UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en 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

septiembre 2025

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

Por Hacer

Palabras clave

Por, Hacer

Abstract

TO DO

Keywords

TO, DO

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 del sistema ni se convierta en un cuello de botella. Para poder desplegar un sistema de detección y respuesta ante ransomware sin afectar al renidimiento del sistema, es fundamental analizar dónde 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 árboles de decisión 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.

Debido a las limitaciones de almacenamiento y capacidad de cómputo disponibles, así como al alcance definido para este trabajo, he 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.

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, Berrueta pretende 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 Filtrado de Paquetes

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.

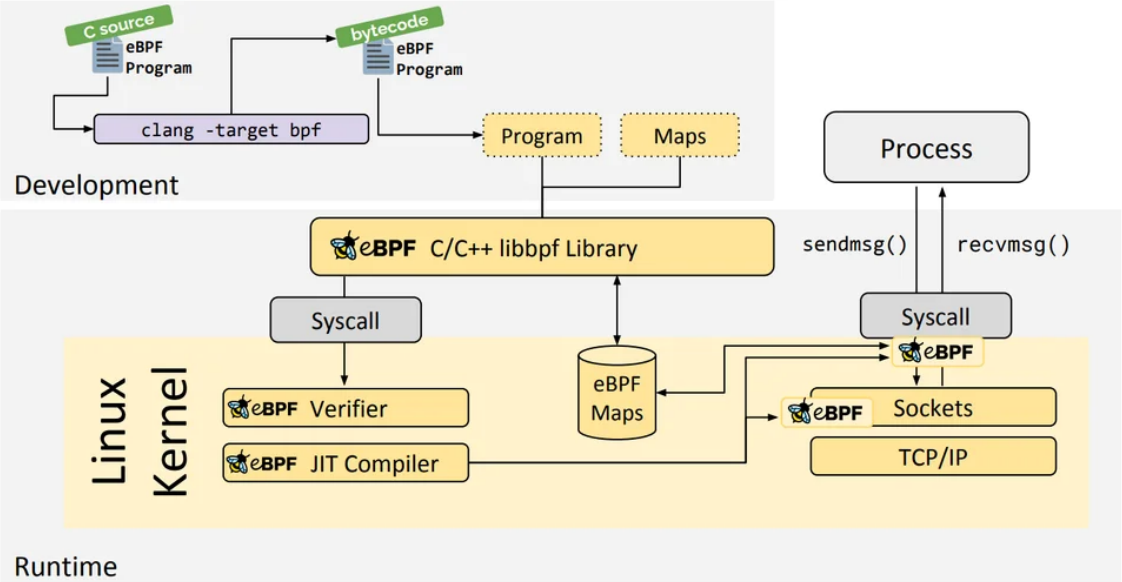

Figura 2.1: Flujo de ejecución de un programa eBPF. Obtenida de [14] siguiendo la licencia Creative Commons Attribution 4.0 International License.

2.3. Artículos Similares y 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 Zhuravchak 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

3.2. Componentes

Los componentes de la arquitectura se dividen en tres módulos principales. El primero corresponde al árbol de decisión, entrenado mediante la biblioteca scikit-learn de Python. El segundo está constituido por el código XDP, el cual incluye una parte autogenerada a partir del árbol de decisión. Finalmente, el tercer módulo es el entorno Mininet, empleado para la ejecución de pruebas en un entorno virtual local. 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.

3.2.1. Árbol de decisión

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 (véase un fragmento de esta función en 3.1). 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. 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 podría 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 datasets de paquetes de red no se apreciaron mejoras de precisión al superar este valor.

Para entrenar el árbol de decisión, una vez definidas las columnas que serán eliminadas, se establece la variable x como el conjunto de columnas que forman parte de los datos de entrada y la variable y como la columna correspondiente al clasificador. A continuación, mediante la función `train_test_split`, se divide 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 procede a entrenar el árbol utilizando los parámetros seleccionados y los datos de entrenamiento. Finalmente, se puede calcular la exactitud del modelo, si bien es recomendable interpretar este valor con cautela, dado que no siempre refleja de forma completamente sólida la capacidad de generalización del clasificador. El código correspondiente al entrenamiento del árbol puede consultarse en 3.2

Durante el desarrollo, se detectó que, por error, se había incluido la variable `eth_type` en la lista de columnas excluidas del entrenamiento. Tras corregir esta omisión y volver a entrenar el modelo, se

observó que la precisión se mantenía, pero esta vez con un nivel menos de profundidad, alcanzando una profundidad máxima de tres niveles a pesar de que el límite establecido era de cuatro. Este resultado sugiere que la inclusión de dicha variable aporta capacidad predictiva al modelo, optimizando su estructura sin pérdida de precisión.

Código 3.1: Fragmento de códgio que transforma parte de los datos de un paquete en un diccionario fácilmente convertible a CSV.

```
39 def paqt2dict(paqt: Packet) -> Dict[str, Any]:
40     '''Función que convierte un paquete en un diccionario con las características extraídas.'''
41     dic = {}
42 
43     try:
44         packet = Ether(paqt) if isinstance(paqt, bytes) else paqt
45     except Exception as e:
46         print(f"Error al procesar el paquete: {e}")
47         return {}
48 
49     # Tamaño total del paquete
50     dic["size"] = len(packet)
51 
52     # Capa Ethernet
53     if packet.haslayer(Ether):
54         dic["eth_src"] = packet[Ether].src
55         dic["eth_dst"] = packet[Ether].dst
56         dic["eth_type"] = packet[Ether].type
57     else:
58         dic["eth_src"] = None
59         dic["eth_dst"] = None
60         dic["eth_type"] = None
```

3.2.2. 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

Código 3.2: Fragmento de código de arbolrw.py encargado de entrenar el árbol de decisión.

```
82  # Especifica la ruta del archivo CSV
83  csv_file_path = '../dataset.csv'
84
85  # Lista de columnas que NO quieres usar para el entrenamiento
86  columnas_a_excluir = ["eth_src", "eth_dst", "ip_src", "ip_dst", "timestamp"]
87
88  # Cargar el CSV
89  try:
90      df = pd.read_csv(csv_file_path)
91  except FileNotFoundError:
92      print(f"Error: El archivo CSV no se encontró en la ruta: {csv_file_path}")
93      exit()
94
95  # Última columna es la clase objetivo
96  nombre_columna_objetivo = df.columns[-1]
97  y = df[nombre_columna_objetivo]
98
99  # Columnas X sin las excluidas
100 columnas_caracteristicas = [col for col in df.columns if col != nombre_columna_objetivo and col not
        in columnas_a_excluir]
101 x = df[columnas_caracteristicas]
102 x = pd.get_dummies(x)
103
104 # División de datos
105 x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
106
107 # Entrenar el modelo
108 modelo = DecisionTreeClassifier(criterion='gini', splitter='best', max_depth=4, random_state=42)
109 modelo.fit(x_train, y_train)
110 exactitud = modelo.score(x_test, y_test)
111 print(f'La exactitud del modelo es: {exactitud:.2f}')
```

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 3.3 se muestra la sección del fichero XDP/arbol1/xdp_usr.c que implementa las funciones principales previamente descritas.

Se añadió la capacidad de autogenerar el código BPF/XDP una vez entrenado el árbol de decisión. Para ello, se recorre de forma recursiva el árbol y, en cada nodo, se genera la condición correspondiente a partir del nombre de la característica (previamente convertido a su nombre real) y del valor de umbral. Esta condición se pasa por una función que, en determinados casos, la sustituye por una equivalente válida. Por ejemplo, la condición `src_port <= inf` se reemplaza por `ip_proto == IPPROTO_TCP || ip_proto == IPPROTO_UDP`, dado que el hecho de que el puerto de destino sea menor que infinito únicamente implica que el puerto existe.

Cuando la condición resultante es válida, si corresponde al hijo izquierdo del nodo se inserta en una estructura if, mientras que si corresponde al hijo derecho se inserta en una estructura else. Para la generación del código se empleó una plantilla Jinja con la estructura prediseñada en C. Durante las pruebas, se observó que el mejor resultado se obtenía extrayendo los datos de las cabeceras IP antes de la evaluación del árbol, de forma que ya estuvieran disponibles en la plantilla. En caso de que el paquete analizado carezca de cabecera IP, dichos valores se establecen en cero.

3.2.3. Mininet

Mininet es un emulador de red que permite diseñar topologías compuestas por hosts finales, switches, routers (aunque de manera no nativa, ofreciendo únicamente el reenvío de paquetes) y enlaces dentro de un único kernel Linux. Además, dispone de una API accesible desde Python, lo que facilita el uso de este lenguaje de programación para la creación y configuración de las topologías de red [23].

Como entorno de pruebas, se utilizó Mininet para implementar una topología inicial, que se muestra

Código 3.3: 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  }
```

en la figura 3.1. Esta configuración consistía en dos hosts, cada uno conectado a un switch distinto, mientras que ambos switches se encontraban interconectados a un único router. Esta topología permitió realizar pruebas preliminares sobre el impacto del filtro basado en árboles de decisión en el rendimiento de la red.

Si bien esta configuración resultó útil para evaluar el funcionamiento básico del filtro, no era completamente representativa de un escenario real con una tarjeta de red equipada con DPU. Aunque dentro de la tarjeta los paquetes llegan al procesador de la DPU a través de Ethernet, la topología de Mininet no reproduce de manera exacta la interacción de los paquetes con el hardware especializado de la NIC. Al añadir más de una interfaz switch además del router, se introducen latencias adicionales que no existen en la NIC, por lo que este diseño inicial sirve únicamente para pruebas preliminares y no refleja completamente las condiciones de un entorno con DPU.

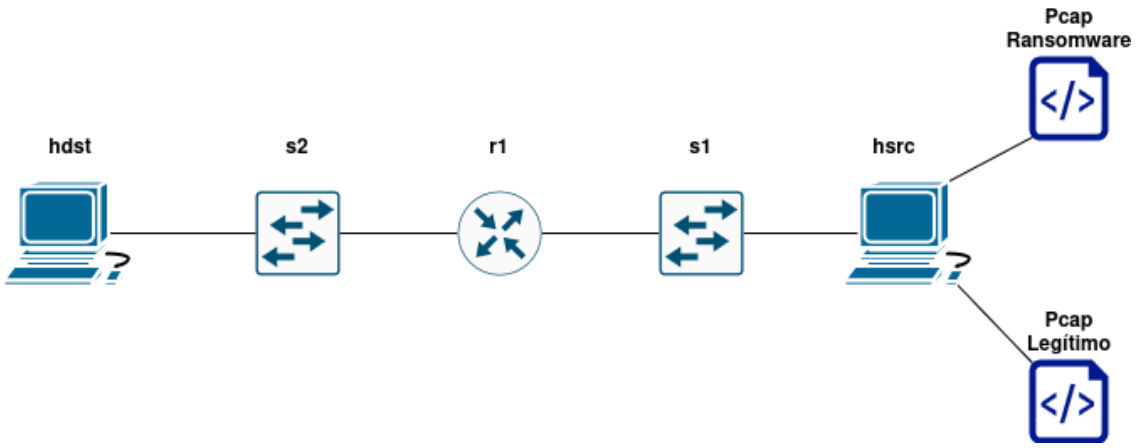

Figura 3.1: Primer diseño de topología de red usada en Mininet preliminares.

Teniendo en cuenta que el entorno de pruebas real consiste en una tarjeta de red con dos puertos a nivel de enlace, se decidió simplificar la topología de Mininet para realizar pruebas más realistas. La red utilizada consta únicamente de dos host conectados a un único switch, eliminando la necesidad de routers adicionales o interfaces suplementarias tal y como se puede ver en la figura 3.2. Esta simplificación permite reducir la complejidad de la topología y minimizar factores externos que podrían afectar a las mediciones, como las latencias introducidas por componentes adicionales de la red virtual.

Además, esta configuración simplificada contribuye a disminuir la pérdida de paquetes durante las pruebas y reduce problemas relacionados con la resolución de direcciones MAC mediante ARP. Al aproximarse más al comportamiento de un entorno real con DPU, la topología proporciona un marco adecuado para medir el rendimiento de la red bajo la influencia del filtro de árbol de decisión. El siguiente paso en el proceso consistirá en realizar pruebas de rendimiento utilizando la herramienta iperf, lo que permitirá cuantificar de manera más precisa el impacto del filtrado sobre el tráfico de red.

En los fragmentos de código 3.4 y 3.5 se muestran extractos de los ficheros redarbolrw.py y redarbolrw1.py, correspondientes a la definición de la topología preliminar y de la topología definitiva,

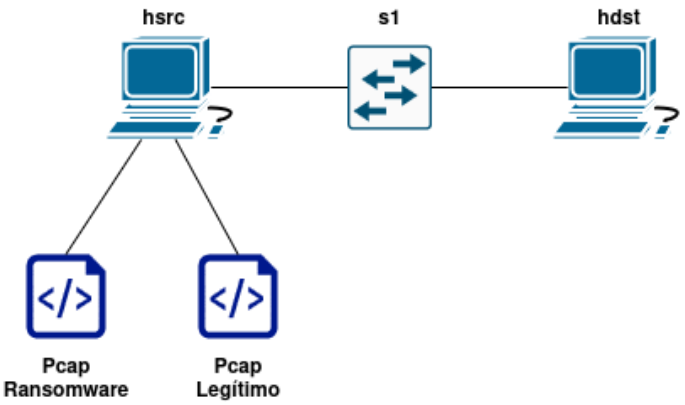

Figura 3.2: Diseño de topología de red usada en Mininet para pruebas.

respectivamente.

Código 3.4: Fragmento de código que define la topología de red usada en las pruebas preeliminares.

```
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 3.5: Fragmento de código que define la topología de red usada en las pruebas finales.

```
8  class SimpleTopo(Topo):
9      """
10     Topología simple compuesta por:
11       -hsrc: Host origen con IP 10.0.1.1/24 y MAC constante
12       -hdst: Host destino con IP 10.0.1.2/24 y MAC constante
13       -s1: Switch conectado a ambos hosts
14     """
15     def build(self):
16         # Crear hosts con direcciones MAC constantes
17         hsrc = self.addHost('hsrc', ip='10.0.1.1/24', mac='00:00:00:00:01:01')
18         hdst = self.addHost('hdst', ip='10.0.1.2/24', mac='00:00:00:00:02:02')
19
20         # Crear switch
21         s1 = self.addSwitch('s1')
22
23         # Conectar hosts al switch
24         self.addLink(hsrc, s1)
25         self.addLink(hdst, s1)
```

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.

[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] B. Lantz, N. Handigol, B. Heller, and V. Jeyakumar, “Introduction to mininet,” 2021. Accedido: 2025-08-16.