en redes en general, alcanzando en muchos casos una mayor precisión que otras técnicas cuando se combinan con métodos de selección de características. Asimismo, se resalta su facilidad de interpretación y rapidez en el entrenamiento, aunque también se identifican limitaciones, especialmente frente a variantes de ransomware que modifican dinámicamente su comportamiento.

Debido a la naturaleza del presente trabajo, resulta especialmente relevante el artículo de Te-Min Liu et al. (2020) [22], en el cual se aborda la detección de ransomware a partir del análisis de características ligeras del tráfico de red. Entre estas características destacan el tamaño de los paquetes, la dirección de destino, el puerto utilizado y la presencia de protocolos no estándar. Además, se complementa este análisis con el estudio de la frecuencia del tráfico, lo que permite identificar patrones de comunicación característicos del ransomware. La propuesta presentada en dicho trabajo se centra, por tanto, en aprovechar información básica pero significativa de las comunicaciones para desarrollar un sistema de detección eficiente y con bajo coste computacional, lo que lo convierte en un enfoque de interés en el ámbito de la seguridad de redes frente a amenazas de este tipo.

3

Implementación

3.1. Arquitectura de la propuesta

En esta sección se ofrece una visión global de la arquitectura propuesta, describiendo los componentes principales que la conforman y el flujo de información entre ellos. Para facilitar la comprensión, se ha elaborado el diagrama mostrado en la figura 3.1, que servirá como referencia a lo largo de este capítulo. Dicho diagrama proporciona una perspectiva general de la integración entre los distintos módulos, sin entrar todavía en los detalles específicos de implementación, los cuales se desarrollan en la sección siguiente.

La arquitectura de la implementación se organiza en tres bloques principales. El primero corresponde al tratamiento de los datos y el entrenamiento del árbol de decisión, mientras que el segundo se centra en la autogeneración del código XDP y el tercero en su posterior ejecución.

En la primera fase, se seleccionan los conjuntos de paquetes que serán utilizados para el entrenamiento del árbol de decisión. Estos datos son procesados en el espacio de usuario mediante un script en Python, encargado de transformar la información en un fichero CSV apto para su uso en el proceso de entrenamiento. A continuación, otro script en Python utiliza dicho fichero CSV para entrenar el árbol de decisión, generando así un modelo inicial.

En la segunda fase, el mismo script aplica un conjunto de reglas de traducción con el objetivo de convertir el árbol de decisión resultante en una serie de condiciones expresadas en código C restringido, compatibles con eBPF/XDP. Posteriormente, estas condiciones se integran en una plantilla Jinja, que permite generar automáticamente el programa XDP completo, el cual incluye tanto la lógica de clasificación derivada del árbol como el resto de estructuras auxiliares necesarias para su correcta ejecución.

La tercera fase corresponde a la ejecución del sistema. En este punto, el proceso principal consiste en la utilización de un Makefile, encargado de automatizar la compilación y despliegue del entorno. Dicho Makefile genera, por un lado, el bytecode del programa XDP, y por otro, el ejecutable del programa en espacio de usuario. Este último tiene como funciones tanto la carga del bytecode en el kernel del

sistema como la monitorización en tiempo real de los paquetes que son descartados o aceptados en función de las reglas definidas por el árbol de decisión.

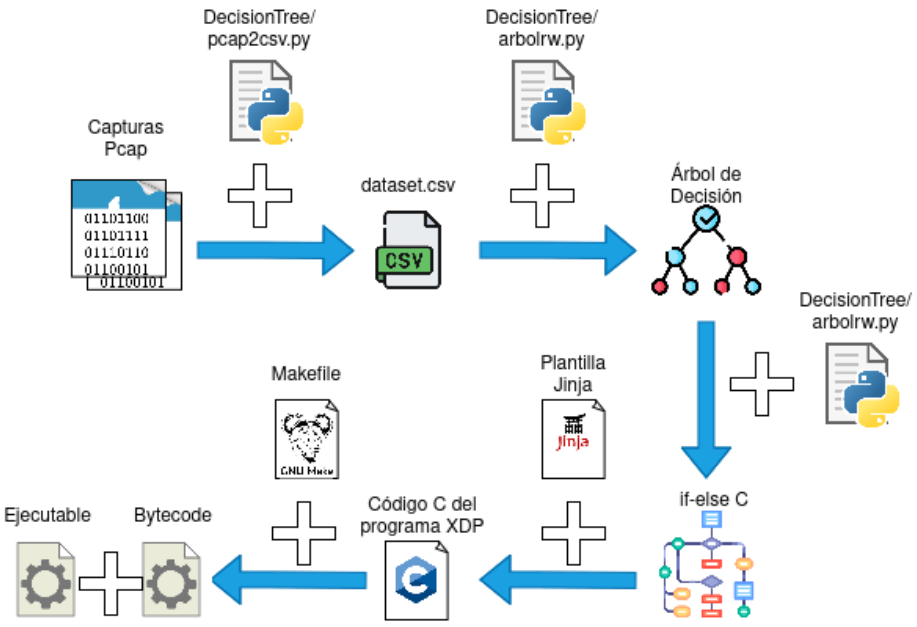

Figura 3.1: Arquitectura de generación de filtros eBPF

Los componentes de la arquitectura se organizan en tres módulos principales. El primero corresponde al tratamiento de datos y a su conversión desde paquetes de red a un fichero CSV. El segundo módulo está constituido por el árbol de decisión, entrenado mediante la biblioteca scikit-learn de Python. Finalmente, el tercer módulo corresponde al código XDP, el cual incluye una parte autogenerada a partir del árbol de decisión. Cabe destacar el fichero `DecisionTree/arbolrw.py`, encargado tanto del entrenamiento del árbol de decisión como de la autogeneración del código C, apoyándose para ello en una plantilla Jinja.

3.2. Tratamiento de datos

El conjunto de datos empleado para el entrenamiento del árbol de decisión se ha generado a partir de un fichero CSV de 294,6 MB elaborado por el autor. Para su creación, se utilizó la versión más reciente del script pcap2csv.py, el cual permite la unión de dos o más ficheros PCAP en un único fichero CSV, así como la diferenciación entre paquetes de tráfico malicioso y legítimo mediante el uso de las banderas -l y -m. El script almacena únicamente información de los paquetes hasta el nivel de aplicación, sin analizar el contenido de este nivel, debido a la complejidad que supone obtener dichos datos en C a nivel de kernel.

El proceso de conversión comienza definiendo un diccionario con todas las posibles características

que puede tener un paquete de red. A continuación, se leen los paquetes uno a uno y cada uno se procesa mediante una función que devuelve un diccionario en el que cada clave corresponde al nombre de una columna y cada valor al dato asociado a ese paquete en particular. En caso de que un paquete carezca de alguno de los campos —por ejemplo, los relativos a IP—, dichos valores se registran como None en lugar de omitirse, con el fin de mantener un formato uniforme en el CSV y evitar errores durante el procesamiento posterior.

Una vez transformado el paquete en diccionario, se añade al final la etiqueta “Yes” si corresponde a una muestra de ransomware, o “No” en caso contrario. Posteriormente, el diccionario se incorpora a un array. Cuando dicho array alcanza los 1024 elementos, su contenido se vuelca al fichero CSV y se vacía para liberar memoria. Este procedimiento se repite hasta procesar todos los paquetes, momento en el que se realiza un último volcado con los datos restantes. Se ha optado por este método de lectura y escritura para optimizar tanto el uso de memoria como la frecuencia de accesos de escritura al disco.

Los ficheros PCAP correspondientes al tráfico de ransomware fueron obtenidos, tal y como se explica en el apartado 2.1 por el trabajo de Eduardo Berrueta et al. entre los años 2015 y 2022 [5]. Por otro lado, el fichero PCAP con tráfico legítimo fue generado por el autor mediante la captura de tráfico durante una jornada laboral en un laboratorio universitario, registrando el uso normal de los dispositivos y recogiendo tráfico de protocolos como HTTP, HTTPS, DNS, SSH, entre otros. Con el objetivo de incrementar la precisión del árbol de decisión, se generó adicionalmente tráfico SMB2 mediante la copia de un fichero de gran tamaño a una unidad de almacenamiento ubicada en el CPD de la universidad.

En el proceso de preparación del *dataset*, se eliminaron las direcciones IP, las direcciones MAC y el timestamp de los paquetes, debido al número limitado presente tanto en las capturas de tráfico malicioso como en las de tráfico legítimo. Con ello, se evitó que el árbol de decisión se apoyara en características poco representativas o irrelevantes para un entorno real.

3.3. Árbol de decisión

En esta subsección se detalla el proceso de configuración y entrenamiento del árbol de decisión utilizado en el presente trabajo. Se explican los criterios adoptados para su construcción, las particularidades detectadas durante el desarrollo y las adaptaciones realizadas en función de los resultados obtenidos.

Como métrica de calidad se mantuvo el índice de Gini, dada su fiabilidad y buen rendimiento, y como estrategia de división se empleó el criterio best. Aunque se preveía que el aumento del tamaño del *dataset* respecto a las pruebas pudiera incrementar el tiempo de entrenamiento, en la práctica no se observó un impacto significativo. En caso contrario, se habría considerado cambiar la estrategia a random. En cuanto a la profundidad máxima del árbol, se fijó inicialmente en cuatro niveles, ya que en

pruebas previas con otros conjuntos de datos no se apreciaron mejoras de precisión al superar este valor.

Para entrenar el árbol de decisión, una vez definidas las columnas que serían eliminadas, se estableció la variable `x` como el conjunto de columnas correspondientes a los datos de entrada y la variable `y` como la columna asociada al clasificador. A continuación, mediante la función `train_test_split`, se dividió el conjunto de datos de forma aleatoria, asignando el 80 % de las filas al entrenamiento y el 20 % restante a las pruebas, con el fin de evaluar la fiabilidad del modelo. Una vez realizada esta partición, se procedió a entrenar el árbol utilizando los parámetros seleccionados y los datos de entrenamiento. Finalmente, se calculó la exactitud del modelo, si bien este valor debe interpretarse con cautela, dado que no siempre refleja de forma sólida la capacidad de generalización del clasificador. El código correspondiente al entrenamiento del árbol puede consultarse en el fragmento de código 3.1.

Durante el desarrollo se identificó un error en la configuración del entrenamiento: la variable `eth_type` había sido incluida por error en la lista de columnas excluidas. Una vez corregida esta omisión y tras reentrenar el modelo, se observó que la precisión global se mantenía estable, aunque con una diferencia relevante en la complejidad del árbol, ya que la profundidad máxima alcanzada fue de tres niveles en lugar de cuatro, a pesar de que el límite establecido seguía siendo cuatro. Este comportamiento sugiere que la incorporación de `eth_type` aporta capacidad predictiva al modelo, permitiendo una estructura más optimizada sin comprometer la precisión.

Posteriormente, se decidió eliminar la variable TTL al comprobar que resultaba tan poco fiable como las direcciones IP, introduciendo un nivel elevado de variabilidad. Tras esta modificación, el árbol resultante mostró una mayor amplitud, alcanzando ahora los cuatro niveles de profundidad. Aunque estas observaciones pudieran parecer superficiales, resultan de especial relevancia, ya que tanto el desarrollo del código como la generación automática de este se fundamentan en los árboles de decisión obtenidos, y en la evolución observada de uno respecto al otro.

Código 3.1: Fragmento de código de arbolrw.py encargado de entrenar el árbol de decisión.

```
114     columnas_caracteristicas = [col for col in df.columns if col != nombre_columna_objetivo and col not
            in columnas_a_excluir]
115     x = df[columnas_caracteristicas]
116     x = pd.get_dummies(x)
117
118     # División de datos
119     x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
120
121     # Entrenar el modelo
122     modelo = DecisionTreeClassifier(criterion='gini', splitter='best', max_depth=4, random_state=42)
123     modelo.fit(x_train, y_train)
124     exactitud = modelo.score(x_test, y_test)
125     print(f'La exactitud del modelo es: {exactitud: .2f}')
```

3.4. Código XDP y Código autogenerado

Tal y como se indicó en el apartado 2.2, los programas XDP constan de dos componentes diferenciados. El primero de ellos es un programa escrito en C que se compila de forma convencional y cuya ejecución tiene lugar íntegramente en el espacio de usuario. El segundo componente corresponde a un programa que se traduce a bytecode y que, posteriormente, es cargado por el primero en el espacio del kernel, donde finalmente se ejecuta.

El programa de usuario inicia comprobando que se haya especificado la interfaz de red sobre la que se ejecutará el programa XDP y verificando que dicha interfaz exista efectivamente en el equipo. A continuación, mediante la función `bpf_object__open_file`, el programa analiza el objeto XDP y prepara las estructuras necesarias en el espacio de usuario, como los mapas. Asimismo, se crea en memoria un `struct bpf_object`, desde el cual se podrá acceder a estas estructuras en el espacio de usuario. Una vez preparado el entorno de usuario, se carga el bytecode correspondiente en el espacio de kernel mediante la función `bpf_object__load`.

El programa C aún requiere el descriptor de archivo (file descriptor, FD) del mapa para poder interactuar con él, el cual se obtiene a través de la función `bpf_object__find_map_fd_by_name`, que recibe como parámetros la estructura previamente creada y el nombre del mapa definido en el programa XDP. Por seguridad y persistencia, se fija el mapa en `/sys/fs/bpf/` mediante la función `bpf_obj_pin`; esto permite acceder al mapa incluso si el proceso que lo creó finaliza de forma inesperada y, además, sirve para comprobar si el mapa ya existe, lo que indica que el programa se ha ejecutado previamente o se encuentra en ejecución en ese momento. Como última tarea para terminar de cargar el programa XDP se obtiene el descriptor de archivo de la función principal del programa XDP a través de la función `bpf_program__fd`, tras esto se adjunta dicha función al *hook* XDP que es el controlador de la gráfica a través de la función `bpf_set_link_xdp_fd`. Una vez cargado el programa XDP, el programa de usuario se dedica a monitorizar los cambios del mapa y a mostrarlos en pantalla, dicho mapa contiene tanto el número de paquetes descartados como el número de paquetes que el programa XDP ha permitido pasar. En el fragmento de código 3.2 se muestra la sección del fichero XDP/arbol1/xdp_usr.c que implementa las funciones principales previamente descritas.

En relación con el código que se ejecuta en el espacio del kernel, en una primera fase la traducción del árbol de decisión se realizaba de manera manual. Es decir, se convertía el árbol entrenado con la librería scikit-learn de Python a código C restringido, con el objetivo de comprender su funcionamiento interno. Una vez adquirido este entendimiento, se implementó la capacidad de autogenerar el código eBPFXDP mediante el uso de una plantilla intermedia de Jinja. El proceso consiste en recorrer de manera recursiva el árbol de decisión, sustituyendo cada nodo y sus hijos por condiciones if-else en formato C restringido. A continuación, este código autogenerado se incorpora en una plantilla Jinja que mantiene la estructura del programa eBPF/XDP final, incluyendo las funciones auxiliares y la función main, a excepción del árbol de decisión. Una vez completado este proceso, la plantilla Jinja rellenada

Código 3.2: Fragmento de código que muestra como se carga el objeto, obtiene y fija el mapa y como se adjunta el programa al *hook* XDP

```
28  obj = bpf_object__open_file("xdp_kern.o", NULL);
29  if (!obj) {
30      perror("Error al abrir el objeto BPF");
31      return 1;
32  }
33
34  if (bpf_object__load(obj)) {
35      perror("Error al cargar el programa BPF");
36      return 1;
37  }
38
39  map_fd = bpf_object__find_map_fd_by_name(obj, "packet_count");
40  if (map_fd < 0) {
41      perror("Error al obtener FD del mapa BPF");
42      return 1;
43  }
44
45  if (bpf_obj_pin(map_fd, MAP_PATH) < 0 && errno != EEXIST) {
46      perror("Error al fijar el mapa BPF en /sys/fs/bpf/");
47      return 1;
48  }
49
50  // Obtener el descriptor del programa XDP
51  prog_fd = bpf_program__fd(bpf_object__find_program_by_name(obj, "ransomware_tree"));
52  if (prog_fd < 0) {
53      perror("Error al obtener FD del programa BPF");
54      return 1;
55  }
56
57  if (bpf_set_link_xdp_fd(ifindex, prog_fd, 0) < 0) {
58      perror("Error al adjuntar el programa BPF a la interfaz");
59      return 1;
60  }
```

se vuelca en el fichero XDParbol_pruebaxdp_kern.c.

Debido a la forma en que funciona el entrenamiento de modelos en la biblioteca scikit-learn, así como a la representación de los datos en lenguaje C, fue necesario realizar un proceso de traducción adicional para la autogeneración del código XDP. Este proceso consistió en adaptar las condiciones presentes en el árbol de decisión al formato requerido por el lenguaje C. Durante el entrenamiento, scikit-learn transforma determinadas variables categóricas en rangos numéricos manejables por el propio programa. Por ejemplo, si una característica puede adoptar tres categorías distintas, el algoritmo genera tres rangos de valores continuos (`0-0.5`, `0.51-1`, `1.01-1.5`, por ejemplo) y asigna a cada categoría un valor específico dentro de dichos rangos, normalmente el límite superior. Asimismo, existe la posibilidad de que, en lugar de utilizar rangos discretos, scikit-learn emplee la condición `<= inf`, lo que en realidad significa que el valor de ese dato simplemente existe. Este último caso se da especialmente en aquellas características cuyo valor no es obligatorio y, por lo tanto, pueden no estar presentes en todos los registros.

En el caso concreto del árbol de decisión preliminar utilizado en el presente trabajo, mostrado en la figura 3.2, se observa la condición `src_port <= inf`, y en pruebas anteriores apareció la condición equivalente `dst_port <= inf`. En ambos casos, dichas condiciones se sustituyen por la expresión `ip_proto == IPPROTO_TCP || ip_proto == IPPROTO_UDP|`, ya que el hecho de que el puerto de origen o destino sea menor que infinito no implica más que la existencia de un puerto, lo cual solo resulta cierto para protocolos como TCP y UDP.

Existen otros elementos que presentan valores binarios, como la comprobación de si el protocolo de transporte corresponde a TCP o UDP. En estos casos, scikit-learn representa la condición mediante rangos numéricos, normalmente `0-0.5` y `0.51-1`. Se ha comprobado que, en este tipo de decisiones booleanas, el rango menor asociado a la condición con el operador “menor o igual” (`<=`) se corresponde con el valor lógico “true”, mientras que el resto representa “false”. Por ejemplo, cuando el árbol debe verificar si el protocolo de red es TCP, la traducción que se realiza consiste en detectar la aparición de una condición del tipo `protocolo_IP_TCP <=`. En este punto no resulta relevante el rango concreto utilizado, ya que, al tratarse de un valor binario, este puede variar en futuros entrenamientos sin afectar a la validez de la traducción.

La adaptación de estas condiciones al lenguaje C requiere una transformación adicional, ya que en este lenguaje la forma de identificar el protocolo de un paquete se basa en los números de protocolo IP. En el caso de TCP, por ejemplo, el campo correspondiente adopta el valor 6, por lo que la condición se traduce finalmente a `ip_proto == 6`. Una lógica similar se aplica a otros protocolos o campos específicos, como las banderas de IP, que también se representan mediante valores numéricos concretos.

La eliminación del campo `TTL`, junto con la generación del nuevo árbol de decisión que se muestra en la figura 3.3, introdujo únicamente una regla adicional de traducción del estilo `<= inf`. En este

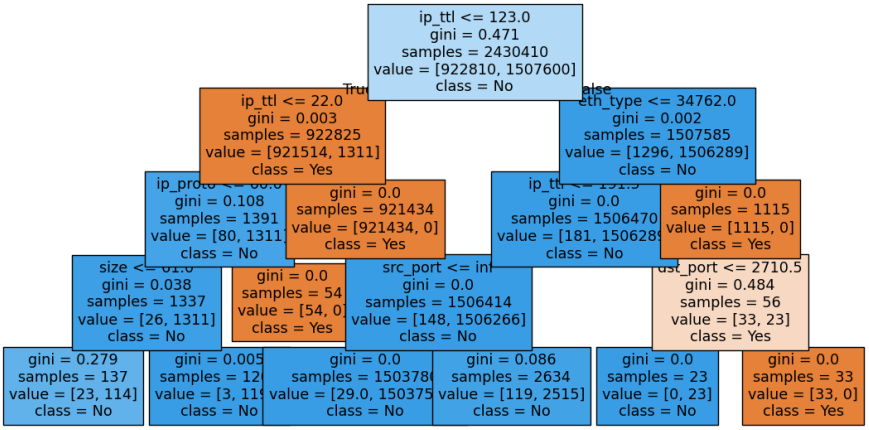

Figura 3.2: Árbol de decisión obtenido de un entrenamiento incluyendo el TTL como dato de entrenamiento.

caso, la condición añadida corresponde a `ip_tot_len <= inf`. Esta traducción, al igual que las anteriores, puede observarse en el fragmento de código 3.3, donde se recogen de forma conjunta las normas aplicadas para transformar las condiciones producidas por el modelo en expresiones válidas en C restringido.

La fase final del proceso de autogeneración consiste en integrar el árbol de decisión en la plantilla intermedia definida con Jinja mencionada anteriormente. Esta plantilla establece la estructura general del programa eBPF/XDP, en la cual se insertan dinámicamente las reglas de clasificación obtenidas durante el entrenamiento. De este modo, se separa la lógica de decisión —que varía en función del modelo— de la infraestructura del programa, que permanece constante.

En términos de contenido, la plantilla define un programa eBPF/XDP diseñado para inspeccionar paquetes de red en el espacio del kernel y tomar decisiones en función de los valores extraídos de sus cabeceras. En primer lugar, incluye las dependencias necesarias: cabeceras de Linux relacionadas con eBPF, Ethernet, IP, TCP y UDP, además de utilidades para la conversión de endianness. Posteriormente, se declara un mapa BPF de tipo `BPF_MAP_TYPE_ARRAY` denominado `packet_count`, con un máximo de dos entradas, destinado a contabilizar los paquetes según la acción aplicada: permitidos o descartados. Los valores asociados a este mapa se almacenan en contadores de tipo `__u64`.

La plantilla incorpora también un conjunto de funciones auxiliares para el parsing de los paquetes. A nivel de red, se extraen campos como la cabecera IP, el TTL, la longitud total, el identificador, las flags y el offset de fragmentación. A nivel de transporte, se recuperan las cabeceras TCP/UDP junto con los puertos de origen y destino. Todas estas funciones validan los límites de memoria mediante los punteros `data` y `data_end`, garantizando que no se produzcan accesos inválidos.

El núcleo del programa lo constituye la función principal `ransomware_tree`, asociada al *hook*

Código 3.3: Fragmento de código de arbolrw.py encargado de traducir el árbol de decisión a C.

```
19  def condiciones_validas(cond : str) -> str:
20      '''
21      Define reglas especiales para ciertos casos
22      '''
23
24      if cond == "src_port <= inf" or cond == "dst_port <= inf":
25          return "ip_proto == IPPROTO_TCP || ip_proto == IPPROTO_UDP"
26
27      if "protocolo_IP_TCP <=" in cond:
28          return "ip_proto == 6"
29      elif "protocolo_IP_TCP =" in cond:
30          return "ip_proto != 6"
31
32      if "protocolo_IP_UDP <=" in cond:
33          return "ip_proto == 17"
34      elif "protocolo_IP_TCP =" in cond:
35          return "ip_proto != 17"
36
37      if "ip_frag_flags_DF <=" in cond:
38          return "ip_frag_flags != 2"
39      elif "ip_frag_flags_DF >" in cond:
40          return "ip_frag_flags == 2"
41
42      if "ip_frag_flags_MF <=" in cond:
43          return "ip_frag_flags != 1"
44      elif "ip_frag_flags_MF >" in cond:
45          return "ip_frag_flags == 1"
46
47      if cond == "ip_tot_len <= inf":
48          return "ip_tot_len != 0"
49
50      if "ip_ttl" in cond:
51          return f"{cond} && ip_ttl != 0"
52
53      # Añadir más reglas si es necesario
54
55      return cond
```

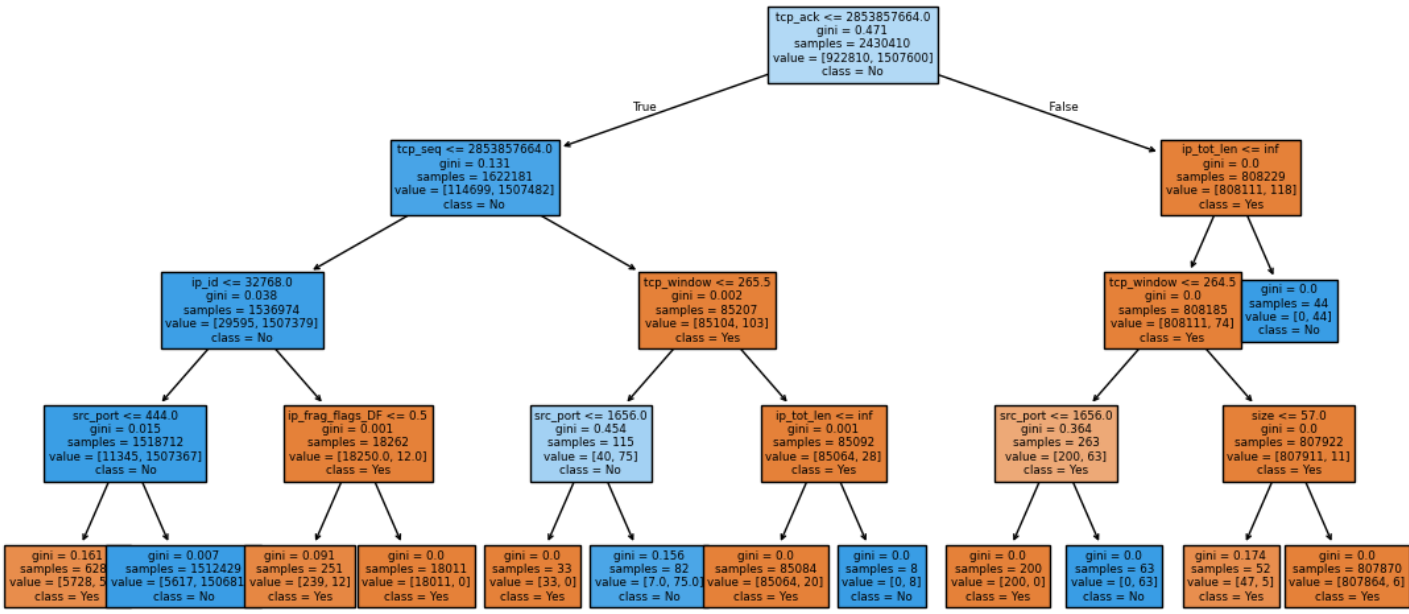

Figura 3.3: Árbol de decisión usado para las pruebas de rendimiento.

XDP. Esta función recibe cada paquete y determina si debe permitirse su paso (`XDP_PASS`) o descartarse (`XDP_DROP`). De forma predeterminada, el tráfico IPv6 se ignora y se pasa sin procesamiento adicional. La función extrae metadatos como el TTL, la longitud total, el identificador, las flags, el offset, así como parámetros de la cabecera TCP (puertos, número de secuencia, ACK, ventana, tamaño, entre otros). En el bloque identificado como `{{ decision_tree }}` se inserta dinámicamente la lógica del árbol de decisión generada durante el entrenamiento, la cual define el criterio de clasificación del tráfico (por ejemplo, ransomware frente a tráfico legítimo). Finalmente, el mapa `packet_count` se actualiza en función de la acción aplicada, permitiendo monitorizar en todo momento el número de paquetes aceptados y descartados.

En conjunto, esta plantilla actúa como un esqueleto flexible que proporciona toda la estructura necesaria para el filtrado de tráfico en el kernel, a la vez que facilita la integración dinámica del árbol de decisión mediante Jinja. Su diseño modular permite mantener un programa ligero, eficiente y adaptable a distintos escenarios de clasificación de tráfico malicioso. La función principal puede verse en el fragmento de código 3.4.

3.5. Compilación y Despliegue

En la presente sección se describen las herramientas y la preparación del entorno necesarias para la ejecución de Mininet, el entrenamiento del árbol de decisión y la implementación del filtro XDP. Para los entornos de ejecución en Python se ha utilizado la versión `3.10.12`, junto con un entorno virtual creado mediante el comando `python3 -m venv`, en el cual se han instalado, a través del gestor de

Código 3.4: Fragmento de código que muestra la función principal XDP en la plantilla Jinja

```
104 // ----------Programa principal ----------
105 SEC("xdp")
106 int ransomware_tree(struct xdp_md *ctx)
107 {
108     void *data_end = (void *)(long)ctx->data_end;
109     void *data = (void *)(long)ctx->data;
110     struct ethhdr *eth = data;
111     int action = XDP_PASS;
112
113     if ((void *)eth + sizeof(*eth) > data_end)
114         return XDP_PASS;
115
116     if (bpf_ntohs(eth->h_proto) == ETH_P_IPV6)
117         action = XDP_PASS;
118
119     // Variables IP
120     __u8 ip_ttl = get_ip_ttl(data, data_end);
121     __u16 ip_tot_len = get_ip_tot_len(data, data_end);
122     __u16 ip_id = get_ip_id(data, data_end);
123     __u8 ip_frag_flags = get_ip_frag_flags(data, data_end);
124     __u16 ip_frag_offset = get_ip_frag_offset(data, data_end);
125
126     // Variables TCP
127     struct tcphdr *tcp = get_tcp_header(data, data_end);
128     __u32 tcp_ack = tcp ? bpf_ntohl(tcp->ack_seq) : 0;
129     __u32 tcp_seq = tcp ? bpf_ntohl(tcp->seq) : 0;
130     __u16 tcp_window = tcp ? bpf_ntohs(tcp->window) : 0;
131     __u32 size = (void *)(long)ctx->data_end -(void *)(long)ctx->data;
132
133     // Puertos
134     __be16 src_port = tcp ? tcp->source : get_src_port(data, data_end);
135     __be16 dst_port = tcp ? tcp->dest : get_dst_port(data, data_end);
136
137     {{ decision_tree }}
138
139     if ( action != XDP_DROP)
140         action = XDP_PASS;
141
142     __u32 key = (action == XDP_DROP) ? 0 : 1;
143     __u64 *count = bpf_map_lookup_elem(&packet_count, &key);
144     if (count)
145         __sync_fetch_and_add(count, 1);
```

paquetes *pip*, todas las dependencias requeridas para el desarrollo del proyecto.

La lista completa de paquetes utilizados puede consultarse en el fichero `requirements.txt` incluido en el repositorio de código. Entre las dependencias más relevantes se encuentran *mininet*, *jinja2* y *scikit-learn*, empleadas respectivamente para el manejo de la API de Mininet y la definición de topologías de red, el rellenado de la plantilla Jinja utilizada para la autogeneración del código XDP, y el entrenamiento del árbol de decisión. Adicionalmente, para poder ejecutar Mininet es necesario disponer previamente del comando *mn* instalado en el sistema, ya que este proporciona las utilidades básicas para la creación y gestión de las topologías de red.

Si bien tanto Mininet como el árbol de decisión resultaron relativamente sencillos de trasladar a otros entornos de ejecución, no ocurrió lo mismo con XDP. Esto se debe a que XDP depende de manera directa de múltiples configuraciones del sistema operativo, así como de las versiones concretas de diversas librerías del sistema y de librerías auxiliares empleadas de forma indirecta por estas. En determinados sistemas operativos, algunas de dichas dependencias se encontraban obsoletas o marcadas como deprecated, lo que complicó la portabilidad y la reproducibilidad del entorno de ejecución, especialmente al migrar entre distribuciones o versiones de kernel distintas.

Para mitigar estas complicaciones, se optó por desarrollar un Makefile dinámico, el cual puede consultarse en el fichero `XDP/arbol_prueba` del repositorio. Este Makefile comienza comprobando la arquitectura del sistema mediante el comando `$(shell uname -m)`, lo que permite asignar posteriormente, a través de estructuras condicionales, el valor adecuado a la variable `BPF_ARCH`. Dicha variable se emplea más adelante para definir correctamente las flags de compilación específicas de la arquitectura.

Antes de iniciar el proceso de compilación propiamente dicho, el Makefile genera de forma dinámica las variables *CFLAGS* y *LIBS*. Para ello, se utilizan los comandos `$(shell pkg-config --cflags libbpf 2>/dev/null)` y `$(shell pkg-config --libs libbpf 2>/dev/null) -ldl`, respectivamente. Este enfoque resulta especialmente relevante, ya que el código ha sido adaptado para ser compatible con distintas versiones de `libbpf`, la librería encargada de proporcionar la API de eBPF/XDP, facilitando así la portabilidad y la correcta compilación en entornos heterogéneos.

El proceso de despliegue de la arquitectura sigue una secuencia bien definida. En primer lugar, se debe activar el entorno virtual de Python, garantizando así que todas las dependencias necesarias se encuentren disponibles en el entorno de ejecución. A continuación, se ejecuta el script en Python encargado del entrenamiento del árbol de decisión y de la posterior generación del código XDP, proceso que culmina con la creación del código fuente mediante el uso de la plantilla Jinja.

Una vez generado el código XDP, se procede a ejecutar el Makefile, el cual, apoyándose en herramientas como *gcc*, *Clang* y *LLVM*, compila los distintos componentes del sistema, generando el ejecutable del programa de espacio de usuario (`xdp_usr`), el fichero objeto correspondiente al programa del kernel (`xdp_kern.o`) y el bytecode asociado a `xdp_kern`. Finalmente, manteniendo

activo el entorno virtual de Python, se lanza mediante un script Python el entorno de Mininet, desde el cual se pueden ejecutar las distintas pruebas y experimentos definidos en el presente trabajo.

4

Experimentos y Resultados

4.1. Entorno de prueba Mininet

Para el diseño de esta emulación se tomó como referencia la familia de SmartNICs NVIDIA BlueField-2 [23], ya que en una fase inicial del proyecto se contempló la posibilidad de utilizar una SmartNIC real de dicha familia —concretamente la NVIDIA A30X Converged Accelerator—, opción que finalmente no pudo llevarse a cabo debido a una avería del hardware disponible.

El modo de operación previsto para dicha SmartNIC era el denominado *loopback* [24], en el cual el tráfico recibido por la primera interfaz de red es redirigido de forma obligatoria hacia la segunda interfaz, pasando previamente por la DPU, donde se ejecuta el programa XDP acoplado al driver de red 4.1. De este modo, los paquetes son procesados y devueltos al flujo de datos sin intervención de la CPU del host. Este esquema de funcionamiento es habitual en tareas de seguridad, como la implementación de firewalls o sistemas de inspección de tráfico, ya que permite ejecutar las funciones de filtrado de forma aislada y reducir la carga computacional del sistema anfitrión.

La adopción de este modelo resulta especialmente adecuada para el presente trabajo, dado que el árbol de decisión implementado puede considerarse un subtipo de firewall especializado. Al ejecutar la lógica de clasificación directamente sobre la ruta de datos y fuera del host, se favorece un diseño más eficiente y alineado con los principios de procesamiento en línea que caracterizan a las SmartNIC modernas.

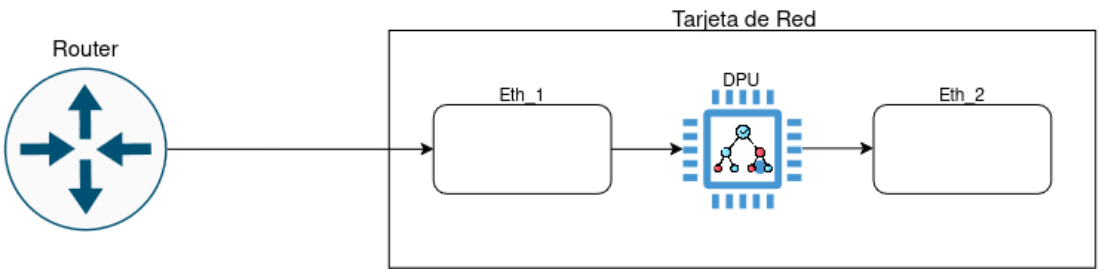

Figura 4.1: Diseño Real de la topología de red

Teniendo en cuenta que el entorno de referencia real consiste en una tarjeta de red con dos puertos

a nivel de enlace, se decidió simplificar la topología de Mininet con el objetivo de reproducir de forma más fiel el comportamiento de dicho escenario. La red utilizada consta únicamente de dos hosts conectados a un único switch, que actúa como punto de interconexión y emulación del camino de datos, eliminando la necesidad de routers adicionales o interfaces suplementarias, tal y como se muestra en la figura 4.2. Esta simplificación permite reducir la complejidad de la topología y minimizar factores externos que podrían afectar a las mediciones, como latencias artificiales introducidas por componentes adicionales de la red virtual.

Además, esta configuración contribuye a disminuir la variabilidad en los resultados experimentales y a reducir efectos secundarios propios de la emulación, como el tráfico de control asociado a la resolución de direcciones MAC mediante ARP. Al aproximarse al comportamiento funcional de un entorno con DPU en modo *inline*, la topología proporciona un marco adecuado para evaluar el rendimiento de la red bajo la influencia del filtro basado en árbol de decisión. El siguiente paso en el proceso consiste en la realización de pruebas de rendimiento utilizando la herramienta iperf, lo que permitirá cuantificar de manera más precisa el impacto del filtrado sobre el tráfico de red.

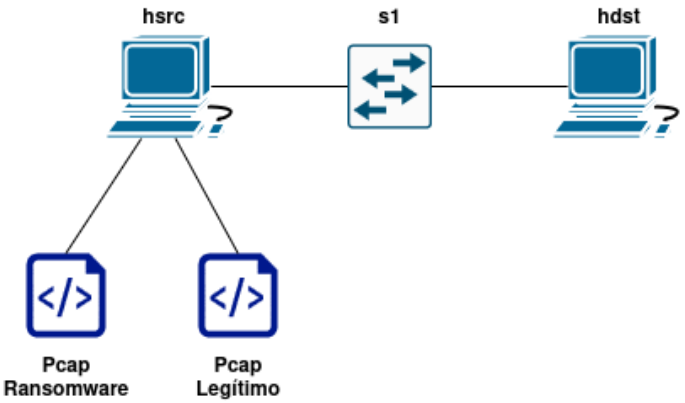

Figura 4.2: Diseño de topología de red usada en Mininet para pruebas.

En los fragmentos de código 4.1 y 4.2 se muestran extractos de los ficheros redarbolrw.py y redarbolrw1.py, correspondientes a la definición de la topología preliminar y de la topología definitiva, respectivamente.

4.2. Herramientas

En la presente sección se describen las herramientas empleadas tanto para la generación de tráfico como para la obtención de métricas asociadas. Si bien todas ellas persiguen el mismo objetivo general —evaluar el rendimiento de la red bajo distintas condiciones de carga—, difieren en el tipo de tráfico que generan y en su origen. Esta complementariedad justifica la utilización de las tres, ya que permite llevar a cabo un análisis más completo y ofrecer una visión más precisa del comportamiento del sistema.

Código 4.1: Fragmento de código que define la topología de red usada en las pruebas preliminares.

```
22  class MyTopo(Topo):
23      """
24      Topología compuesta por:
25        -hsrc: host origen con IP 10.0.1.1/24
26        -hdst: host destino con IP 10.0.2.1/24
27        -s1 y s2: switches conectados a hsrc y hdst respectivamente
28        -r1: router con dos interfaces, r1-eth1 en 10.0.1.254/24 y r1-eth2 en 10.0.2.254/24,
29              encargándose de la interconexión entre las dos subredes.
30      """
31      def build(self):
32          # Crear hosts
33          hsrc = self.addHost('hsrc', ip='10.0.1.1/24')
34          hdst = self.addHost('hdst', ip='10.0.2.1/24')
35
36          # Crear switches
37          s1 = self.addSwitch('s1')
38          s2 = self.addSwitch('s2')
39
40          # Agregar router: a priori asignamos la IP de la red 10.0.1.0/24;
41          # luego se asigna la segunda interfaz manualmente en el enlace a s2.
42          r1 = self.addNode('r1', cls=MyRouter, ip='10.0.1.254/24')
43
44          # Conectar hsrc al switch s1
45          self.addLink(hsrc, s1)
46
47          # Conectar el router a s1: en la red 10.0.1.0/24 (puerta de enlace para hsrc)
48          self.addLink(s1, r1, intfName2='r1-eth1', params2={'ip': '10.0.1.254/24'})
49
50          # Conectar el router a s2: en la red 10.0.2.0/24 (puerta de enlace para hdst)
51          self.addLink(s2, r1, intfName2='r1-eth2', params2={'ip': '10.0.2.254/24'})
52
53          # Conectar hdst al switch s2
54          self.addLink(hdst, s2)
```

Código 4.2: Fragmento de código que define la topología de red usada en las pruebas finales.

```
8  import time
9
10
11 class SimpleTopo(Topo):
12     def build(self):
13         hsrc = self.addHost(
14             'hsrc',
15             ip='10.0.1.1/24',
16             mac='00:00:00:00:01:01'
17         )
18
19         hdst = self.addHost(
20             'hdst',
21             ip='10.0.1.2/24',
22             mac='00:00:00:00:02:02'
23         )
24
25         s1 = self.addSwitch('s1', failMode='standalone')
```

4.2.1. TCPReplay

TCPReplay es una herramienta diseñada para el reenvío de paquetes de red previamente capturados mediante utilidades como tcpdump o Wireshark. Su principal funcionalidad consiste en reproducir dichos paquetes, bien respetando la velocidad original con la que fueron capturados, o bien a una velocidad definida por el usuario. En este último caso, la herramienta permite ajustar de manera explícita el ritmo de envío, ya sea en función de paquetes por segundo (pps) o de la tasa de transmisión expresada en megabits por segundo (Mbps) [25].

TCPReplay aporta dos ventajas clave para la evaluación experimental. En primer lugar, permite reproducir tráfico real contenido en ficheros PCAP, lo que posibilita realizar ensayos más representativos que los basados únicamente en tráfico sintético —por ejemplo, incluyendo protocolos distintos de TCP/UDP (ICMP, ARP, SMB, etc.) y secuencias de paquetes reales—. En segundo lugar, ofrece un control fino sobre la velocidad de reproducción (tanto en paquetes por segundo como en Mbps y otras modalidades), lo que facilita la emulación de distintos escenarios de carga de red (desde tráfico esporádico hasta condiciones de estrés). Estas capacidades hacen de TCPReplay una herramienta especialmente adecuada para validar el comportamiento del filtro XDP frente a tráfico heterogéneo y para estudiar su rendimiento bajo diferentes perfiles de carga.

4.2.2. Script de Reenvio

Una limitación de *tcpreplay* cuando se utiliza de forma aislada es que no permite emular de manera realista un escenario de ataque, ya que en este tipo de situaciones el sistema suele recibir simultánea-

mente tráfico benigno y tráfico malicioso. Por este motivo, se desarrolló un script propio —se puede encontrar en `Mininet/script_reenvio.py`— que hace uso de *tcpreplay* con el objetivo de reproducir un escenario más representativo, combinando ambos tipos de tráfico en una misma ejecución.

El funcionamiento del script se divide en dos fases. En primer lugar, se realiza una fase de preparación en la que se genera un fichero PCAP auxiliar, el cual se elimina una vez finalizada la prueba. Dicho fichero se construye a partir de fragmentos de los PCAP legítimo y malicioso proporcionados como entrada. Durante esta fase, la selección de paquetes se realiza de manera aleatoria: en cada iteración se escoge al azar si se extraerán paquetes del PCAP legítimo o del malicioso, tras lo cual se leen y escriben 100 paquetes consecutivos de la captura seleccionada. Este proceso se repite hasta alcanzar el número total de paquetes que se pretende transmitir, calculado como el producto de la tasa objetivo de paquetes por segundo y la duración del experimento, añadiéndose adicionalmente un 20 % de margen para garantizar que el fichero auxiliar contenga suficientes paquetes ante posibles variaciones o imprevistos durante la reproducción.

Una vez generado el PCAP auxiliar, se ejecuta la segunda fase, consistente en reenviar dicho fichero a la tasa y duración especificadas como parámetros de entrada por el usuario, utilizando nuevamente *tcpreplay*. Inicialmente se consideró implementar esta reproducción directamente mediante *Scapy*, seleccionando los paquetes de forma aleatoria en tiempo de ejecución sin generar un PCAP previo. No obstante, esta aproximación se descartó debido a que, en pruebas preliminares, no fue posible alcanzar tasas de paquetes por segundo suficientemente elevadas para el propósito experimental del presente trabajo.

4.2.3. TCPDump

Tanto TCPReplay como el script desarrollado en este trabajo no disponen de funcionalidades propias para medir el rendimiento de la red, limitándose únicamente al reenvío de paquetes. Por este motivo, resulta necesario complementar su uso con herramientas de captura de tráfico, como TCPDump, que permiten registrar los paquetes transmitidos y analizar posteriormente métricas relevantes como el *throughput*, la latencia o la pérdida de paquetes [26].

TCPDump es una herramienta de línea de comandos que permite capturar y visualizar paquetes de red en tiempo real. Es capaz de filtrar el tráfico en función de distintos criterios, como direcciones IP, puertos o protocolos, y de guardar las capturas en ficheros para su análisis posterior. Además, TCPDump permite examinar cabeceras de protocolos como Ethernet, IP, TCP y UDP, proporcionando información detallada sobre el flujo de datos entre hosts, lo que lo convierte en un recurso fundamental para evaluar cómo afectan mecanismos de filtrado, como el árbol de decisión XDP, al rendimiento de la red.

4.3. Métricas Relevantes

Para poder analizar el rendimiento del sistema, tanto con el árbol de decisión activo como con este inactivo, es necesario centrarse en una serie de métricas relevantes. Estas métricas permitirán cuantificar de manera objetiva el impacto que el árbol de decisión tiene sobre el comportamiento general del sistema, proporcionando una base sólida para la comparación entre ambos escenarios.

La primera métrica a considerar es el rendimiento o *throughput*, el cual mide la cantidad de información útil que se recibe por segundo. Dado que se trata de un entorno de red, el estándar es expresar esta métrica en bits por segundo (bps) en lugar de utilizar unidades como bytes por segundo u otras equivalentes, ya que permite una representación más precisa y ampliamente aceptada en el ámbito de las comunicaciones de red.

$$\text{Throughput (bps)} = \frac{8 \times B_{\text{recibidos}}}{T} \tag{4.1}$$

Además de medir el rendimiento en bits por segundo, resulta de interés cuantificar también el número de paquetes por segundo (PPS) que el sistema es capaz de procesar. Aunque a primera vista esta métrica pueda parecer redundante respecto a la anterior, ambas reflejan aspectos distintos del comportamiento del sistema. En efecto, una misma cantidad total de información puede estar dividida en un número diferente de paquetes, y cada uno de ellos requiere una serie de operaciones de procesamiento que deben ejecutarse de manera independiente –como la lectura de cabeceras, la evaluación de condiciones o el acceso a estructuras de datos internas–. Por este motivo, el rendimiento medido en paquetes por segundo permite identificar posibles cuellos de botella asociados al manejo intensivo de paquetes pequeños y evaluar con mayor precisión el impacto del árbol de decisión sobre la carga computacional del sistema.

$$\text{PPS} = \frac{N_{\text{paquetes}}}{t_{\text{fin}} - t_{\text{inicio}}} \tag{4.2}$$

Otra métrica fundamental en un entorno de red es el porcentaje de pérdida de paquetes (*packet loss rate*). Este parámetro resulta esencial para evaluar la fiabilidad y eficiencia de un sistema, ya que un alto rendimiento en términos de bits por segundo (bps) no necesariamente implica un funcionamiento óptimo si el sistema no es capaz de procesar o asimilar correctamente todo el tráfico recibido. En este sentido, un porcentaje elevado de pérdida puede indicar cuellos de botella, saturación del sistema o limitaciones en la capacidad de procesamiento del tráfico, comprometiendo la eficacia global de la solución.

$$\text{Pérdida (\%)} = 100 \times \frac{N_{\text{enviados}} - N_{\text{recibidos}}}{N_{\text{enviados}}} \tag{4.3}$$

No obstante, la interpretación de esta métrica requiere un análisis diferenciado cuando el filtro ba-

sado en el árbol de decisión se encuentra activo. En este caso, parte de los paquetes son descartados de manera intencionada por el propio filtro al ser clasificados como potenciales amenazas. Por tanto, resulta fundamental distinguir entre los paquetes perdidos debido a posibles cuellos de botella generados al añadir una nueva capa de procesamiento y aquellos paquetes que son eliminados de manera deliberada por el filtro como parte de su funcionamiento normal. Esta distinción permite evaluar con precisión tanto la eficiencia del sistema como la eficacia del mecanismo de filtrado implementado.

$$\text{Pérdida Real}(\,\%) = 100 \times \frac{N_{\text{enviados}} - N_{\text{recibidos}} - N_{\text{filtrados}}}{N_{\text{enviados}}} \tag{4.4}$$

Existe además la métrica denominada *goodput*, la cual representa el rendimiento efectivo de una muestra de tráfico expresado en paquetes por segundo (PPS), considerando únicamente aquellos paquetes que son entregados con éxito y excluyendo las pérdidas producidas durante la transmisión.

$$\text{Goodput} = \text{pps\_medidos} \times (1 - \frac{\text{porcentaje\_perdida\_real}}{100}) \tag{4.5}$$

Por último, otras dos métricas relevantes en el análisis experimental son el porcentaje de uso de la CPU y el porcentaje de uso de la memoria RAM. Estas métricas permiten evaluar el impacto del filtro sobre los recursos computacionales del sistema y determinar si su ejecución introduce un cuello de botella o un factor limitante asociado al consumo de hardware, especialmente en escenarios de alta carga de tráfico.

4.4. Metodología Seguida

Antes de llevar a cabo los experimentos de rendimiento, el primer paso consiste en verificar si el árbol de decisión entrenado presenta una tasa de acierto real lo suficientemente adecuada como para considerar que, con un entrenamiento más exhaustivo, podría emplearse como un filtro efectivo. En este trabajo se estableció como umbral mínimo deseable una tasa de acierto real del 60 %, dado que el objetivo principal no es evaluar la eficacia del modelo, sino analizar el impacto que dicho filtro puede tener sobre el rendimiento del sistema.

Las pruebas realizadas en este ámbito, como la que se muestra en la figura 4.3, han arrojado una tasa de acierto situada entre el 65 % y el 70 %. Esta tasa se ha calculado como el cociente entre el número de paquetes maliciosos correctamente rechazados y el número total de paquetes maliciosos enviados, multiplicado por cien. Con ello se confirma que el árbol utilizado posee una calidad suficiente para servir como base en los experimentos de rendimiento desarrollados en este trabajo.

El experimento consiste en generar tráfico de red a una tasa determinada durante un intervalo de tiempo fijo, ejecutando primero las pruebas con el filtro desactivado y, posteriormente, repitiéndolas con el filtro XDP activado. En concreto, se realizan iteraciones con tasas comprendidas entre 64 (2^6)

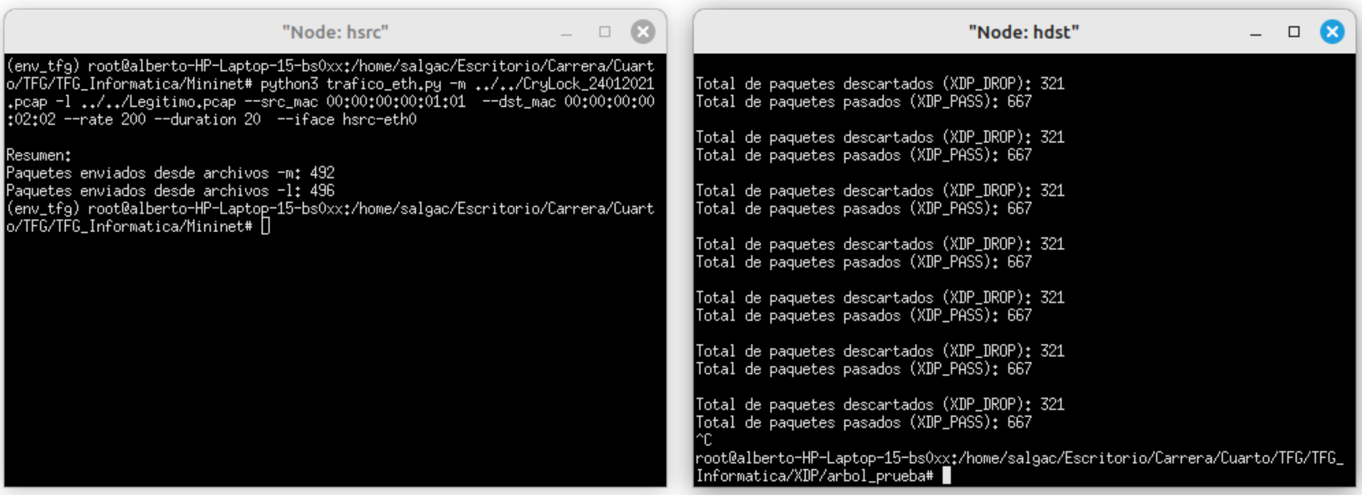

Figura 4.3: Prueba realizada para comprobar la tasa de acierto real del árbol de decisión.

y 16.384 (2^{14}) paquetes por segundo, manteniendo una duración de 10 segundos por ejecución. Este procedimiento se aplica de forma independiente para cada herramienta empleada —cuyo papel y objetivo se describirán en párrafos posteriores—. Tras cada iteración, se registran las métricas consideradas más relevantes, con la excepción del rendimiento en bits por segundo (bps), cuya obtención y tratamiento se detalla más adelante.

Para ejecutar estas pruebas de manera reproducible, se empleó un único script en Python encargado de automatizar tanto el despliegue del entorno como la ejecución completa del conjunto de iteraciones. En primer lugar, el script inicializa la topología en Mininet y obtiene los identificadores de proceso (PID) asociados a los hosts *hsrc* y *hdst*, lo que permite lanzar comandos directamente sobre cada host emulado. A continuación, se ejecuta el bloque de iteraciones con el filtro XDP desactivado, incorporando un breve *sleep* entre ejecuciones. Una vez finalizado este bloque, se introduce una pausa más prolongada con el objetivo de reducir el posible efecto residual de las pruebas previas sobre las siguientes mediciones. Posteriormente, el script compila y activa el filtro XDP y repite el mismo rango de iteraciones bajo condiciones equivalentes.

Finalmente, tras cada iteración —tanto con el filtro activo como desactivado— el script vuelca las métricas registradas a un fichero CSV asociado a dicha prueba. Los resultados completos obtenidos durante la experimentación pueden consultarse en el directorio `ResultadoExperimento` del repositorio.

Con el fin de validar los resultados obtenidos en la experimentación y poder extraer conclusiones fundamentadas, se han elaborado diversas gráficas orientadas a responder a una serie de cuestiones específicas. Estas representaciones, junto con el análisis asociado, pueden agruparse en cuatro bloques principales: capacidad y escalabilidad del envío, pérdidas de paquetes, actividad del filtro y coste del sistema.

Comenzando por el primer bloque, relativo a la capacidad y escalabilidad del envío, se han seleccionado las figuras `pps_measured vs pps_target` (4.4) y `goodput_pps vs pps_target` (4.5). La primera permite comprobar si el entorno experimental es capaz de generar y transmitir el tráfico solicitado de forma consistente, evaluando el grado de correspondencia entre la tasa objetivo y la tasa efectivamente medida. La segunda, siguiendo la misma línea, permite analizar el rendimiento útil (goodput) en función de la tasa esperada, estimando la tasa efectiva de entrega correcta al considerar únicamente los paquetes entregados sin pérdida real. En conjunto, ambas representaciones permiten validar la consistencia del entorno de pruebas y detectar posibles fenómenos de saturación cuando la tasa alcanzada deja de crecer proporcionalmente a medida que aumenta la carga.

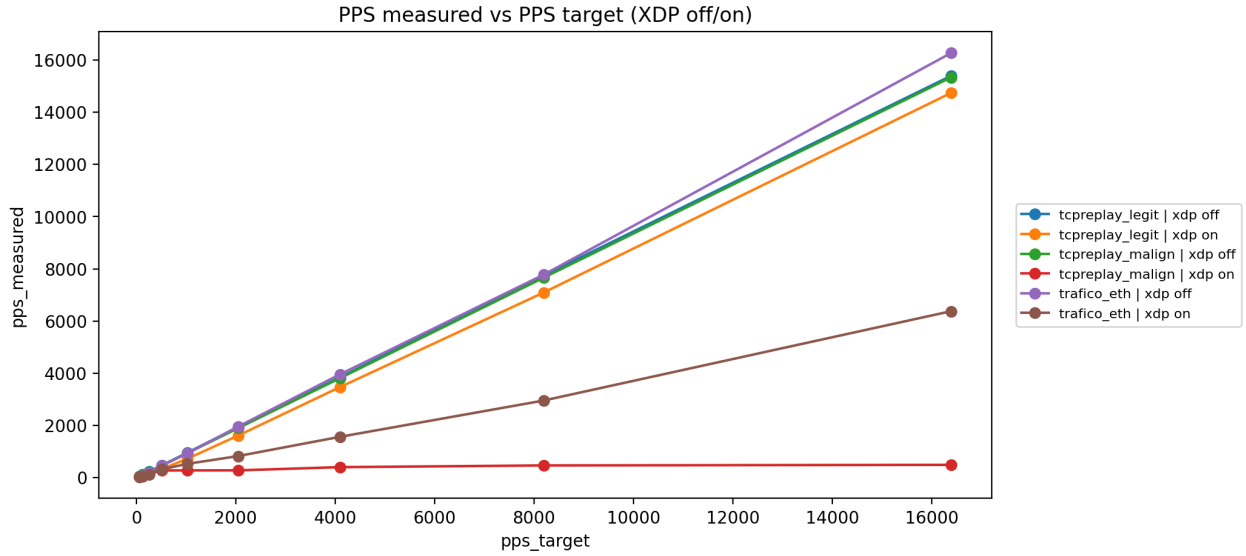

Figura 4.4: Gráfica que muestra la comparativa entre los pps medidos y los esperados.

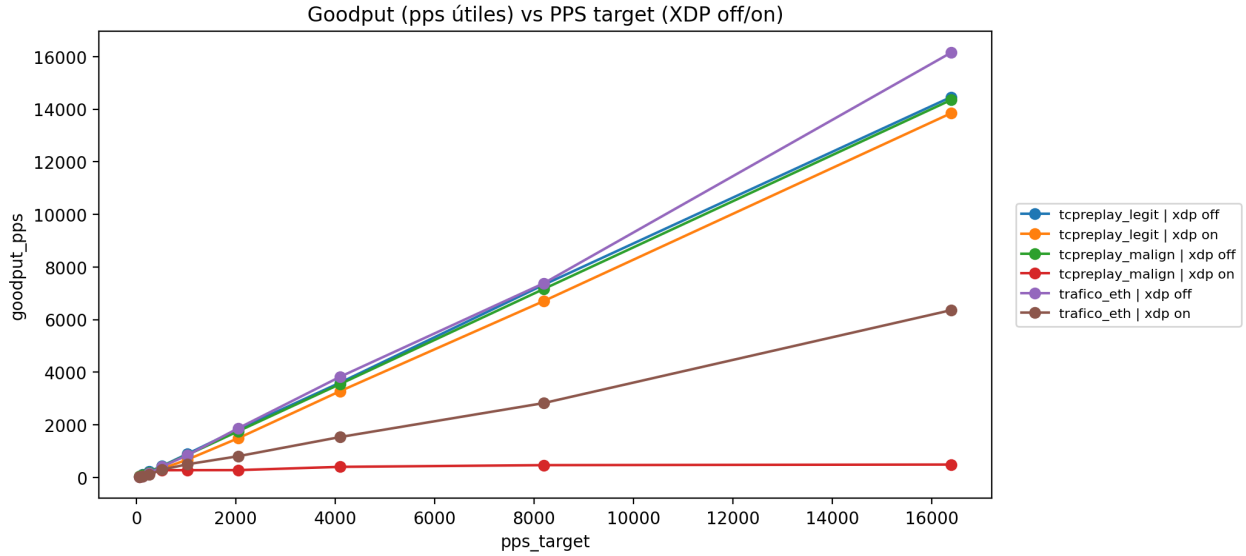

Figura 4.5: Gráfica que muestra la comparativa entre rendimiento útil y los pps esperados.

En el segundo bloque, centrado en el análisis de pérdidas, se han seleccionado las figuras `lost_percent vs` (4.6) y `lost_real_percent vs pps_target` (4.7). La primera representa el porcentaje de pérdida medido globalmente durante cada iteración, incluyendo tanto pérdidas debidas a limitaciones del entorno de transmisión como paquetes descartados intencionadamente por el filtro cuando este está activo. La segunda, por su parte, muestra el porcentaje de pérdida real, calculado descontando del total los paquetes asociados al filtrado (`XDP_DROP`), lo que permite cuantificar la degradación efectiva de la comunicación debida a pérdidas no intencionadas. Consideradas conjuntamente, ambas figuras permiten diferenciar entre una pérdida aparente elevada provocada por el funcionamiento normal del filtro y una pérdida real asociada a degradación de rendimiento, proporcionando una interpretación más precisa del impacto del filtrado sobre el rendimiento de la red.

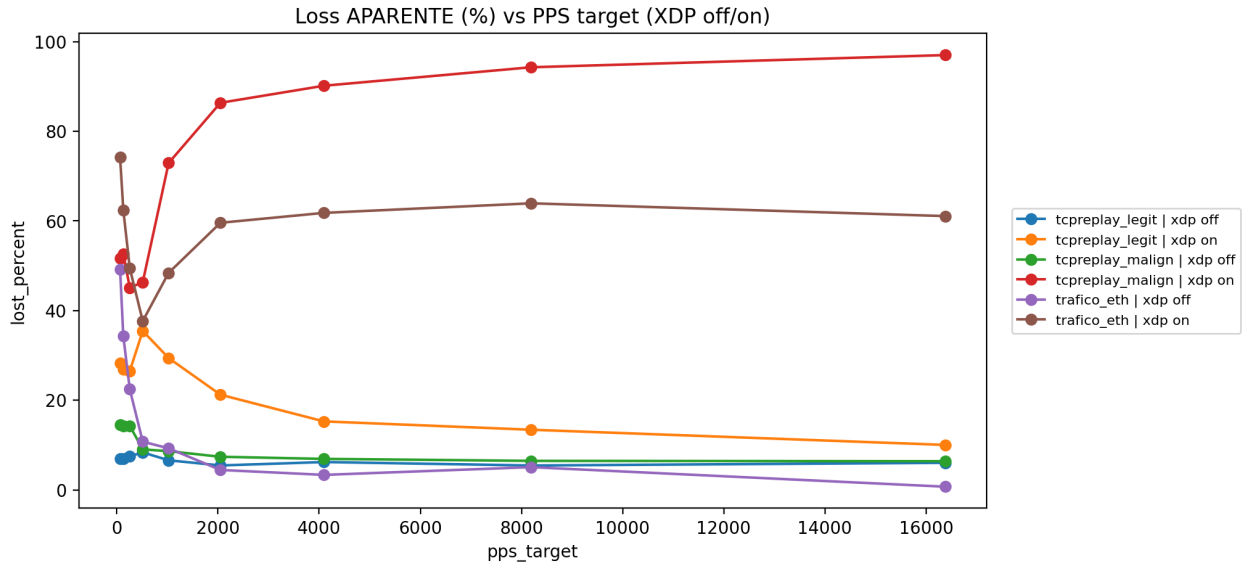

Figura 4.6: Gráfica que muestra la comparativa entre el porcentaje de pérdida aparente y los pps esperados.

En el tercer bloque, orientado a caracterizar la actividad del filtro, se han seleccionado las figuras `paquetes_filtrados vs pps_target` (4.8) y `loss_gap vs pps_target` (4.9). La primera muestra el número de paquetes descartados por el filtro, correspondiente al contador de eventos `XDP_DROP`, y permite comprobar de forma directa que el mecanismo de filtrado se encuentra actuando durante la ejecución del experimento. La segunda representa la diferencia entre la pérdida aparente y la pérdida real, cuantificando el porcentaje de pérdida observado que se explica por el filtrado intencionado y no por degradación de la comunicación. En conjunto, ambas figuras permiten interpretar correctamente el incremento de pérdidas observado con el filtro activado, distinguiendo entre descartar tráfico de forma deliberada y pérdidas atribuibles a limitaciones del entorno de red o del generador de tráfico.

Por último, el cuarto bloque analiza el coste computacional asociado al experimento, empleando las figuras `cpu_usage_percent vs pps_target` (4.10) y `mem_usage_percent vs pps_target`

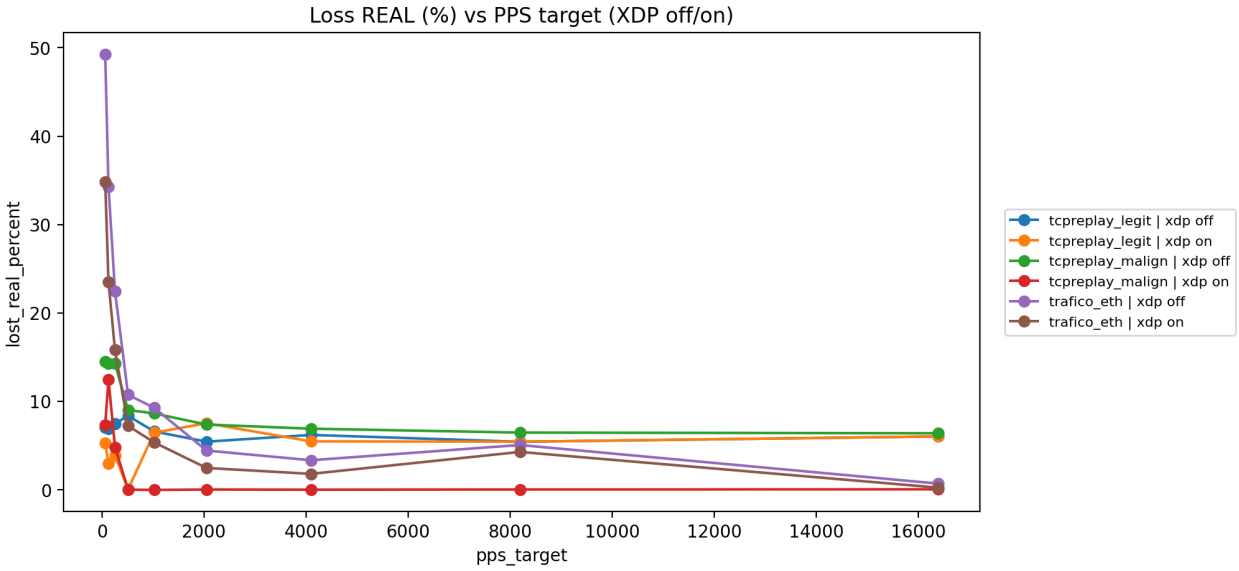

Figura 4.7: Gráfica que muestra la comparativa entre el porcentaje de pérdida real y los pps esperados.

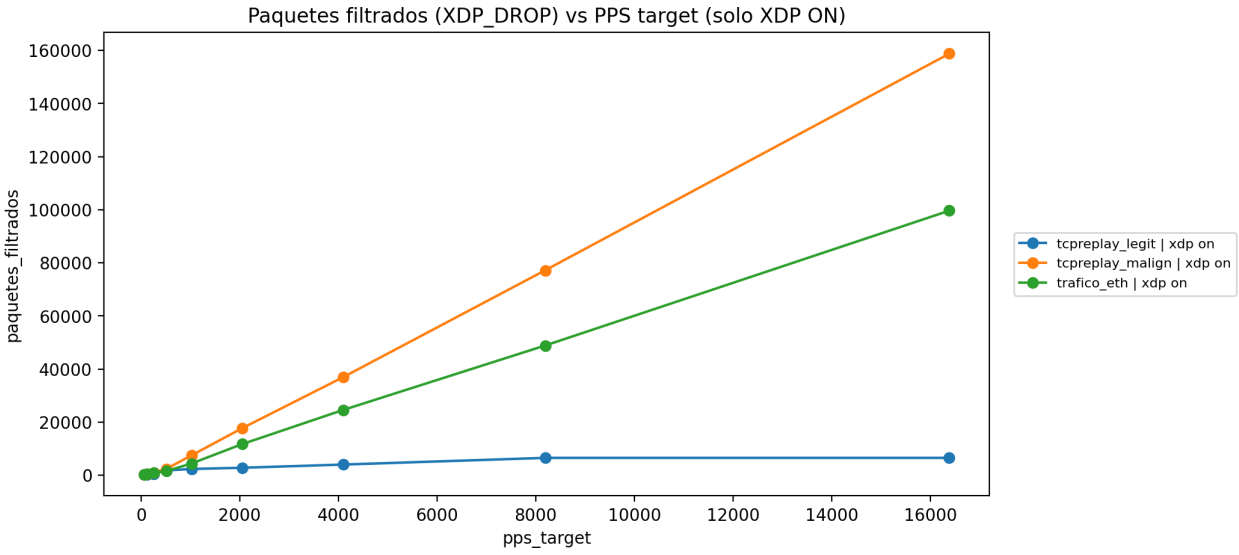

Figura 4.8: Gráfica que muestra la comparativa entre los paquetes filtrados y los pps esperados.

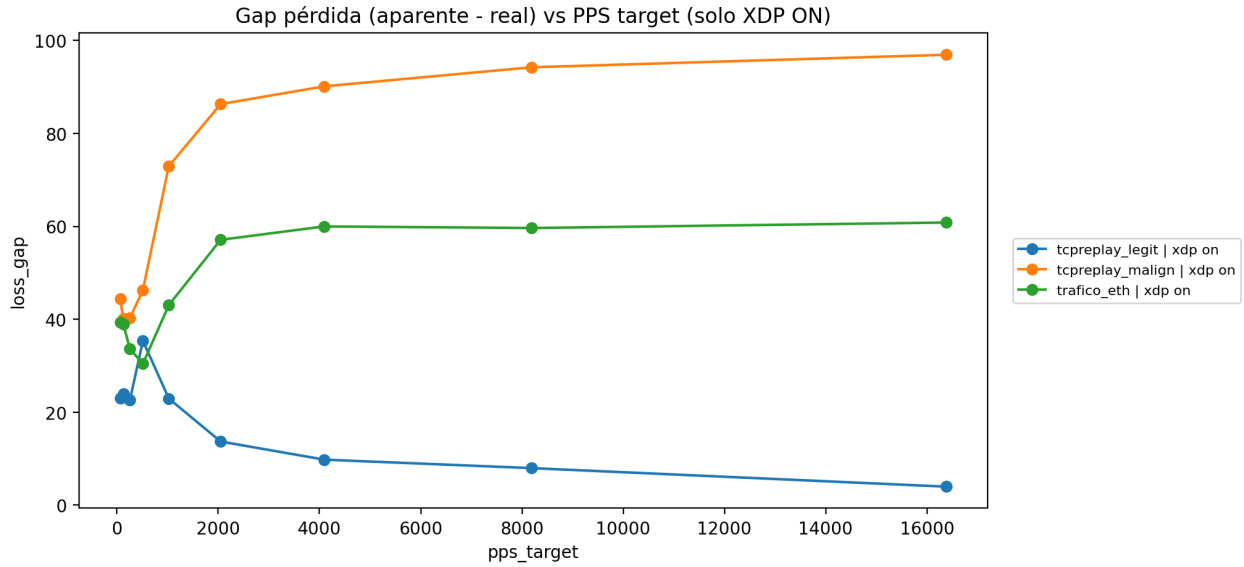

Figura 4.9: Gráfica que muestra la diferencia entre la pérdida aparente y la pérdida real (gaploss) respecto a los pps esperados.

(4.11). Estas representaciones permiten observar la evolución del consumo de recursos a medida que aumenta la carga de tráfico, comparando el comportamiento con el filtro desactivado y activado. De este modo, se evalúa si el filtrado introduce una sobrecarga sostenida que pudiera constituir un cuello de botella en el sistema, o si las variaciones observadas responden principalmente a la propia generación y transmisión del tráfico en el entorno de pruebas. Este análisis complementa las métricas de rendimiento de red, proporcionando una visión global del impacto del filtrado tanto en términos de entrega de paquetes como de uso de recursos.

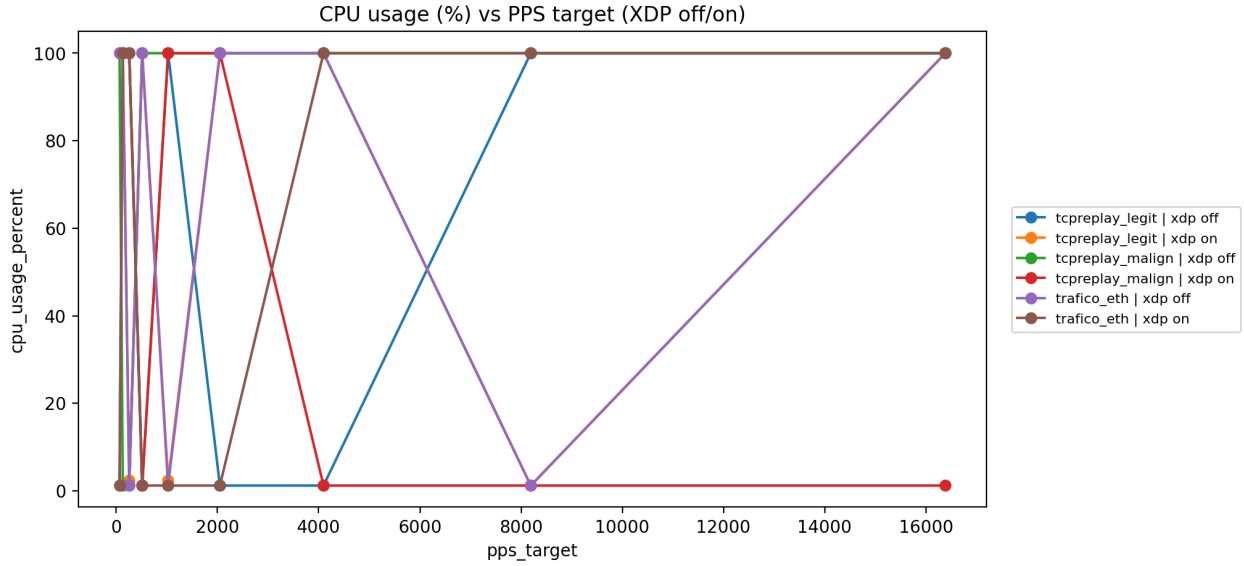

Figura 4.10: Gráfica que muestra la comparativa entre el uso de la CPU y los pps esperados.

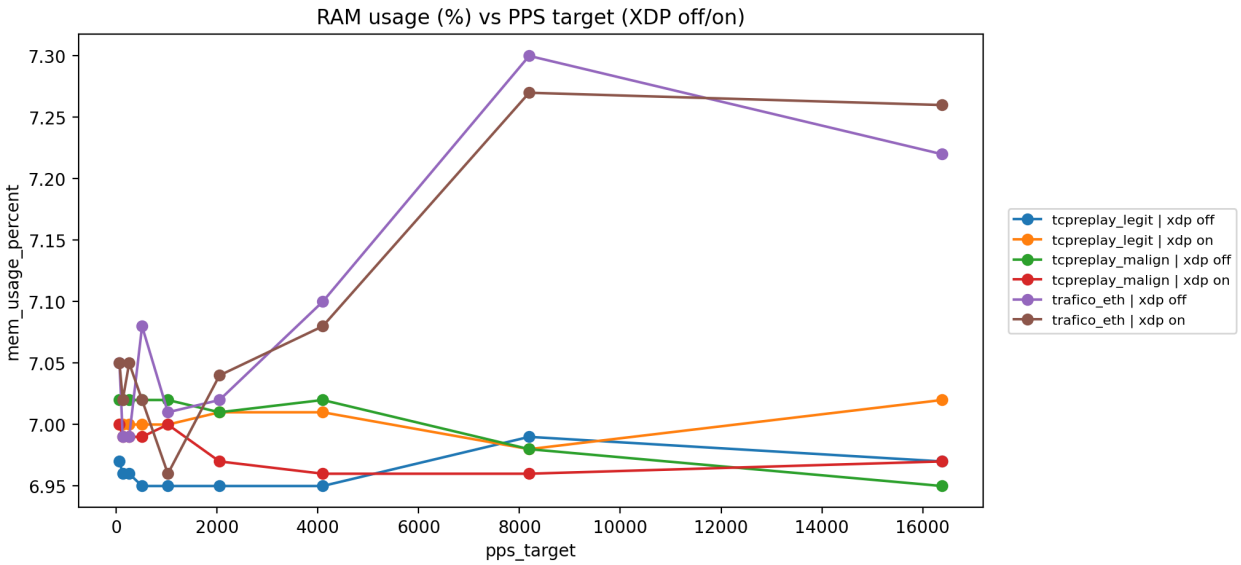

Figura 4.11: Gráfica que muestra la comparativa entre el uso de la memoria RAM y los pps esperados.

Aunque el *throughput* en bits por segundo es una métrica habitual en la evaluación del rendimiento de red, en este trabajo no se incluyó su medición experimental directa. Esto se debe, en primer lugar, a que dicha métrica no puede aplicarse de manera sencilla en escenarios basados en tcpreplay, ya que no se dispone del tamaño en bits de cada paquete en el momento de su reenvío. Calcular este valor dinámicamente durante la transmisión introduciría un overhead adicional suficientemente elevado como para alterar de forma apreciable la propia métrica que se pretende medir.

Como alternativa, se planteó el uso de iperf para la obtención del *throughput* en bits por segundo. Sin embargo, durante las pruebas preliminares se observó que, con el árbol de decisión activo en XDP, las mediciones proporcionadas por esta herramienta resultaban gravemente distorsionadas y poco representativas del comportamiento real del sistema. Por este motivo, se decidió no incluir esta métrica en los experimentos finales y centrar el análisis en métricas basadas en paquetes por segundo, pérdidas de paquetes y consumo de recursos, que permiten una evaluación más fiable del impacto del filtrado.

5

Conclusiones y Trabajos Futuros

5.1. Conclusiones

En esta sección se presentan las conclusiones del presente trabajo, con el objetivo de valorar si, tal y como se estableció en los objetivos planteados, el uso de clasificadores sencillos, como los árboles de decisión, aplicados como filtros preliminares en tareas de ciberseguridad, puede introducir un impacto apreciable sobre el rendimiento de la red. Para ello, se consideran los distintos escenarios de experimentación definidos y las métricas seleccionadas, evaluando la viabilidad de este enfoque desde un punto de vista tanto técnico como práctico.

En primer lugar, se valida que el entorno experimental utilizado es representativo y suficientemente robusto para analizar mecanismos de filtrado en línea, descartando que los efectos observados se deban a deficiencias de la infraestructura de pruebas. Esto puede comprobarse a partir de la coherencia entre la tasa objetivo y la tasa medida de paquetes por segundo, tal y como se muestra en la figura 4.4. Asimismo, se confirma que la distinción entre pérdida aparente y pérdida real resulta esencial para interpretar correctamente el comportamiento del filtro XDP, evitando atribuir al sistema pérdidas que forman parte de su funcionamiento normal.

En este sentido, los resultados confirman que el incremento observado en la pérdida global de paquetes cuando el filtro está activo no se debe a una degradación del rendimiento del sistema, sino al descarte intencionado de tráfico clasificado como malicioso. Esta conclusión se refuerza al analizar la pérdida real, representada en la figura 4.7, donde se observa que el impacto sobre las pérdidas no intencionadas se mantiene contenido. De forma coherente con lo anterior, el mantenimiento de un rendimiento útil comparable en ambos escenarios (figura 4.5) permite concluir que la lógica de clasificación introducida no constituye un cuello de botella significativo en la ruta de datos. Adicionalmente, el análisis de consumo de recursos muestra que el coste computacional asociado al filtrado es reducido, manteniéndose estable tanto en uso de CPU como de memoria (figuras 4.10 y 4.11), lo que refuerza su idoneidad para despliegues prolongados.

En conjunto, los resultados experimentales permiten concluir que la implementación de un filtro basado en árboles de decisión mediante XDP es viable desde el punto de vista del rendimiento. El

sistema mantiene niveles comparables de rendimiento útil, pérdida real y consumo de recursos tanto con el filtro activo como desactivado, incluso a tasas elevadas de paquetes por segundo. Por tanto, estos resultados respaldan el uso de clasificadores sencillos como filtros preliminares de seguridad en la ruta de datos, especialmente en entornos donde la latencia y el rendimiento son críticos, como SmartNICs o arquitecturas *inline*.

5.2. Futuros Trabajos

A partir del presente trabajo se pueden plantear diversas líneas de investigación. En primer lugar, sería interesante reproducir estos experimentos en una Smart NIC real, con el objetivo de evaluar si los árboles de decisión implementados mediante XDP constituyen una opción viable para ser utilizados como filtro preliminar en entornos de red de alto rendimiento.

Una segunda línea de investigación consistiría en realizar pruebas similares empleando árboles de decisión basados en la tecnología DPDK en lugar de XDP, con el fin de comparar la eficiencia y la idoneidad de ambas tecnologías para el filtrado de tráfico en tiempo real.

Finalmente, y como una línea más avanzada, se podría explorar el impacto de la utilización de múltiples árboles de decisión, conformando un *random forest*, sobre el rendimiento del sistema. Esta aproximación permitiría evaluar si compensa la mayor complejidad computacional frente a las ventajas de disponer de árboles especializados para distintas características del tráfico, posibilitando así la creación de un filtro más especializado contra ransomware o, incluso, un sistema más generalista capaz de detectar múltiples tipos de ciberamenazas mediante la combinación de árboles de decisión dedicados a cada una de ellas.

Bibliografía

[1] K. Scarfone and M. Souppaya, "Protecting against ransomware attacks," *National Institute of Standards and Technology (NIST)*, no. NIST CSRC Guide, 2023. Accessed: 2025-06-08.

[2] A. Bleih, "Ransomware annual report 2024." https://cyberint.com/blog/research/ransomware-annual-report-2024/, Jan. 2025. Cyberint (a Check Point Company), Accessed: 2025-06-08.

[3] F. Parola, R. Procopio, R. Querio, and F. Risso, "Comparing user space and in-kernel packet processing for edge data centers," *ACM SIGCOMM Computer Communication Review*, vol. 53, Apr. 2023.

[4] Cisco ComSoc Tech Blog, "Highlights of cisco's internet traffic report & forecast." https://techblog.comsoc.org/2021/12/29/highlights-of-ciscos-internet-traffic-forecast, Dec. 2021. Accessed: 2025-06-08.

[5] E. Berrueta, D. Morató, E. Magaña, and M. Izal, "Open repository for the evaluation of ransomware detection tools," 2020. Accessed: 2025-06-07.

[6] E. Berrueta, D. Morato, E. Magaña, and M. Izal, "Open repository for the evaluation of ransomware detection tools," *IEEE Access*, vol. 8, pp. 65658–65669, 2020.

[7] G. Kim, S. Kim, S. Kang, and J. Kim, "A method for decrypting data infected with hive ransomware," *Journal of Information Security and Applications*, vol. 71, p. 103387, 2022.

[8] Heimdal Security, "Crylock ransomware explained: Origins, how it works and how to remove it." Heimdal Security blog, Mar. 2022. Accessed: 2025-07-08.

[9] Linux Manual Pages, *iptables(8) — Administering IPv4 Packet Filtering and NAT*. die.net, 2025. Online. Accessed: 2025-07-11.

[10] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proceedings of the 13th Systems Administration Conference (LISA'99)*, (Seattle, WA, USA), pp. 229–238, USENIX Association, 1999. https://www.usenix.org/publications/library/proceedings/lisa99/roesch.html.

[11] FreeBSD Manual Pages, *bpf(4) — Berkeley Packet Filter*. The FreeBSD Project, 2025. Online. Accessed: 2025-07-21.

[12] The tcpdump Group, *libpcap User Documentation*. Read the Docs, 2025. Online. Accessed: 2025-07-21.

[13] The Wireshark Foundation, *libpcap — Wireshark Wiki*. Wireshark Wiki, 2025. Online. Accessed: 2025-07-21.

[14] eBPF.io, "What is ebpf?," 2025. Online. Accessed: 2025-07-22.

[15] Aya Contribtors, *XDP — Aya (Rust eBPF): Express Data Path Documentation*. Aya Project, 2025. Online. Accessed: 2025-08-10.

[16] T. Farasat, J. Kim, and J. Posegga, “Smartx intelligent sec: A security framework based on machine learning and ebpf/xdp,” *arXiv preprint arXiv:2410.20244*, 2024. Submitted on 26 Oct 2024.

[17] D. Zhuravchak, A. Tolkachova, A. Piskozub, and V. Dudykevych, “Monitoring ransomware with berkeley packet filter (bpf),” in *Proceedings of the Monitoring Ransomware with Berkeley Packet Filter Conference*, (Kyiv, Ukraine), Nov. 2023.

[18] K. Higuchi and R. Kobayashi, “Real-time defense system using ebpf for machine learning-based ransomware detection method,” in *Proceedings of the Eleventh International Symposium on Computing and Networking Workshops (CANDARW 2023)*, pp. 213–218, IEEE, Dec. 2023.

[19] A. Brodzik, T. Malec-Kruszyński, W. Niewolski, M. Tkaczyk, K. Bocianiak, and S.-Y. Loui, “Ransomware detection using machine learning in the linux kernel,” *arXiv preprint arXiv:2409.06452*, 2024. Cross-listed from cs.CR.

[20] A. Sekar, S. G. Kulkarni, and J. Kuri, “Leveraging ebpf and ai for ransomware nose out,” *arXiv preprint arXiv:2406.14020*, 2024. Submitted June 2024.

[21] I. Bello, H. Chiroma, U. A. Abdullahi, A. Y. Gital, F. Jauro, A. Khan, J. O. Okesola, and S. M. Abdulhamid, “Detecting ransomware attacks using intelligent algorithms: Recent development and next direction from deep learning and big data perspectives,” *Journal of Ambient Intelligence and Humanized Computing*, vol. 12, no. 9, pp. 8699–8717, 2021.

[22] T.-M. Liu, D.-Y. Kao, and Y.-Y. Chen, “Loocipher ransomware detection using lightweight packet characteristics,” in *Proceedings of the KES 2020 Conference, Volume 176 of Procedia Computer Science*, pp. 1677–1683, Elsevier, 2020.

[23] J. Liu, C. Maltzahn, C. Ulmer, and M. L. Curry, “Performance characteristics of the bluefield-2 smartnic,” 2021.

[24] Cisco Systems, Inc., “Cisco nexus smartnic user guide: Local loopback mode.” https://www.cisco.com/c/en/us/td/docs/dcn/nexus3550/smartnic/sw/user-guide/cisco-nexus-smartnic-user-guide/config.html#local-loopback-mode, 2025. Consultado en 2025.

[25] AppNeta / tcpreplay Project, *tcpreplay Manual*, 2025. Accedido: 2025-09-26.

[26] TCPDump Group, *tcpdump(1) - command-line packet analyzer*, 2025. Accedido: 25-10-2025.