UCUENCA

Universidad de Cuenca

Facultad de Ingeniería

Carrera de Ingeniería Electrónica y Telecomunicaciones

Implementación y evaluación de algoritmos de detección de botnets basados en técnicas DGA en la red de comunicación de un Instituto de Educación Superior (IES)

Trabajo de titulación previo a la obtención del título de Ingeniero en Electrónica y Telecomunicaciones

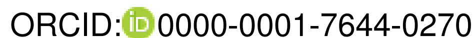
Autores:

Erick Adrián Fernández Orellana

Linder Flavio Quizhpe Quezada

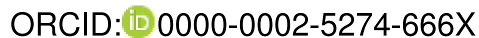
Director:

Darwin Fabian Astudillo Salinas


ORCID: 0000-0001-7644-0270

Co-Director:

Luis Patricio Tello Oquendo


ORCID: 0000-0002-5274-666X

Cuenca, Ecuador

2023-08-03

Resumen

Con la constante evolución de las redes de telecomunicaciones y el aumento exponencial del tráfico en Internet, es necesario prevenir ataques informáticos cada vez más sofisticados. DGAs es una técnica que permite generar dominios maliciosos de forma automática y encubierta para controlar Bots y ejecutar estos ataques. Se propone implementar dos algoritmos de detección de Botnets basadas en DGAs: *MaldomDetector* y N-gramas enmascarados. Estos utilizan aprendizaje automático supervisado y se basan en la extracción de características léxicas y estadísticas de los nombres de dominio. Para llevar a cabo la detección de mAGDs, se utilizará el *framework* BNDF como base. Sin embargo, dado que BNDF no ofrece resultados en tiempo real, se desarrolló un módulo de detección temprana que en base a los algoritmos de detección seleccionados, optimiza el funcionamiento del *framework*. Se diseñaron distintos escenarios de prueba, en entornos controlados y en una red real. En los escenarios controlados, por medio de diversas métricas de evaluación se determinó el rendimiento de detección de los algoritmos. En las pruebas en redes reales, se analizaron las solicitudes DNS junto con las predicciones realizadas por los algoritmos, con el objetivo de evaluar la veracidad de las predicciones. Por último, se evaluó el uso de los recursos computacionales requeridos por cada algoritmo. N-gramas enmascarados demostró un excelente desempeño en términos de clasificación, con un valor de 85.09 % en todas las métricas. *MaldomDetector* mostró un mejor tiempo de procesamiento con 1.38 ms por dominio, convirtiéndose en la mejor opción para redes con recursos limitados.

Palabras clave: Botnet, DGA, MaldomDetector, N-gramas enmascarados, BNDF

El contenido de esta obra corresponde al derecho de expresión de los autores y no compromete el pensamiento institucional de la Universidad de Cuenca ni desata su responsabilidad frente a terceros. Los autores asumen la responsabilidad por la propiedad intelectual y los derechos de autor.

Repositorio Institucional: https://dspace.ucuenca.edu.ec/

Abstract

With the constant evolution of telecommunications networks and the exponential increase in Internet traffic, it is necessary to prevent increasingly sophisticated cyber-attacks. DGAs is a technique that allows for the automatic and covert generation of malicious domains to control Bots and execute these attacks. It is proposed to implement two Botnets detection algorithms based on DGAs: *MaldomDetector* and masked N-grams. These algorithms use supervised machine learning and rely on the extraction of lexical and statistical features from domain names. To carry out the detection of mAGDs, the BNDF framework will be used as a base. However, as BNDF does not provide real-time results, an early detection module was developed to optimize the framework's operation based on the selected detection algorithms. Different test scenarios were designed in controlled environments and on a real network. In the controlled scenarios, various evaluation metrics were used to determine the detection performance of the algorithms. In real network tests, DNS requests were analyzed alongside the predictions made by the algorithms, with the aim of evaluating the accuracy of the predictions. Finally, the computational resource usage required by each algorithm was evaluated. Masked N-grams demonstrated excellent performance in terms of classification, achieving a value of 85.09 % in all metrics. *MaldomDetector* showed a better processing time with 1.38 ms per domain, making it the best option for networks with limited resources.

Keywords: Botnet, DGA, MaldonDetector, masked N-gram, BNDF

The content of this work corresponds to the right of expression of the authors and does not compromise the institutional thinking of the University of Cuenca, nor does it release its responsibility before third parties. The authors assume responsibility for the intellectual property and copyrights.

Institutional Repository: `https://dspace.ucuenca.edu.ec/`

Índice de contenido

1. Introducción 15
1.1. Identificación del problema 15
1.2. Justificación 16
1.3. Alcance 17
1.4. Objetivos 18
1.4.1. Objetivo general 18
1.4.2. Objetivos específicos 18

2. Marco teórico 19
2.1. Botnets 19
2.1.1. Historia 20
2.1.2. Ciclo de vida 22
2.1.3. Estructura 23
2.2. Algoritmo de generación de dominios (DGA) 25
2.2.1. Motivación y descripción 25
2.2.2. Funcionamiento 26
2.2.3. Impacto 27
2.2.4. Clasificación de DGAs 27
2.3. Aprendizaje automático 28
2.3.1. Algoritmo *K-Nearest Neighbors* (KNN) 29
2.3.2. Algoritmo *Support Vector Machine* (SVM) 31
2.3.3. Algoritmo *Random Forests* 34
2.3.3.1. Algoritmo *Decision Trees* 34
2.3.3.2. Aprendizaje conjunto 37
2.3.4. Algoritmo *Multi-Layer Perceptron* (MLP) 38
2.3.5. Perceptrón 38
2.3.6. *Multi-Layer Perceptron* (MLP) 40
2.4. Clasificación de sistemas de detección de Botnets 42

3. Trabajos relacionados 44
3.1. Sistema de detección *MaldomDetector* 45

3.2. Sistema de detección por medio de N-gramas enmascarados 48
3.3. *BotNet Detection Framework* (BNDF) 51

4. Diseño e implementación 55
4.1. Recursos y herramientas 55
4.1.1. Base de datos 55
4.1.2. Selección de algoritmos de aprendizaje automático 57
4.1.2.1. Algoritmo *K-Nearest Neighbors* (KNN) 58
4.1.2.2. Algoritmo *Support Vector Machine* (SVM) 58
4.1.2.3. Algoritmo *Random Forests* 59
4.1.2.4. Algoritmo *Multi-Layer Perceptron* (MLP) 59
4.1.2.5. Ajuste de hiperparámetros 60
4.1.3. Selección de métricas de evaluación 60
4.2. Implementación de algoritmos de detección de mAGDs 63
4.2.1. Algoritmo *MaldomDetector* 63
4.2.2. Algoritmo basado en N-gramas enmascarados 66
4.3. Escenario de trabajo 68
4.4. Arquitectura propuesta 71
4.4.1. Requerimientos previos 71
4.4.2. Instalación de *BotNet Detection Framework* (BNDF) 72
4.4.3. Contribuciones 73

5. Resultados 76
5.1. Evaluación de los algoritmos de aprendizaje automático 77
5.1.1. Algoritmo MaldomDetector 77
5.1.2. Algoritmo N-gramas enmascarados 79
5.2. Escenario de evaluación *offline* 81
5.3. Escenario de evaluación *online* 83
5.4. Evaluación del uso de recursos computacionales 92

6. Conclusiones y trabajos futuros 94
6.1. Conclusiones 94
6.2. Recomendaciones 96
6.3. Trabajos Futuros 97

Referencias **99**

A. Anexo A **106**

A.1. Instalación de *BotNet Detection Framework* (BNDF) 106

A.1.1. Requerimientos previos 106

A.1.1.1. Instalación de *docker 17.06.0* 106

A.1.1.2. Instalación de *docker-compose* 1.27.0 108

A.1.1.3. Instalación de *git* 108

A.1.1.4. Instalación de *curl* 109

A.1.1.5. Instalación de *time* 109

A.1.2. Instalación 109

Índice de figuras

2.1. Arquitectura genérica de una Botnet. 20
2.2. Ciclo de vida de una Botnet. 23
2.3. Hiperplano separador óptimo para el algoritmo SVM [1]. 32
2.4. Arquitectura de una TLU [2]. 39

3.1. Arquitectura de *MaldomDetector* [3]. 46
3.2. *Randomness Measurement Algorithm* (RMA) [3]. 47
3.3. Arquitectura de BNDF desde la perspectiva de los procesos [4]. 52

4.1. Configuración del cliente VPN. 69
4.2. Conexión establecida a través de la VPN. 70
4.3. Acceso al servidor mediante el protocolo *Secure Shell* (SSH). 71
4.4. *Dockers* inicializados exitosamente. 73
4.5. Arquitectura propuesta desde la perspectiva de los procesos. 75

5.1. Evaluación del rendimiento de los algoritmos de aprendizaje automático bajo el enfoque de *MaldomDetector*. 78
5.2. Evaluación del rendimiento de los algoritmos de aprendizaje automático bajo el enfoque de N-gramas enmascarados. 80
5.3. Evaluación del rendimiento de los algoritmos *MaldomDetector*, N-gramas enmascarados y RMA. 82
5.4. Cantidad de solicitudes DNS capturadas por día. 85
5.5. Cantidad de nombres de dominio catalogados como mAGDs por cada algoritmo de detección. 86
5.6. Cantidad de mAGDs únicos y coincidentes detectadas por cada algoritmo. 89
5.7. Porcentaje de solicitudes DNS catalogadas como mAGDs con `_rescode :NOERROR`. 90
5.8. Porcentaje de solicitudes DNS catalogadas como mAGDs con `_rescode :NXDOMAIN`. 91

Índice de tablas

2.1. Ejemplos de mAGDs de cada clase. 28

3.1. Características básicas y derivadas. 46
3.2. Hiperparámetros utilizados y resultados obtenidos en la evaluación de modelos [3]. 48
3.3. Enmascaramiento de caracteres. 49
3.4. Características léxicas y estadísticas. 50
3.5. Conjunto de N-gramas relevantes en el cálculo de características. . . . 50
3.6. Resultados obtenidos en la evaluación de modelos. 51
3.7. Características extraídas para la generación de fingerprints. 53

4.1. Clases de DGAs disponibles en UMUDGA. 56
4.2. Cantidad de muestras seleccionadas en la base de datos. 57
4.3. Hiperparámetros relevantes a configurar en el algoritmo KNN. 58
4.4. Hiperparámetros relevantes a configurar en el algoritmo SVM. 59
4.5. Hiperparámetros relevantes a configurar en el algoritmo *Random Forest*. 59
4.6. Hiperparámetros relevantes a configurar en el algoritmo MLP. 59
4.7. Parámetros de entrada de *GridSearchCV* 60
4.8. Matriz de confusión genérica. 61
4.9. Ajuste de los hiperparámetros en el algoritmo KNN (*MaldomDetector*). 65
4.10. Ajuste de los hiperparámetros en el algoritmo SVM (*MaldomDetector*). 65
4.11. Ajuste de los hiperparámetros en el algoritmo *Random Forests* (*MaldomDetector*). 65
4.12. Ajuste de los hiperparámetros en el algoritmo MLP (*MaldomDetector*). . 65
4.13. Ajuste de los hiperparámetros en el algoritmo KNN (N-gramas enmascarados). 67
4.14. Ajuste de los hiperparámetros en el algoritmo SVM (N-gramas enmascarados). 67
4.15. Ajuste de los hiperparámetros en el algoritmo *Random Forests* (N-gramas enmascarados). 68
4.16. Ajuste de los hiperparámetros en el algoritmo MLP (N-gramas enmascarados). 68

4.17. Características del servidor proporcionado por la Institución de Educación Superior (IES). 69

5.1. Evaluación del rendimiento de los algoritmos de aprendizaje automático bajo el enfoque de *MaldomDetector*. 78
5.2. Evaluación del rendimiento de los algoritmos de aprendizaje automático bajo el enfoque de N-gramas enmascarados. 80
5.3. Evaluación del rendimiento de los algoritmos *MaldomDetector* y N-gramas enmascarados y RMA. 82
5.4. Número de mAGDs detectados por cada algoritmo. 87
5.5. mAGDs totales detectados por cada algoritmo. 88
5.6. Evaluación de los recursos computacionales ocupados por los algoritmos. 92

Dedicatoria

Es difícil expresar con palabras la gratitud que siento hacia Dios en este momento. Él es el único responsable de mis éxitos académicos pasados y futuros, y dedicarle la culminación de esta etapa de mi vida es mi forma de darle la gloria que merece.

Fernando y Norma, ustedes son la bendición más grande de mi vida. Sin su amor y dirección nunca hubiera tenido la capacidad de ver llegar este día, por lo cual, ustedes merecen un reconocimiento aún mayor que el que yo pueda recibir. Es por esto que quiero dedicarles este logro, así como cada una de las palabras que he sido capaz de escribir a lo largo de mi vida académica.

Rosty, Chester, Ringo, Cooper, Layla, y a todas mis mascotas, su simple presencia me ha dado alegría y motivación en momentos donde rendirse hubiera sido la opción más fácil. Se han convertido en parte de mi familia, y sin ustedes, recorrer este camino no tuviera ningún valor. Además y sin lugar a dudas, quiero mencionar y dedicar un agradecimiento especial a Tim Henson, cuyo talento excepcional y estilo único logró deslumbrarme en mi peor momento, demostrándome que rendirse es una elección personal que siempre podemos evitar.

Indiscutiblemente, este momento no hubiera sido posible sin la disposición y confianza de mi profesor y director de tesis, Ing. Darwin Fabián Astudillo Salinas. Agradezco enormemente su apoyo incondicional, así como el conocimiento que me ha compartido incluso más allá de las aulas de clase. Por último, a mi compañero de tesis y amigo cercano, Linder, quiero expresarle un profundo agradecimiento por el apoyo y amistad que me ha brindado desde siempre. Ha sido un placer compartir este viaje académico juntos.

Erick Adrián Fernández Orellana

Dedicatoria

Agradezco a Dios por bendecirme en todo mi camino académico, por darme la oportunidad de cumplir uno de mis sueños, por ser mi guía, mi fortaleza, por darme la salud y la sabiduría. De cada uno de mis logros, la gloria es de Dios.

De manera especial, agradezco a mis padres, Ángel y Carmita, por sus consejos, su apoyo, amor incondicional y todo el sacrificio que han hecho por mí para poder lograr cada una de mis metas. Siempre han sido mi inspiración para superar cualquier obstáculo, y les dedico cada logro en mi vida. A mi hermano Luis Miguel y a la Cloy por su apoyo, motivación y compañía en cada paso de mi vida universitaria. Agradezco también a todas las personas que siempre han creído en mí, me han ayudado y me han dado aliento para continuar y seguir adelante.

También quiero expresar mi agradecimiento a nuestro profesor y director de tesis, Ing. Darwin Fabián Astudillo Salinas, quien nos ha brindado su ayuda, su tiempo y ha compartido sus conocimientos durante el desarrollo de este trabajo de titulación. A mi compañero de tesis, Adrián, por su dedicación en este trabajo, le deseo éxitos en su vida profesional.

Linder Flavio Quizhpe Quezada

Abreviaciones y acrónimos

ANN *Artificial Neural Networks*. 38, 40

APT *Advanced Package Tool*. 106, 107

BNDF *BotNet Detection Framework*. 2, 3, 5–7, 17, 18, 43–46, 51, 52, 54, 55, 57, 58, 64, 68, 71–73, 79, 81, 91, 95, 98, 106, 109, 110

Bot Aféresis de Robot. 2, 3, 16, 19–26, 43–45, 54, 55, 61, 73–75

Botnet *Robot Network*. 2–4, 7, 16–27, 42–44, 51, 55, 57, 60, 63, 74, 84, 94

CART *Classification and Regression Trees*. 50

CC *Command and Control*. 19, 20, 22–27, 43, 45, 74, 75, 84, 90

CPU Unidad Central de Procesamiento. 72

DDoS *Distributed Denial of Service*. 16, 22

DGA *Domain Generation Algorithm*. 2–4, 8, 16–19, 24–28, 43–45, 47, 51, 54–57, 74, 75, 87, 89, 90, 97, 98

DNS *Domain Name System*. 2, 3, 7, 17, 19, 22, 26, 42–46, 51–54, 64, 66, 68, 69, 72–76, 83–85, 87, 89–91, 94–96, 106

ELK *Elasticsearch-Logstash-Kibana*. 52

FQDN *Fully Qualified Domain Name*. 26, 43, 84, 97

GBM *Gradient Boosting Machine*. 50

GELF *Graylog Extended Log Format*. 52, 83

GPG *GNU Privacy Guard*. 106, 107

HTTP *Hypertext Transfer Protocol*. 23, 72

HTTPS *Hypertext Transfer Protocol Secure*. 72, 106

IA Inteligencia Artificial. 19

IES Institución de Educación Superior. 9, 17, 18, 45, 55, 57, 68–70, 74, 76, 83–85, 95, 96

IoT *Internet of Things*. 22

IP *Internet Protocol*. 22–26, 42, 43, 69, 75, 97

IRC *Internet Relay Chat*. 21, 23

ISP *Internet Service Provider*. 97

KNN *K-Nearest Neighbors*. 4, 5, 8, 29–31, 51, 57, 58, 65, 67, 77–79, 81, 92, 95, 96

mAGD *Malicious Algorithmically-Generated Domains*. 2, 3, 5, 7–9, 26–28, 43, 44, 46, 51, 55, 56, 63, 64, 73–77, 81, 83, 85–92, 94, 95, 97, 98

MLP *Multi-Layer Perceptron*. 4, 5, 8, 38, 40, 41, 57, 59, 65, 68, 77–81, 95

P2P *Peer to peer*. 21, 24

RAM *Random Access Memory*. 69

RBF *Radial Basis Function*. 34

ReLU *Rectified Lineal Unit*. 40

RMA *Randomness Measurement Algorithm*. 7, 9, 17, 43, 45–47, 54, 63, 64, 73, 74, 76, 81, 82, 85, 87–89, 91, 95, 96

RSA *Rivest-Shamir-Adleman*. 70

SLD Second-Level Domain. 52, 97

SMTP *Simple Mail Transfer Protocol*. 21

SO Sistema Operativo. 22

SSH *Secure Shell*. 7, 68, 70, 71

SVM *Support Vector Machine*. 4, 5, 7, 8, 31, 32, 51, 57–59, 65, 67, 77, 79, 95

TCP *Protocolo de Control de Transmisión*. 42

TLD *Top-Level Domain*. 26, 52

TLU *Threshold Logic Unit*. 7, 38–40

TTL *Time To Live*. 42

UMUDGA *University of Murcia domain generation algorithm dataset*. 8, 56, 57, 94

USB *Universal Serial Bus*. 22

VPN *Virtual Private Network*. 7, 68–70

YAML *Ain't Markup Language*. 72

Introducción

Este capítulo presenta la identificación del problema, justificación, alcance y los objetivos del presente proyecto de titulación.

1.1. Identificación del problema

Las redes de telecomunicaciones están en constante evolución, lo que implica replantear los paradigmas en los que se sustentan, enfocándose en la generación de nuevas tecnologías para la conectividad, y en el desarrollo de nuevas arquitecturas para su despliegue. Los usuarios pueden percibir esta evolución como un aumento en la cobertura y velocidad de la transferencia de datos. Sin embargo, desde el punto de vista de un proveedor de Internet, mantener la integridad de la red es otro ámbito que se busca mejorar.

Tal como se menciona en [5], las redes de telecomunicaciones han evolucionado de un mero medio de comunicación a una infraestructura computacional omnipresente. Con el crecimiento constante de aplicaciones, se tiene como consecuencia la generación de una gran cantidad de tráfico de datos transmitido por Internet que potencialmente puede contener información confidencial. Esto también abre la posibilidad a un mayor riesgo de ataques informáticos malintencionados, los cuales, cada vez se vuelven más sofisticados, peligrosos y complejos. Como resultado, los administradores de las redes se esfuerzan para fortalecer la seguridad ante las vulnerabilidades de estas, haciendo que la seguridad haya pasado de ser una característica complementaria a una inherente, en cualquier red de telecomunicaciones con el fin de salvaguardar su integridad.

Según [6], las necesidades actuales en seguridad para cualquier red de telecomunicaciones conectada a Internet, se enfocan en asegurar la detección de intrusos informáticos para generar medidas de protección y así precautelar los datos de los usuarios, y los recursos de la red. Una intrusión se refiere a una acción no autorizada que tiene como objetivo acceder o manipular información, o hacer que un sistema sea inutilizable o vulnerable [7]. En el caso de una intrusión exitosa, puede dar lugar a diversos tipos de ataques informáticos, de los cuales se destacan los siguientes tipos:

el ataque distribuido de denegación de servicio (DDoS, por sus siglas en inglés), gusanos, virus, *spam*, obtención de acceso privilegiado a los datos de un *host*, *phishing*, fraude de *clicks*, propagación de *malware*, *adware*, *spyware*, entre otros [8, 9]. Una vez que un *host* ha sido comprometido por una intrusión, los atacantes pueden desplegar aplicaciones que se ejecutan de forma automática y oculta para los usuarios, y que son controladas a través de Internet.

Cuando un *host* se ve comprometido por una intrusión, es denominado Bot y a su vez, una Botnet es un grupo de Bots distribuidos geográficamente [10], por lo cual, son redes difíciles de detectar [9]. Los atacantes tienen como objetivo primordial precautelar la conectividad con sus Bots. Actualmente, la mayoría de Botnets utilizan algoritmos de generación de nombres de dominio (DGA, por sus siglas en inglés) para ocultar la comunicación y eludir los sistemas de detección. Estos algoritmos generan una lista pseudoaleatoria de posibles dominios de servidores desde los cuales el atacante podrá controlar las acciones del Bot [10].

1.2. Justificación

A pesar de que las Botnets existen desde aproximadamente 1999, con la aparición de *Sub7* y *Pretty Park*, actualmente aún no se han encontrado soluciones totalmente efectivas [11], provocando así cada vez más variedades de ataques. Algunas estadísticas alarmantes revelan que en 2017, según [12], los servicios financieros fueron uno de los objetivos más atacados por medio de las Botnets llegando a representar el 77,44 % del total de ataques. Así mismo, [13] indica que en el primer trimestre de 2021, el número de servidores utilizados para controlar las Botnets aumentó en un 24 % con un total de 1660 nuevos servidores en comparación del cuarto trimestre del 2020.

La mayor preocupación para cualquier atacante que utilice una Botnet, es mantener el control de sus Bots por el mayor tiempo posible. Por este motivo, los DGAs resultan atractivos debido a su facilidad de implementación y gran capacidad de reutilización. Por otra parte, la mayor preocupación de un analista de ciberseguridad en este ámbito, es identificar los dominios de red generados por los DGAs y bloquear el acceso de la

red a estos dominios de forma inmediata.

Aunque estos algoritmos han evolucionado continuamente para evitar la detección de los dominios generados por mecanismos de seguridad, afortunadamente, tal como lo manifiesta [14], estos dominios aún pueden ser distinguidos en función de sus características intrínsecas. Las técnicas de detección de Botnets se han clasificado de diferentes maneras a través de distintos enfoques en función del autor, pero en general, las clasificaciones más actuales se basan en el análisis del tráfico del tipo DNS.

1.3. Alcance

El objetivo del presente trabajo de titulación es implementar algoritmos de detección de Botnets basadas en el uso de DGAs. Esto se llevará a cabo mediante el análisis de los flujos DNS en la red de una IES. Se propone utilizar como base el *framework* de libre distribución BNDF para optimizarlo a través de la incorporación de algoritmos más eficientes en la detección de dominios generados por DGAs, en comparación con el algoritmo determinista *Randomness Measurement Algorithm* (RMA) introducido en el *framework*. Se propone la implementación de los algoritmos de detección *Maldom-Detector* [3] y “N-gramas enmascarados” [15], los cuales, han demostrado una mayor precisión en comparación con RMA.

Si bien la inclusión de estos algoritmos de detección variará en dependencia de sus enfoques individuales, estos comparten etapas de implementación similares. Se partirá de una recopilación de bases de datos confiables y debidamente etiquetadas de dominios benignos y maliciosos a partir de fuentes de libre distribución. Estas bases de datos se utilizarán para entrenar los modelos seleccionados, siendo esta una etapa fundamental para el funcionamiento de los detectores en todos los modelos revisados. Finalmente, se validarán las implementaciones realizadas a través de dominios benignos y maliciosos no empleados durante la fase de entrenamiento, siendo esta una prueba recurrente en este tipo de estudios. Además, se evaluará el rendimiento de los mismos utilizando el tráfico circundante en la red de una IES en tiempo real.

1.4. Objetivos

1.4.1. Objetivo general

Implementar y evaluar dos algoritmos de detección de Botnets basados en técnicas de DGAs en una red de una IES.

1.4.2. Objetivos específicos

- Revisar el estado del arte con el fin de identificar algoritmos de detección de DGAs actuales y eficientes en términos de precisión y uso de recursos computacionales.
- Implementar dos algoritmos de detección de DGAs en el *framework* BNDF.
- Comparar la precisión en la detección de Botnets de los algoritmos implementados.
- Analizar el rendimiento de detección de los algoritmos al ejecutarse en tiempo real en la red de una IES.

Marco teórico

En este capítulo se presenta una revisión teórica de los conceptos necesarios para comprender e implementar sistemas eficientes de detección de Botnets que empleen dominios generados por DGAs. Para esto, en primera instancia se explica el funcionamiento de las Botnets y su ciclo de vida (Sección 2.1). Además, dado que el uso de los DGAs permite a las Botnets evadir los sistemas de detección, entonces conocer sus distintos tipos y principios de funcionamiento resulta relevante para el desarrollo de nuevos sistemas de detección más robustos (Sección 2.2). Otro aspecto a explorar, es el uso de técnicas basadas en Inteligencia Artificial (IA), como, *Machine Learning*, mediante la cual se puede identificar patrones en una base de datos, como por ejemplo, la estructura usada en nombres de dominios maliciosos y así detectar posibles *hosts* infectados (Sección 2.3). Finalmente, en la Sección 2.4 se presenta una clasificación de los sistemas de detección de Botnets que se apoya en el análisis del tráfico del tipo DNS.

2.1. Botnets

Cuando se emplea el término Bot, de manera general se hace referencia a programas informáticos capaces de ejecutar tareas de forma automatizada y configurable. Desde sus primeras implementaciones, los mismos, fueron desarrollados con buenas intenciones para apoyar a los administradores de sistemas. Sin embargo, debido a sus grandes capacidades, los ciberdelincuentes han hecho un mal uso de ellos y los han vuelto, una de las mayores amenazas presentes en Internet [16, 17]. Es por esto que de ahora en adelante, al utilizar el término Bot, haremos referencia a los empleados con objetivos maliciosos. Entonces, como punto de partida es necesario definir claramente los siguientes conceptos:

- **Bots:** Son dispositivos informáticos como computadoras, terminales móviles, servidores, entre otros, con conectividad a Internet, que han sido comprometidos por un *malware*, siendo controlados de manera remota con la finalidad de realizar acciones ilícitas [18–20].
- **Servidor de comando y control (CC):** Es un servidor utilizado por ciberdelin-

cuentes con el propósito de comunicarse con un grupo de Bots previamente generados, permitiendo así, administrarlos, propagar comandos de ejecución, recopilar datos confidenciales y actualizar el estado del *malware*, todo esto de manera anónima [17, 19, 21].

- **Botmaster:** Es el propietario y administrador de toda la estructura de una Botnet. Sus tareas consisten en generar y propagar el *malware* responsable de la infección, y mantener el servidor de CC activo y accesible para los Bots, con el objetivo de controlar sus acciones [20, 22].

Por lo tanto, al utilizar el término Botnet se está haciendo referencia a una red de Bots distribuidos geográficamente, los cuales son administrados y controlados por un *Botmaster*, a través de un servidor de CC para ejecutar una gran variedad de ataques, abarcando desde la recopilación de información confidencial hasta utilizarlos como un medio para desplegar acciones maliciosas hacia otros sistemas en la red [18, 23]. La arquitectura general de una Botnet se presenta en la Figura 2.1.

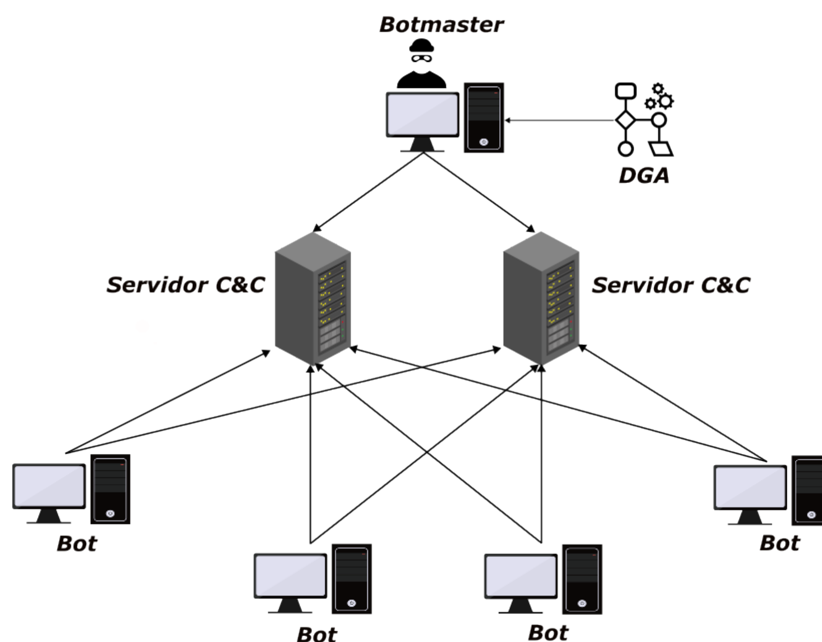


Figura 2.1: Arquitectura genérica de una Botnet.

2.1.1. Historia

A continuación, se realizará un recuento histórico sobre la evolución de las Botnets, en donde, se hará énfasis en las más relevantes de cada época, para poner de manifiesto

el grado de impacto que han tenido sobre los dispositivos y recursos de la red.

La primera Botnet conocida fue *Eggdrop*, la cual, data del año 1993. Esta Botnet a diferencia de la gran mayoría de las presentes en la actualidad, fue creada con el propósito de prestar servicios de *chat online* de manera legítima, por medio del protocolo *Internet Relay Chat* (IRC) [24].

No fue hasta 1999 que las Botnets empezaron a ser utilizadas con propósitos maliciosos. En ese año apareció la Botnet *Sub7*, la cual, se distribuía en forma de un virus troyano y utilizaba nuevamente el protocolo IRC para crear un canal de comunicación con el atacante. Esta Botnet permitía principalmente el robo de archivos y credenciales [11]. Simultáneamente, el mismo año se lanzó la Botnet *Pretty Park*, la cual, se distribuía en forma de un gusano a través de archivos con extensión "*.exe*" adjuntos en correos electrónicos y también operaba por medio del protocolo IRC. *Pretty Park* tenía la capacidad de escanear el directorio del correo electrónico de la máquina infectada para realizar ataques de *spam* y seguir propagándose [11].

Para la década del 2000 la evolución de las Botnets fue tanta que adquirieron nuevas habilidades, como por ejemplo, la comunicación *Peer to peer* (P2P). En 2007 se detectó la Botnet *Zeus*, la cual, disponía una media de 3.6 millones de Bots por ataque, era especialista en robo de información financiera, distribución de *ransomware* y la ejecución de códigos maliciosos. En ese mismo año, se desarrollaron las Botnets *Cutwail* y *Storm*, las cuales, fueron las responsables de generar la mayor parte de *spam* entre 2007 y 2010. *Cutwail* generaba una media de 74 billones de ataques diarios y funcionaba por medio del protocolo SMTP, mientras que *Storm* generaba 3 billones de ataques diarios amparados en el protocolo P2P [11, 24].

Durante la década del 2010 el marketing tuvo una tendencia a migrar hacia el desarrollo de anuncios en línea, esto provocó que los atacantes informáticos usen las capacidades de las Botnets para adulterar estadísticas publicitarias. En el año 2015 surgió la Botnet *Methbot*, la cual, imitaba el comportamiento de un usuario real para evitar su detección y estafar a los anunciantes. Esta Botnet generó pérdidas estimadas en 3 millones de dólares por día [24, 25].

Para el 2016 apareció una de las Botnets más conocidas y distribuidas a nivel mundial.

Esta Botnet denominada como *Mirai*, tenía el objetivo de infectar dispositivos *Internet of Things* (IoT) que ejecutaran el Sistema Operativo (SO) Linux a gran escala. Fue el medio para el despliegue del ataque distribuido de denegación de servicios (DDoS, por sus siglas en inglés) masivo a la empresa *DynDNS (Dynamic Network Services, Inc)*, el cual, es un proveedor de nombres de dominio; el ataque fue tan masivo que llegó a generar una tasa de transferencia de datos de hasta 1.2 Tbps [16, 24].

2.1.2. Ciclo de vida

El ciclo de vida de una Botnet consta de cuatro fases típicas, siendo estas: explotación, reunión, ejecución del ataque y, mantenimiento y actualización [17, 26, 27]. Cabe mencionar que como un requisito previo a la creación de una Botnet, se requiere que el *Botmaster* cuente con un nombre de dominio asociado a una dirección IP, el cual, operará como un punto de reunión para el control de los Bots.

- **Fase de explotación:** Consiste en aprovechar vulnerabilidades de *software* en los *hosts* para infectarlos a través del uso de códigos maliciosos que pueden distribuirse a través de descargas de archivos, unidades USB, correos electrónicos, entre otros. Una vez que el *host* ha sido comprometido y convertido en un Bot, ejecutará solicitudes DNS en busca de conectividad con la dirección IP del servidor CC.
- **Fase de reunión:** Etapa donde los Bots constantemente buscan comunicarse con el *Botmaster* a través del servidor CC establecido. Dado que tener una dirección IP estática es riesgoso para el encubrimiento del servidor CC, el *Botmaster* cambiará continuamente su nombre de dominio. Los Bots han sido previamente equipados con funciones dinámicas de búsquedas DNS para adaptarse a estos cambios y encontrar el servidor CC actualizado.
- **Fase de ejecución:** Una vez que la conexión entre el *Botmaster* y sus Bots se ha establecido, el *Botmaster* puede ordenar la ejecución de distintos tipos de ataques mediante el envío de instrucciones. Esto puede incluir actividades maliciosas como el envío de *spam*, ataques DDoS, robo de información o cualquier otra acción dañina que haya sido programada en el Bot.

- **Fase de mantenimiento y actualización:** Periodo donde recursivamente los Bots actualizan sus binarios a través del servidor CC con la finalidad de mantener la comunicación oculta de los sistemas de detección por el mayor tiempo posible. Después de completar estas tareas, los Bots regresan periódicamente a la fase de reunión.

Estas fases son iterativas y continúan en un bucle, permitiendo al *Botmaster* controlar y utilizar los Bots para llevar a cabo actividades maliciosas de manera persistente y encubierta. El flujo de las fases del ciclo de vida de una Botnet se presenta en la Figura 2.2.

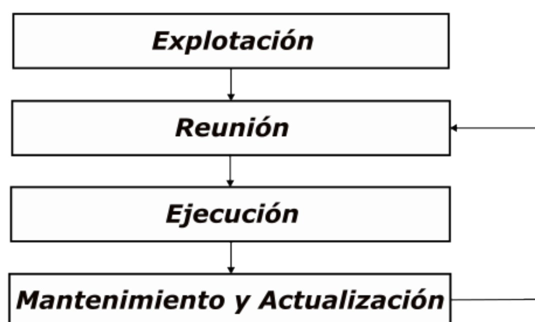


Figura 2.2: Ciclo de vida de una Botnet.

2.1.3. Estructura

Una de las fortalezas más importantes de cualquier Botnet es generar grandes redes de computadoras que tratan de mantener la mayor disponibilidad posible, para lo cual, los atacantes han desarrollado diversas estructuras de despliegue para tratar de evitar problemas en la comunicación. Entre las estructuras conocidas están:

- **Estructura centralizada:** Es un despliegue de red que emplea el enfoque de cliente/servidor, por lo cual, es la implementación más adoptada por las Botnets. En este enfoque los Bots se comunican con el servidor CC a través de una dirección IP predefinida previamente por el *Botmaster*. La comunicación se realiza mediante protocolos de comunicación como IRC o HTTP. Esta estructura de red es ampliamente usada debido a su facilidad de implementación, comunicación directa entre el servidor CC y los Bots resultando en una baja latencia

en la difusión de comandos de control, ataques con mayor coordinación y, mejor administración de la Botnet. Sin embargo, la desventaja de este enfoque es la existencia de un único punto de falla, lo cual, implica que el desmantelamiento de la Botnet se puede dar fácilmente. Es común el uso de *blocklists* para interrumpir la comunicación con el servidor CC a nivel de *firewall* [24, 28].

Los *Botmasters* han desarrollado varias técnicas para ocultar sus servidores CC, como *fast-flux* y DGA. La técnica *fast-flux* permite ocultar el servidor CC utilizando una serie de servidores *proxy* que apuntan a distintos nombres de dominio a través de múltiples direcciones IP. Al cambiar constantemente las direcciones IP asociadas con el nombre de dominio, la Botnet puede mantenerse activa, incluso si algunos nodos son detectados y desmantelados. Por otro lado, el uso de DGAs consiste en que tanto el *Botmaster* como los Bots, generen aleatoriamente listas de dominios mediante una semilla específica. El *Botmaster* debe ser el primero en hacerlo registrando periódicamente uno de los dominios, de modo que el Bot pueda consultar toda la lista generada hasta encontrar la dirección del servidor CC [24, 28].

- **Estructura descentralizada:** También conocida como “Estructura P2P”, consiste en un despliegue de red en donde cualquier nodo puede actuar como servidor CC, por lo cual, no hay un único servidor CC centralizado. Esto genera una resiliencia mayor ante sistemas de detección, y por lo tanto, corrige la debilidad de la estructura centralizada, al no poseer un único punto de falla gracias a la poca interdependencia entre los Bots. En esta estructura existe una mayor dificultad para identificar la cantidad total de Bots en la red y el alcance de la infección. Además, cada Bot tiene una lista de nodos con los cuales podrán interactuar. La comunicación entre los nodos se realiza a través de aplicaciones *web* o *software* de aplicaciones. Sin embargo, hay que tener en cuenta que este enfoque requiere mayor esfuerzo de implementación y administración. Por la naturaleza misma de la estructura de la red, la difusión de los comandos de control necesita un mayor tiempo para alcanzar a la totalidad de la Botnet. Además, dado que los nodos se conectan directamente entre sí, cada nodo debe administrar y mantener sus propias conexiones con otros nodos, volviéndose una tarea demandante

proporcionalmente al aumentar el número de nodos [24, 28].

- **Estructura híbrida:** Esta estructura combina los elementos del enfoque centralizado y descentralizado, destinando Bots adicionales para las tareas de control, administración y coordinación de la Botnet; estos Bots se denominan sirvientes ya que, actúan como clientes y servidores de manera simultánea. Además existen Bots descentralizados que actúan únicamente como clientes y son los responsables de ejecutar los ataques maliciosos [24].

  Este enfoque a diferencia de las estructuras centralizadas y descentralizadas ofrece una mayor robustez y redundancia ante sistemas de detección, ya que, evita el problema del único punto de falla de la estructura centralizada, y mejora la coordinación de la Botnet con relación a la estructura descentralizada.

  Sin embargo, este enfoque requiere un mayor esfuerzo en su implementación y administración debido a las nuevas capacidades y requerimientos de los Bots sirvientes.

Cada estructura tiene sus ventajas y desafíos. La elección de la estructura dependerá de los objetivos y las necesidades del *Botmaster* al establecer y administrar la Botnet.

2.2. Algoritmo de generación de dominios (DGA)

2.2.1. Motivación y descripción

Cuando un *host* es comprometido por un *malware* y ha sido convertido en un Bot, su principal objetivo es establecer una comunicación con el *Botmaster* en busca de instrucciones; este proceso se conoce como "Comunicación de Comando y Control" y es la etapa más determinante en el éxito de un ataque. En un inicio, las Botnets solían comunicarse con el servidor de CC a través de una dirección IP codificada en el ejecutable del *malware*. Sin embargo, este método era susceptible de ser descubierto por los administradores de la red a través de ingeniería inversa o por inspección de los archivos de registro, lo que llevaba a bloquear dicha dirección en las correspondientes listas de bloqueo (*blocklists*), destruyendo así el canal de comunicación [15, 16].

El algoritmo de generación de dominios (DGA) nace de la necesidad de mejorar el

proceso de comunicación de CC, siendo una técnica ampliamente utilizada por las Botnets para eludir la detección de los sistemas de seguridad. Por medio de rutinas pseudoaleatorias y de una semilla específica, esta técnica le permite al *Botmaster* generar varios nombres de dominio que al combinarlos con un dominio de nivel superior (TLD), pueden usarse para registrar distintos nombres de dominio completos (*Fully Qualified Domain Name*s (FQDNs)) de manera recurrente para establecer el servidor CC y así evitar que los sistemas de detección los coloquen en *blocklists*. El atacante periódicamente abandona el dominio malicioso generado algorítmicamente (mAGD) para evitar su identificación por parte de los sistemas de detección. Este proceso se repite constantemente al cambiar la semilla, lo que genera una secuencia única y diferente de dominios [15, 29–32].

2.2.2. Funcionamiento

En una Botnet basada en DGA, tanto el *Botmaster* como los Bots ejecutan de manera periódica el mismo algoritmo con una semilla específica. Sin embargo, el *Botmaster* será el primero en realizarlo con la finalidad de asociar un mAGD a la dirección IP del servidor de CC en el sistema DNS local. Posteriormente, los Bots envían consultas DNS hacia todos los mAGDs que se obtengan con la semilla escogida, hasta poder resolver una dirección IP válida. Si la consulta es exitosa, se obtendrá la dirección IP del servidor CC y entonces se establecerá la comunicación de comando y control [23, 28, 30, 33].

Prácticamente, cualquier tipo de información disponible a nivel de *host* puede ser utilizada como una semilla en la generación de los mAGDs, como, por ejemplo: la fecha actual, los temas de tendencia en *Twitter*, las previsiones meteorológicas, el valor de cambio de moneda entre diferentes países, etc [15, 21]. Sin embargo, las semillas pueden ser clasificadas en función de su naturaleza [16, 21]:

- Variables dependientes del tiempo.
- Variables independientes del tiempo.
- Basadas en diccionarios.

2.2.3. Impacto

El uso de DGAs brinda varias ventajas a los administradores de Botnets. En primer lugar, dificulta la identificación y localización de los servidores de CC por parte de los sistemas de seguridad, ya que los nombres de dominio generados son pseudoaleatorios, es decir, que no siguen un patrón predecible. Además, el hecho de cambiar periódicamente los nombres de dominio implica que los sistemas deben ser diseñados para operar en tiempo real. Esto permite que la comunicación de las Botnets se mantenga oculta y que el servidor de CC no esté expuesto durante largos periodos de tiempo. La generación de múltiples nombres de dominio proporciona redundancia al servidor CC, lo que permite que la Botnet siga operando incluso si uno de los dominios es bloqueado [28, 30].

La problemática más grande que experimentan los administradores de la red al momento de detectar y detener Botnets, es la asimetría de recursos necesarios para lograrlo. Específicamente, un *Botmaster* genera miles de posibles mAGDs pero únicamente asocia uno al servidor de CC esto implica que los administradores de la red deberán identificar todos los mAGDs para poder asegurar el desmantelamiento del canal de comunicación. Esto combinado con la ejecución periódica del (DGA) obliga que los sistemas de detección cuenten con un tiempo de respuesta muy rápido [19, 32].

2.2.4. Clasificación de DGAs

Las técnicas DGA han ido evolucionando continuamente para evitar que los mAGDs sean detectados por mecanismos de seguridad, y tal como lo manifiesta [20, 34] estos algoritmos pueden ser clasificados según el método de generación utilizado:

- **DGA basado en caracteres:** Algoritmo que genera mAGDs compuestos por caracteres aleatorios del diccionario inglés.
- **DGA basado en palabras:** Algoritmo que genera mAGDs compuestos por combinaciones aleatorias de palabras del diccionario inglés, como sustantivos, verbos y adjetivos. Estos mAGDs pueden adquirir la apariencia de un dominio legítimo ya que, en algunos casos pueden contener interpretaciones implícitas.

- **DGAs mixtos:** Algoritmo que genera mAGDs compuestos por combinaciones aleatorias de caracteres y palabras.

A continuación, en la Tabla 2.1 se muestran ejemplos de mAGDs obtenidos mediante la ejecución de diferentes familias de DGAs, pertenecientes a las clasificaciones previamente mencionadas [32].

Tabla 2.1: Ejemplos de mAGDs de cada clase.

Clase	Familia	Ejemplo
Basadas en caracteres	cryptolocker	keibffyvryomh.org
Basadas en palabras	matsnu	wedding-muscle.com
Mixta	banjori	gcjmellefrictionlessv.com

2.3. Aprendizaje automático

El aprendizaje automático es una disciplina del campo de la inteligencia artificial, que tiene como objetivo dotar a los dispositivos informáticos con la capacidad de aprender de manera autónoma a partir de datos específicos de entrenamiento [2, 35]. A través del uso de una gran variedad de algoritmos, basados en técnicas matemáticas, el aprendizaje automático permite procesar extensas cantidades de datos, extraer información relevante, identificar patrones y generar predicciones para eventos futuros [35]. Su aplicación es ideal para entornos dinámicos, donde se tienen frecuentemente nuevas entradas, y para sistemas con una extensa cantidad de datos y listas de reglas [2]. Con la ayuda de esta herramienta, se han podido evidenciar avances significativos en la resolución de problemas presentes en varios campos, como la ciberseguridad, medicina, economía, entre otros [36].

Una manera de clasificar a los modelos de aprendizaje automático, es en función del nivel de supervisión proporcionado durante el entrenamiento, a continuación se muestran las dos principales clasificaciones [2, 37]:

- **Aprendizaje supervisado:** En este enfoque, el modelo de aprendizaje es entrenado con un conjunto de datos, que contiene pares de ejemplos, conformados por las características de entrada y las correspondientes salidas (o etiquetas) deseadas [36, 37]. Recibe su nombre por la analogía con la supervisión de un profesor, el cual en el transcurso del entrenamiento, guía al alumno (algoritmo),

con la respuesta deseada ante una entrada específica [36]. El principal objetivo consiste en que el algoritmo pueda aprender a partir de estos ejemplos y posteriormente realizar una predicción para un nuevo dato de entrada [2].

Existen dos aplicaciones comunes que se basan en este tipo de aprendizaje: la clasificación y regresión de datos. En la clasificación, el objetivo del algoritmo es predecir la etiqueta o clase correspondiente a una entrada específica, para esto se requiere un conjunto preestablecido de clases. Por otro lado, en la regresión, el algoritmo se enfoca en predecir un valor numérico asociado a la entrada proporcionada [36].

- **Aprendizaje no supervisado:** A diferencia del aprendizaje supervisado, este tipo de modelo no utiliza un conjunto de entrenamiento con etiquetas de salida para las características de entrada. En su lugar, su objetivo es extraer conocimiento a partir de los valores proporcionados como entrada, y así lograr descubrir patrones o estructuras que puedan influir en las predicciones ante nuevos datos [36, 37].

2.3.1. Algoritmo *K-Nearest Neighbors* (KNN)

Es un algoritmo de aprendizaje supervisado ampliamente utilizado en la clasificación y regresión de datos, se considera uno de los algoritmos más simples en términos de funcionamiento, sin embargo, representa una herramienta poderosa y versátil utilizada en varios sistemas y aplicaciones [2, 36].

Para predecir una nueva muestra, el algoritmo identifica los K vecinos más cercanos a ella, lo que implica calcular la distancia que existe entre la nueva muestra y los ejemplos del conjunto de entrenamiento. Existen varias métricas de distancia para este propósito, como por ejemplo: la distancia *Euclidiana*, *Manhattan*, *Hamming*, *Tanimoto*, entre otras. La distancia de *Minkowski* generaliza a varias de estas y se define como muestra la Ecuación (2.1) [38].

$$\left(\sum_{j=1}^{P} |x_{aj} - x_{bj}|^{q}\right)^{\frac{1}{q}} \tag{2.1}$$

Donde:

- q es el orden de la distancia. Para $q = 1$ se tiene la de *Manhattan* y para $q = 2$ la *Euclidiana*.
- a y b son dos puntos en un espacio P-dimensional
- x_{aj} es la coordenada j-ésima del punto a.
- x_{bj} es la coordenada j-ésima del punto b.

El tipo de métrica de distancia a utilizar, dependerá en gran medida al contexto del sistema o aplicación y puede influir en la precisión de las predicciones [2].

En [36], se menciona que la forma más básica del algoritmo KNN se da cuando $K = 1$, es decir, que se selecciona únicamente al vecino más cercano a la muestra. Este vecino tiene asociada una salida o etiqueta conocida, la cual se toma directamente como la predicción a la nueva muestra de entrada.

También se puede considerar un número arbitrario para K, en este caso, si se utiliza el algoritmo KNN como clasificador, la predicción es la etiqueta que se encuentra con mayor frecuencia entre los vecinos más cercanos. Sin embargo, si es utilizado para la regresión de datos, se puede utilizar métricas estadísticas, como la media y mediana, para determinar la predicción en base a las salidas de las muestras vecinas. Es importante indicar, que si existe un empate entre dos o más clases a la cual pertenezca la nueva muestra de entrada, se selecciona aleatoriamente una de las clases o se observa al $K + 1$ vecino más cercano [36, 38].

Si bien es posible asignar cualquier número arbitrario a K, para los diferentes sistemas o aplicaciones, existirá un valor óptimo que permitirá obtener mejores resultados en la clasificación o regresión de datos. Este valor óptimo se puede hallar mediante un proceso de muestreo [36], donde se evalúa la precisión o el error en función del valor de K utilizado, para posteriormente seleccionar el valor que permitió optimizar el algoritmo.

Como el funcionamiento del algoritmo se basa en las distancias entre muestras, se debe tener en cuenta que las escalas en las que se encuentran las diferentes características (predictores) en un sistema en particular, pueden tener un impacto negativo

en el desempeño del algoritmo. Si las escalas son muy diferentes, puede existir un sesgo en el cálculo de las distancias, es decir, las características con escalas más grandes pueden influir dominantemente en la distancia total calculada. Para evitar esta situación, se recomienda centrar y escalar todas las características antes de aplicar el algoritmo KNN [2, 38]. Además, si se trabaja con una cantidad considerable de datos, este algoritmo puede representar costos computacionales importantes para sus desarrolladores. El tiempo necesario para calcular las distancias entre muestras y la memoria que se necesita para guardar estos datos pueden representar una desventaja en el uso de este algoritmo. Otra problemática a considerar, surge cuando no existen algunos valores correspondientes a las características para una muestra específica, en este caso, no se puede calcular la distancia entre dos muestras correctamente, por lo tanto, es posible que el algoritmo deba omitir estas características a la hora de calcular la distancia, o utilizar a su vez valores más recurrentes en las demás muestras [36].

2.3.2. Algoritmo *Support Vector Machine* (SVM)

Se trata de un algoritmo de aprendizaje automático supervisado, el cual representa una herramienta potente en la clasificación y regresión de datos, siendo utilizado tanto en modelos lineales como en no lineales. Permite identificar valores atípicos e incluso minimizar sus efectos en la regresión [2, 38]. Las áreas en las cuales se utiliza ampliamente este algoritmo, incluyen la visión por computadora, categorización de textos, bioinformática, entre otros [39].

El funcionamiento de este algoritmo, se basa en determinar un hiperplano que divida las muestras en dos categorías específicas. Para lograr esto, se define una métrica denominada como margen, la cual representa la distancia entre las muestras limite de dos categorías, tal como lo muestra la Figura 2.3 [38]. En este contexto, se conoce como hiperplano separador óptimo, a aquel que permite tener el máximo margen de separación entre las dos categorías [1, 39].

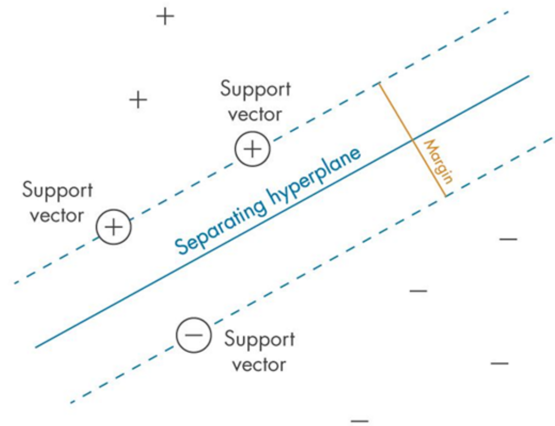


Figura 2.3: Hiperplano separador óptimo para el algoritmo SVM [1].

Si se tiene un conjunto de entrenamiento previamente categorizado en dos clases, las cuales han sido codificadas con los valores 1 y -1, y los vectores x_i representan las características para una muestra x específica de dicho conjunto, la clasificación basado en el hiperplano separador óptimo crea un valor de decisión $D(x)$ que se evalúa en $D(x) > 0$ para categorizar a la muestra en la clase 1 o en caso contrario a la -1. De manera similar, es posible definir y generalizar la ecuación de decisión para una nueva muestra de entrada mediante una función discriminante lineal, que se especifica en función de las características del hiperplano separador como se muestra en la Ecuación (2.2) y su forma extendida en la Ecuación (2.3) [38].

$$D(\mathbf{u}) = \beta_0 + \beta'\mathbf{u} \tag{2.2}$$

$$D(\mathbf{u}) = \beta_0 + \sum_{j=1}^{P} \beta_j u_j \tag{2.3}$$

La Ecuación (2.3) puede expresarse en función de los datos asociados a cada muestra como lo indica la Ecuación (2.4) [38].

$$D(\mathbf{u}) = \beta_0 + \sum_{i=1}^{n} y_i \alpha_i \mathbf{x_i}'\mathbf{u} \tag{2.4}$$

Donde:

- $\mathbf{x_i}$ es un vector que representa las características para la i-ésima muestra del conjunto de entrenamiento.
- α_i es el coeficiente asociado con los vectores de soporte para la i-ésima muestra, $\alpha_i \geq 0$.
- y_i es la etiqueta de clase para la i-ésima muestra del conjunto de entrenamiento.
- n es el número total de muestras.
- $\mathbf{u}$ es el vector de características de la muestra de entrada.

Cuando los datos de entrenamiento son completamente separables, los coeficientes α de la Ecuación (2.4) son cero para todas las muestras que no se encuentran en el margen. Por otra parte, los valores de α distintos de cero corresponden a las muestras que sí se encuentran en el límite del margen, es decir, los puntos más cercanos al hiperplano de separación. Estos puntos, con coeficientes α distintos de cero, se conocen como vectores de soporte, la ecuación de predicción está en función de estos valores, por este motivo a la clasificación de margen máximo se la conoce como máquina de vectores de soporte [1, 38].

El modelo presentado, puede ser extendido a sistemas no lineales mediante el uso del truco del *kernel* [38]. Esta técnica se basa en la idea de que aquellos datos que no son separables de manera lineal, en su espacio dimensional original, pueden volverse separables en un espacio dimensional superior. Para esto, se debe mapear los datos a un espacio de mayor dimensión, mediante funciones *kernel* específicas y posteriormente encontrar el hiperplano óptimo del sistema [36, 40]. Por lo tanto, la Ecuación (2.4), puede ser generalizada en la Ecuación (2.5).

$$D(\mathbf{u}) = \beta_0 + \sum_{j=1}^{P} y_i \alpha_i K(\mathbf{x_i}, \mathbf{u}) \tag{2.5}$$

El término $K(\mathbf{x_i}, \mathbf{u})$ representa la función *kernel* entre los vectores $\mathbf{x_i}$ y $\mathbf{u}$, para el caso lineal esta función es únicamente el producto punto entre los dos vectores. Sin embargo para sistemas no lineales se pueden utilizar distintas funciones. Por ejemplo, en la Ecuación (2.6) se define el *kernel* polinomial, donde d corresponde al grado del polinomio. Otra opción comúnmente utilizada es la función de base radial gaussiana

(RBF, por sus siglas en inglés), la cual está definida en la Ecuación (2.7), donde el valor de γ es una variable de control de amplitud. Finalmente, en la Ecuación (2.8) se muestra el *kernel* basado en la función *sigmoid* [38].

$$Polinomial = (\mathbf{x}'\mathbf{u} + 1)^d \tag{2.6}$$

$$RBF = exp(-\gamma \left\| \mathbf{x} - \mathbf{u} \right\|^2) \tag{2.7}$$

$$Sigmoid = tanh(\mathbf{x}'\mathbf{u} + 1) \tag{2.8}$$

La elección del *kernel* a utilizar, para un conjunto específico de datos, debe basarse en una serie de experimentos variando sus parámetros característicos, no obstante, se recomienda en primera instancia, probar el *kernel* lineal, sobre todo si la longitud de datos es extensa o el número de características es considerable [36, 40].

2.3.3. Algoritmo *Random Forests*

Es un algoritmo de aprendizaje automático supervisado que pertenece a la categoría de *Ensemble Learning* (Aprendizaje Conjunto). Puede emplearse tanto en tareas de clasificación como de regresión, y destaca como uno de los modelos más potentes y populares en la actualidad [2].

2.3.3.1. Algoritmo *Decision Trees*

Este algoritmo es el componente principal en el algoritmo de “Bosques Aleatorios”, perteneciendo también a los modelos de aprendizaje automático supervisado. Los “Árboles de Decisión”, agrupan jerárquicamente múltiples declaraciones $if - then$ hasta conducir a una decisión. Cada declaración es denominada como “Nodo”, en donde, el nodo inicial se denomina como “Raíz” y el nodo que realiza la toma de la decisión final se denomina como “Nodo Terminal” u “Hoja”; en otras palabras, los nodos representan cada una de las preguntas que se realizan durante la clasificación

o la regresión de un conjunto de muestras, mientras que las hojas representan las respuestas obtenidas por el modelo [2, 36].

El funcionamiento de un árbol de decisión ya sea en tareas de clasificación o regresión, consiste en seguir la trayectoria generada por las distintas declaraciones $if - then$ al evaluar el conjunto de muestras de interés. Durante el entrenamiento, se seleccionan todas las declaraciones $if - then$ en función de la característica más informativa que permita identificar la dinámica del conjunto de muestras, hasta llegar a una hoja. Por lo tanto, existirá una única trayectoria para cada muestra del conjunto total de muestras [38].

Dado que los conjuntos de muestras generalmente poseen características representadas en un dominio continuo, las declaraciones $if - then$ buscan determinar si una muestra tiene características en un rango específico, por medio de preguntas como: "*¿La característica i es mayor al valor umbral a?*". Cada declaración divide en un subconjunto nuevo las muestras del nodo anterior. Dado que cada evaluación solo considera una característica a la vez, los subconjuntos forman regiones paralelas a los ejes. Esta división se repite recursivamente hasta que la hoja contenga únicamente muestras de una sola clase o un solo valor de regresión; las hojas de este estilo se denominan como "Hojas Puras". Un árbol con únicamente hojas puras, es totalmente preciso en las muestras de entrenamiento [36].

La diferencia entre un árbol de decisión de clasificación y uno de regresión radica en la manera en la que la hoja realiza la decisión final; para la clasificación, la hoja predecirá un valor discreto (una clase definida), mientras que para la regresión, la hoja predecirá un valor continuo obtenido al promediar los valores objetivos de todas las muestras en dicha hoja [2, 36].

Entre las ventajas y desventajas que posee este algoritmo, se tienen [38]:

- **Ventajas:**
  1. Al finalizar el entrenamiento, la lógica detrás del funcionamiento del modelo generado puede ser interpretada muy fácilmente de manera visual, incluso sin la necesidad de tener conocimientos previos sobre él.
  2. No se necesita de un preprocesamiento de los datos de entrenamiento co-

mo la normalización o el centrado de características.

3. Puede ajustar conjuntos de muestras complejos tanto en tareas de clasificación como de regresión.

- **Desventajas:**

1. Dado que los límites de decisión son ortogonales, el rendimiento del árbol es muy sensible a la rotación de las muestras de entrenamiento. Si las características no pueden ser distinguidas correctamente mediante subespacios rectangulares, los modelos generados presentarán un notable error de predicción.

2. El modelo generado puede llegar a ser muy diferente si las muestras de entrenamiento varían levemente, denotando una inestabilidad marcada en este algoritmo.

3. Construir un árbol en el cual todas sus hojas sean puras conduce a modelos muy complejos y muy ajustados a los datos de entrenamiento (sobreajuste). Existen dos estrategias comunes para evitar el sobreajuste: el podado previo (*pre-pruning*), en donde se detiene la creación del árbol antes de tiempo y el podado posterior (*post-pruning*), en donde se eliminan los nodos que contienen poca información luego de la construcción del árbol. Sin embargo, aún podando el árbol, estos tienden a sobreajustarse, no rindiendo óptimamente.

Para lidiar con el problema de la sensibilidad a la rotación, es común utilizar la técnica de “Análisis de Componentes Principales” para orientar de mejor manera la muestras de entrenamiento. Por otra parte, en el proceso del podado previo, se suelen utilizar criterios como la profundidad máxima del árbol, la cantidad máxima de hojas o la cantidad mínima de muestras en un nodo, como criterios de parada durante la construcción del árbol de decisión, con el motivo de evitar el sobreajuste. En el proceso de podado posterior, en cambio, se eliminan los nodos innecesarios luego de la construcción del árbol, determinando dichos nodos a través de pruebas estadísticas como la prueba de “chi-cuadrado”, al evaluar si el incremento de la pureza en un nodo con respecto al anterior, no es significativo (superior al 5 %) [2].

Si bien las técnicas anteriores permiten mejorar el rendimiento de los árboles de decisión, es ampliamente recomendable emplear técnicas de aprendizaje conjunto en su lugar, ya que permiten generar modelos de aprendizaje óptimos y estables. El algoritmo de bosques aleatorios, nace de agrupar múltiples árboles de decisión en un solo modelo de aprendizaje, presentando un rendimiento predictivo mucho mejor que el de los árboles individuales [38].

2.3.3.2. Aprendizaje conjunto

Es una técnica de aprendizaje automático que se basa en el fenómeno estadístico conocido como "Sabiduría de la Multitud". Este fenómeno establece que entre más modelos de aprendizaje colaboren para realizar una predicción, el sesgo irá en descenso, llegando a generar incluso predicciones más precisas que las obtenidas por el mejor predictor individual [2].

En un modelo de aprendizaje conjunto existen dos enfoques para realizar la predicción. Si la predicción se realiza seleccionando la clase con la mayor cantidad de votos, el modelo es denominado de "Votación Dura". Si los modelos individuales pueden determinar las probabilidades de cada clase, entonces la predicción puede realizarse promediando las predicciones, generando un modelo denominado de "Votación Suave" [36].

Para que el modelo de aprendizaje conjunto sea factible de implementar se debe cumplir que los modelos individuales sean no correlacionados, con el objetivo de evitar la ocurrencia de errores iguales entre modelos distintos. Esto se puede lograr al utilizar modelos de aprendizaje muy diferentes o entrenando el mismo modelo de aprendizaje varias veces con subconjuntos de muestras aleatorias o incluso con características diferentes [2].

En base a lo anterior, el algoritmo de bosques aleatorios entrena distintos árboles de decisión con subconjuntos aleatorios de muestras de entrenamiento, o de características, aunque incluso se puede realizar una combinación de ambos enfoques. En otras palabras, un bosque aleatorio está formado por un grupo árboles de decisión distintos entre sí, que pueden generalizar bien un subconjunto de muestras de entrenamien-

to, pero sobreajustarse en otros. Sin embargo, al promediar todas las predicciones individuales se logra abordar el sobreajuste mencionado [36]. El funcionamiento del algoritmo de bosques aleatorios puede explicarse por medio del pseudocódigo mostrado en el Extracto 2.1.

```
Definir la cantidad m de Arboles de Decision a crear
for i = 1 to m do
    Seleccionar un subconjunto de muestras de entrenamiento
    Entrenar el arbol en el subconjunto seleccionado
    for each split do
        Seleccionar aleatoriamente k (< Total) caracteristicas
        Seleccionar la mejor caracteristica entre las k
            seleccionadas y dividir los datos
    end
    Usar el criterio de parada escogido
end
```

Extracto de código 2.1: Algoritmo *Random Forests* básico [36].

2.3.4. Algoritmo *Multi-Layer Perceptron* (MLP)

Los procesos biológicos han inspirado directamente el funcionamiento de muchas de las invenciones tecnológicas actuales. Las redes neuronales artificiales (ANN, por sus siglas en inglés) surgieron con el objetivo de intentar crear una máquina inteligente tomando como inspiración la arquitectura del cerebro. Si bien, las ANNs inicialmente se inspiraron en las redes neuronales biológicas, con el paso del tiempo han ido evolucionando y se han desarrollado modelos de aprendizaje automático mucho más sofisticados. Dichos modelos son parte integral del campo conocido como *Deep Learning* (aprendizaje profundo), que engloba a un grupo de algoritmos versátiles y potentes para abordar tareas complejas de aprendizaje automático [2].

2.3.5. Perceptrón

La arquitectura más básica que una ANN puede presentar, es el perceptrón, que consta únicamente con una capa de neuronas artificiales del tipo TLU. Cada TLU se en-

cuentra conectada a todas las entradas y tanto las entradas como las salidas son valores numéricos. Esta unidad se encarga de calcular una suma ponderada de las entradas a través de la Ecuación (2.9) y utiliza esta suma en una función de paso para generar la salida correspondiente tal como se muestra en la Ecuación (2.10) [2].

$$z = w_1x_1 + w_2x_2 + ... + w_nx_n = \mathbf{x^T w} \tag{2.9}$$

$$h_w(\mathbf{x}) = step(z) = step(\mathbf{x^T w}) \tag{2.10}$$

Donde:

- $\mathbf{x}$ es el vector de entrada.
- $\mathbf{w}$ es el vector de pesos asociados a cada entrada.
- $h_w(\mathbf{x})$ es la salida numérica calculada.

La arquitectura de una TLU para un caso de 3 entradas, se muestra en la Figura 2.4.

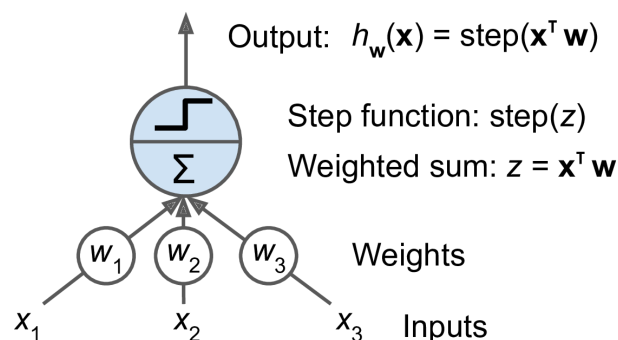


Figura 2.4: Arquitectura de una TLU [2].

En el caso de los perceptrones, se suele utilizar la función escalonada de *Heaviside* mostrada en la Ecuación (2.11) o la función signo de la Ecuación (2.12), como función de paso:

$$heaviside(z) = \begin{cases} 0 & \text{si } z < 0 \\ 1 & \text{si } z \geq 0 \end{cases} \tag{2.11}$$

$$sgn(z) = \begin{cases} -1 & \text{si } z < 0 \\ 0 & \text{si } z = 0 \\ +1 & \text{si } z > 0 \end{cases} \quad (2.12)$$

Con el uso de una única neurona TLU, es posible realizar tareas de clasificación binarias lineales simples. Entrenar una neurona TLU para este caso implica el encontrar los valores adecuados para el vector $\mathbf{w}$. Cabe mencionar que por si solos, los perceptrones no poseen la capacidad de resolver problemas con un grado de dificultad mayor, como por ejemplo, la clasificación "XOR" [2].

2.3.6. *Multi-Layer Perceptron* (MLP)

Para superar las limitaciones presentadas en los perceptrones, surgió una arquitectura de ANN denominada como perceptrón multicapa, la cual, soluciona estos problemas al apilar múltiples perceptrones en una sola estructura. En un MLP se tienen los siguientes componentes [2, 36]:

- Una capa de entrada.
- Una o más capas ocultas de neuronas TLU.
- Una capa de salida de neuronas TLU.

Al apilar varios perceptrones es necesario realizar múltiples sumas ponderadas. Inicialmente se calculan las sumas de las capas ocultas para combinarlas en una suma ponderada de salida. Hay que tener en cuenta que existirán múltiples pesos entre cada capa involucrada [2].

Dado que la combinación de múltiples sumas ponderadas puede abstraerse como una única suma ponderada, implicaría que apilar varios perceptrones carecería de un propósito funcional. Es por esto que en un MLP se agregan no linealidades entre cada suma ponderada, reemplazando la función de paso con funciones de activación no lineales como: la logística (sigmoidea) de la Ecuación (2.13), la unidad lineal rectificada (ReLU, por sus siglas en inglés) de la Ecuación (2.14) o la tangente hiperbólica (tanh) de la Ecuación (2.15). La introducción de estas funciones de activación conlleva a que

los MLP sean teóricamente capaces de aproximarse a cualquier función continua [2].

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.13}$$

$$ReLU(z) = max(0, z) \tag{2.14}$$

$$tanh(z) = 2\sigma(2z) - 1 \tag{2.15}$$

En los modelos de MLP, inicialmente los pesos de toda la red se determinan de manera aleatoria, lo que significa que pueden generarse modelos muy diferentes aun si estos comparten los mismos parámetros. Sin embargo, esto no implica cambios abruptos en la precisión en los resultados. Una vez establecidos los pesos iniciales, se emplea el algoritmo de retropropagación para ajustar cada peso de conexión hasta que la red converja a una solución. Además, es necesario someter a los datos de entrada a un proceso de reescalado de tal manera que las características varíen de igual forma, buscando alcanzar una media 0 y una varianza 1 de manera ideal [2].

Entre las ventajas y desventajas que posee este algoritmo, destacan [2]:

- **Ventajas:**
  1. Tienen la capacidad de ajustarse a grandes cantidades de muestras y características, permitiendo construir modelos complejos.
  2. Presentan una gran flexibilidad de configuración al poder modificar el número de capas ocultas, el número de neuronas en cada capa y las funciones de activación utilizadas, para adaptar la red a diferentes tipos de problemas y dominios de datos.
- **Desventajas:**
  1. Al tratar con conjuntos de datos grandes y complejos, el entrenamiento suele extenderse prolongadamente.
  2. Los datos deben ser preprocesados antes del entrenamiento.

3. Mayor complejidad computacional en el ajuste de los pesos de la red cuando se utilizan múltiples capas ocultas.
4. Para características muy diferentes, los modelos de aprendizaje basados en el uso de árboles suelen entregar mejores resultados.

2.4. Clasificación de sistemas de detección de Botnets

A lo largo de los años han surgido numerosas técnicas de detección de Botnets como medidas de seguridad ante diversos tipos de ataques. Estas técnicas se han clasificado a través de varios enfoques en función del autor. Entre las clasificaciones más utilizadas se encuentran las técnicas basadas en el análisis del tráfico DNS. En [41] se presenta la clasificación correspondiente a este enfoque:

- **Detección basada en flujo *(flow-based detection) y (flux-based detection)***: Esta técnica implica la extracción y el análisis de varios parámetros propios del flujo de datos o del flujo IP en la red, en base a esto, el sistema puede clasificar el trafico como benigno o malicioso. Algunos parámetros comúnmente usados para este propósito son: el número de paquetes, *bytes* por paquete, *bytes* por segundo, número de protocolos de enlace TCP de tres vías, número de consultas DNS, número de distintas direcciones IP resueltas, número de respuestas DNS respondidos, entre otros. El tráfico malicioso originado por las Botnets puede provocar desviaciones inusuales en el flujo de datos o en el flujo IP de la red, los sistemas de detección basados en este enfoque tienen como objetivo detectar estas variaciones.
- **Detección basada en anomalías:** Los sistemas que han sido comprometidos y que forman parte de una Botnet, presentan un comportamiento inusual en comparación con los sistemas íntegros. Por lo tanto, esta técnica se centra en encontrar patrones anómalos en la red causados por sistemas comprometidos que forman parte de una Botnet. Se analizan parámetros como el número de consultas DNS fallidas, valores de TTL y asignaciones de DNS-IP para identificar valores atípicos. La detección de estas anomalías puede indicar la presencia de tráfico malicioso asociado a las Botnets.

- **Detección basada en DGA:** El objetivo de este tipo de sistema de detección es distinguir entre los nombres de dominio maliciosos, generados a través de DGAs por *Botmasters* para ocultar sus servidores CC, y aquellos que han sido creados de manera legítima.

  Este tipo de sistemas se enfocan en el análisis y extracción de componentes de las consultas DNS realizadas por los *hosts* en una red, centrándose en el nombre de dominio, ya que comúnmente los dominios que han sido generados a través de DGAs difieren considerablemente en comparación con los legítimos. Por lo tanto, es posible emplear enfoques de detección basados en características diferenciadoras entre nombres de dominio benignos y maliciosos, y posteriormente utilizar aprendizaje automático para lograr clasificarlos y así detectar Bots en una red [34].

  A su vez, estos sistemas de detección se pueden dividir en dos categorías: clasificadores que únicamente requieren el nombre de dominio y clasificadores que requieren de información contextual adicional para la detección, como direcciones IP, patrones de consulta, búsqueda de coincidencias en *blocklists* conocidas, entre otros [42].

- **Detección de infecciones por Bots:** Este enfoque se centra en identificar los *hosts* que han sido comprometidos y forman parte de la Botnet. Se utilizan parámetros como la cantidad de FQDNs y direcciones IP configuradas en el dispositivo, así como la comprobación de su existencia en *blocklists* de servicios de seguridad. Mediante esta detección, se busca identificar los sistemas infectados y prevenir su participación en actividades maliciosas.

El presente trabajo tiene como objetivo incluir dentro del *framework* BNDF, la implementación de dos sistemas de detección de Botnets pertenecientes al enfoque basado en DGAs: *MaldomDetector* y N-gramas enmascarados. Ambos sistemas basan su enfoque de detección en la extracción de características léxicas y estadísticas en conjunto con técnicas de aprendizaje automático. Han sido seleccionados debido a su precisión superior con respecto al algoritmo de detección de mAGDs, RMA, implementado por defecto en el *framework* BNDF.

Trabajos relacionados

En este capítulo se revisan varias soluciones relevantes en el ámbito de la detección de Botnets, y específicamente, se profundiza en la arquitectura, configuración y bases teóricas de dos sistemas de detección de Botnets de interés (*MaldomDetector* y “N-gramas enmascarados”) los cuales usan aprendizaje automático supervisado y representan opciones viables en términos de implementación. Además, se detalla el funcionamiento del *framework* BNDF, el cual, es el sistema en el que se incluyen las contribuciones de este trabajo.

Dado que la presente propuesta se enfoca en implementar algoritmos detección de Bots basados en DGA, se destacan los siguientes trabajos:

- En [43] se propone una técnica que consiste en la captura constante y pasiva de los flujos DNS en el *Gateway* de una red supervisada con el objetivo de extraer parámetros del tráfico DNSs, como por ejemplo la longitud y el valor esperado de un nombre de dominio, para distinguir si corresponde a un mAGD o a un dominio legítimo. La propuesta basa su fiabilidad (precisión general promedio de hasta el 92.3 %) en una fase de entrenamiento, en donde, en función de datos previamente clasificados como benignos y maliciosos, genera modelos de detección de dominios maliciosos.
- En [44] se propone una técnica basada en un enfoque de medición de la aleatoriedad a través de un análisis léxico de los nombres de dominio en función del lenguaje. Para ello, se analizan las diferencias perceptibles entre nombres de dominios benignos y maliciosos en función de los caracteres que los componen. Debido a su dependencia del lenguaje, esta propuesta ha implementado un motor de búsqueda *web* para estimar la aleatoriedad del dominio. Los resultados demostraron una precisión de detección de DGAs del 90.29 %.
- En [45] se muestra una implementación que utiliza el análisis de caracteres mediante un enfoque de N-gramas, el cual, es un modelo semántico que caracteriza la relación entre los morfemas vecinos, en conjunto con una red neuronal convolucional profunda. Los resultados obtenidos destacan una precisión promedio del 98.29 % cuando se utilizó una estructura 3-gramas.

- Al analizar el trabajo de titulación realizado en el año 2021 en el área de Electrónica y Telecomunicaciones definido en [46], así como el artículo científico [4] denominado, *Real-time Bot Infection Detection System Using DNS Fingerprinting and Machine-learning*, se observa el desarrollo y la optimización de un sistema de detección de Bots denominado como *BotNet Detection Framework* (BNDF), el cual, se basa en el análisis del tráfico DNS de una red de una Institución de Educación Superior (IES). Este sistema, entre sus técnicas de detección cuenta con un algoritmo denominado *Randomness Measurement Algorithm* (RMA), el cual, en base a atributos específicos capturados de los flujos DNS, permite medir la aleatoriedad del nombre de dominio de entrada. Usando únicamente un detector de anomalías, se determinó que el 6.4 % de los *hosts* analizados en la IES durante 10 días fueron Bots. Sin embargo, entre estos casos detectados se encontraron varios falsos positivos. Al emplear el algoritmo RMA, se determinó que únicamente el 0.003 % fueron casos confirmados de Bots basados en DGA.

  Finalmente, cabe mencionar que el sistema de detección descrito en [3], denominado como *MaldomDetector*, a diferencia de RMA, complementa su funcionalidad con aprendizaje automático, requiriendo listas debidamente etiquetadas con nombres de dominio benignos y maliciosos, así como varios atributos relacionados con los caracteres de cada nombre de dominio. La precisión de detección de Bots al emplear *MaldomDetector* es del 97.82 %, superando la precisión del algoritmo RMA que es del 83.14 %.

3.1. Sistema de detección *MaldomDetector*

MaldomDetector es un sistema de detección de dominios maliciosos presentado en [3], el cual, emplea técnicas de aprendizaje automático supervisado, a través de la extracción de características intrínsecas del nombre de dominio. Debido a que el conjunto de características que extrae *MaldomDetector* es fácil de determinar, puede detectar tempranamente comunicaciones basadas en DGA, posibilitando a los sistemas de seguridad, neutralizar los canales de CC.

La arquitectura de este sistema, se presenta en la Figura 3.1, constando de dos mó-

dulos secuenciales.

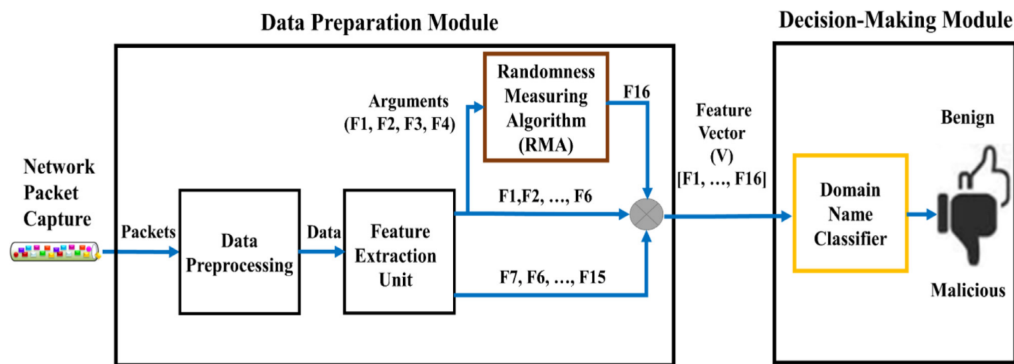


Figura 3.1: Arquitectura de *MaldomDetector* [3].

1. **Módulo de preparación de datos:** Es el encargado de adecuar los paquetes de solicitud DNS con el objetivo de obtener un vector de características V para cada nombre de dominio solicitado. Este vector esta conformado por dos tipos de características. Las características básicas son las extraídas directamente del nombre de dominio, mientras que las características derivadas son las que se obtienen a partir de las básicas. Inicialmente, se consideró que un nombre de dominio podía ser clasificado como benigno o malicioso mediante el uso de 16 características intrínsecas. Sin embargo, mediante un análisis de correlación de los componentes principales, los autores de [3] determinaron que únicamente basta el uso de 12 por cada vector V, evitando así, el uso de las características F9, F11, F13 y F14. En la Tabla 3.1, se muestran las 12 características seleccionadas que se emplean por *MaldomDetector* para la clasificación.

Tabla 3.1: Características básicas y derivadas.

Tipo	Característica	Nombre	Descripción
Básicas	F1	entropy	La entropía del nombre de dominio.
	F2	max-sequential-consonants	Máximo número de letras consonantes secuenciales encontradas dentro del dominio.
	F3	max-sequential-vowels	Número máximo de letras vocales secuenciales encontradas dentro del dominio.
	F4	length-domain	Longitud del nombre de dominio.
	F5	consonants	El número total de letras consonantes del dominio.
	F6	vowels	El número total de letras vocales del dominio.
Derivadas	F7	ratio-entropy-to-length-domain	La razón entre la entropía y la longitud del dominio.
	F8	ratio-consonants-to-vowels	La razón entre el número total de letras consonantes y el número total de letras vocales del dominio.
	F10	ratio-vowels- to-length-domain	La razón entre el número total de letras vocales y la longitud del dominio.
	F12	ratio-max-sequential-vowels-to-length-domain	La razón entre el máximo número de letras de vocales secuenciales y la longitud del dominio.
	F15	ratio-max-sequential-consonants-to-max-sequential-vowels	La razón entre el máximo número de letras de consonantes secuenciales y el máximo número de letras de vocales secuenciales del dominio.
	F16	Randomness	La salida del algoritmo RMA

La característica F16, denominada como **Randomness**, es el algoritmo de detección de mAGDs empleado por defecto en BNDF, el cual, resulta ser determinista y ha sido creado con el objetivo de medir la aleatoriedad de los caracteres

de un nombre de dominio, en pos de inferir si su procedencia se dio de manera legítima o a través de un DGA. En la Figura 3.2 se puede observar el funcionamiento del algoritmo en un formato de diagrama de flujo.

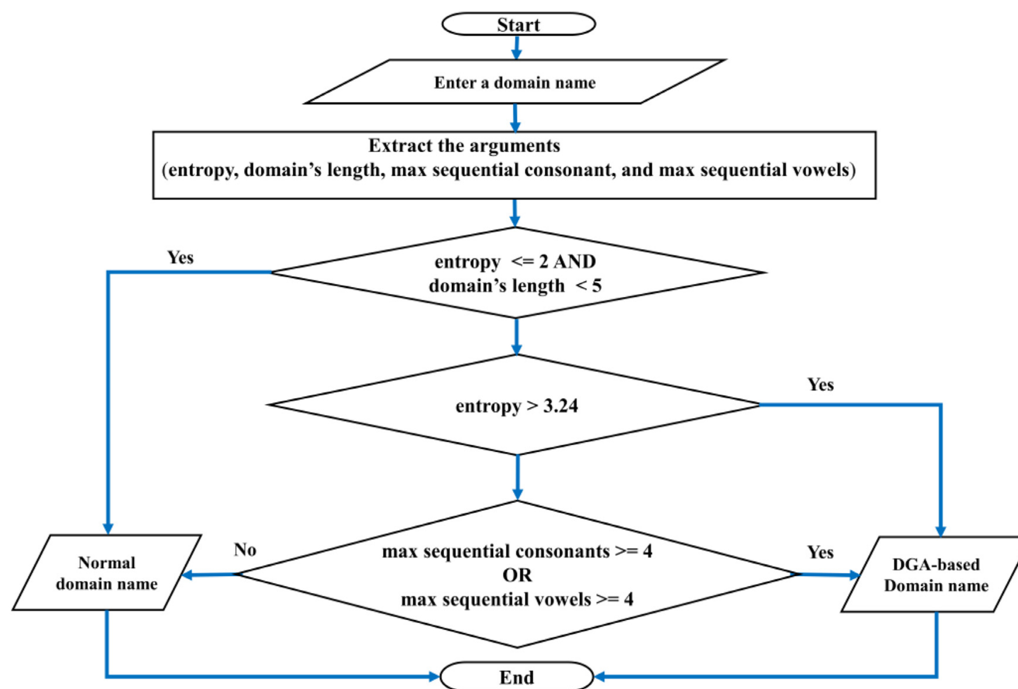


Figura 3.2: *Randomness Measurement Algorithm* (RMA) [3].

El algoritmo RMA emplea las características básicas de: la entropía (F1), el máximo número de consonantes secuenciales (F2), el máximo número de vocales secuenciales (F3) y la longitud del nombre de dominio (F4). En [3] se han definido umbrales para cada característica mencionada como se mostró en la Figura 3.2, los cuales, combinados con condicionales permiten medir la aleatoriedad propia de un dominio. Este parámetro es de interés debido a que los dominios generados por un DGA, en su gran mayoría contienen cadenas sin un sentido legible, es decir, contienen una aleatoriedad alta.

2. **Módulo de toma de decisiones:** El vector de características V de un dominio se usa como la entrada del módulo de toma de decisiones. Este módulo consta de varios clasificadores binarios basados en aprendizaje automático, los cuales, deciden si un nombre de dominio es benigno o malicioso.

   Para entrenar los modelos de aprendizaje, en [3] recopilaron dos grupos de muestras de igual cantidad, haciendo un total de 170000 muestras. Las mues-

tras etiquetadas como maliciosas fueron obtenidas del sitio [47], mientras que el grupo de muestras etiquetadas como benignas se obtuvieron de [48]. Cabe mencionar que se ha utilizado la técnica de validación cruzada con 10 *folds* para cada modelo de aprendizaje con el objetivo de lograr una generalización eficiente en la clasificación. Se entrenaron los siguientes 5 clasificadores basados en aprendizaje automático, siendo sus resultados los mostrados en la Tabla 3.2.

Tabla 3.2: Hiperparámetros utilizados y resultados obtenidos en la evaluación de modelos [3].

Nombre del Algoritmo	Hyperparámetros	Accuracy (Validación cruzada 10-fold)	FPR	Precision	Recall	F1 score
Decision tree (Fine Tree)	Max no. of splits: 100 Split criterion: Gini's diversity index Surrogate decision splits: off	94.39	0.05	0.94	0.94	0.94
Ensemble (Boosted Tree)	Ensemble method: Ada boost max no. of splits: 20 No. of learners: 30 Learning rate: 0.1	94.42	0.05	0.94	0.94	0.94
Naïve Bayes (Gaussian)	Distribution names: Gaussian	92.33	0.08	0.91	0.92	0.92
SVM (Linear)	Kernel function: linear Box constraint level:1 Kernel scale: auto	94.14	0.05	0.94	0.93	0.94
KNN (Coarse)	No. of neighbours: 100 Distance metric: Euclidean Distance weight: equal	94.52	0.04	0.95	0.93	0.94

3.2. Sistema de detección por medio de N-gramas enmascarados

El sistema de detección de dominios maliciosos presentado en [15], emplea técnicas de aprendizaje automático supervisado, a través de la extracción de características léxicas y estadísticas obtenidas del nombre de dominio. Para esto se usa un enfoque basado en caracterizar las ocurrencias de los N-gramas de los nombres de dominio, los cuales son secuencias contiguas de N elementos de una muestra de texto dada. En dicho documento se utilizan 44 características para entrenar modelos de aprendizaje, de las cuales, 15 son determinadas en función de la metodología presentada en [49].

En [49] se determinó que, usar la ocurrencia de todas las posibles combinaciones de los N-gramas para caracterizar el comportamiento de un conjunto de datos, desencadena en un aumento exponencial de las características necesarias a administrar. Ya que este enfoque es inviable para alfabetos extensos, se planteó un enfoque sintetizado denominado como N-gramas enmascarados, en el cual, el alfabeto de símbolos se reduce únicamente a 4 caracteres distintos. Emplear N-gramas enmascarados con-

siste en sustituir cada caracter de la muestra dada por un caracter que representa su tipo, en función de la Tabla 3.3.

Tabla 3.3: Enmascaramiento de caracteres.

Caracteres	**Máscara**
Constantes	c
Vocales	v
Dígitos numéricos	n
Símbolos restantes	s

La reducción del alfabeto a 4 caracteres reduce también el número de posibles combinaciones de N-gramas a 4^N. Aunque esta reducción de combinaciones conlleva a la pérdida de información, esto no afecta de manera notable en la precisión del sistema. Además, en [49] se utilizan 18 características adicionales tanto léxicas como estadísticas. En sus experimentos emplean N-gramas para valores de N = 1, 2, 3 y 4, para posteriormente determinar un *ranking* de las 15 características más influyentes en la detección de nombres de dominio maliciosos, por medio del método de selección presentado en [50].

Estas 15 características (F1-F15) han sido empleadas por [15] en conjunto con 29 características adicionales de su contribución, las cuales, se basan en la frecuencia de ocurrencia de caracteres. El *set* completo de características se muestra en la Tabla 3.4.

Tabla 3.4: Características léxicas y estadísticas.

Característica	Nombre
F1	Número de caracteres
F2	Número diferente de caracteres
F3	Número de consonantes
F4	Número de vocales
F5	Media 1-grama
F6	Varianza 1-grama
F7	Desviación estándar 1-grama
F8	Desviación estándar 2-gramas
F9	Ocurrencias de secuencias cc
F10	Ocurrencias de secuencias cv
F11	Ocurrencias de secuencias vc
F12	Ocurrencias de secuencias ccc
F13	Ocurrencias de secuencias cvc
F14	Ocurrencias de secuencias vcc
F15	Ocurrencias de secuencias vcc
F16	Razón entre las letras más frecuentes y la cantidad de caracteres
F17	Razón entre las letras menos frecuentes y la cantidad de caracteres
F18 - F30	Ocurrencias de los 2-gramas más frecuentes
F31 - F44	Ocurrencias de los 3-gramas más frecuentes

Es importante indicar que, para obtener las 29 características basadas en la frecuencia de ocurrencia de caracteres (F16 - F44), es necesario disponer de los conjuntos de caracteres más y menos utilizados en los nombres de dominio, específicamente para poder calcular las características (F16 - F17). Así mismo, para el cálculo de las características (F18 - F30) y (F31 - F44) se emplean los conjuntos de 2-gramas y 3-gramas más frecuentes, respectivamente. Estos conjuntos, se detallan en la Tabla 3.5.

Tabla 3.5: Conjunto de N-gramas relevantes en el cálculo de características.

Descripción	N-gramas
1-grama más frecuentes	a, e, i, o, s, r, n, t
1-grama menos frecuentes	f, k, w, v, x, j, z, q
2-gramas más frecuentes	in, er, an, re, es, ar, on, or, te, al, st, ne, en
3-gramas más frecuentes	ing, ion, ine, ter, lin, ent, the, ers, and, est, tio, tra, tor, art

Para la elaboración del clasificador, se optó por emplear modelos de aprendizaje pertenecientes a la categoría de *ensemble learning*, como C5.0, *Gradient Boosting Machine* (GBM), *Random Forests* y *Classification and Regression Trees* (CART); junto

con modelos de aprendizaje tradicionales como SVM y KNN. El uso de estos dos enfoques se debe a la búsqueda de un modelo de aprendizaje que sea capaz de clasificar los nombres de dominio con una alta tasa de precisión a partir de las 44 características previamente mencionadas. En la etapa de entrenamiento, cabe mencionar que se ha utilizado la técnica de validación cruzada con 10 *folds* para cada modelo de aprendizaje con el objetivo de lograr una generalización eficiente en la clasificación. En los experimentos relatados se determinó que todos los modelos de *ensemble learning* superaron a los modelos de aprendizaje tradicionales. Finalmente, se determinó que el hecho de incluir las 29 características adicionales por parte de [15] permitió que su enfoque de N-gramas enmascarados supere significativamente al enfoque tradicional presentado en [49].

Para entrenar los modelos de aprendizaje se generó una base de datos equilibrada de 50600 muestras en total. 25300 de las muestras corresponden a mAGDs distribuidas en 19 familias diferentes de DGAs obtenidas de [51], mientras que las 25300 muestras restantes corresponden a dominios legítimos obtenidos de [52]. La base de datos completa puede ser encontrada en [53]. Finalmente, los resultados obtenidos por cada modelo de aprendizaje se ilustran en la Tabla 3.6.

Tabla 3.6: Resultados obtenidos en la evaluación de modelos.

Classifier Name	Accuracy	Sensitivity
C5.0	0.97	0.97
gbm	0.96	0.97
RandomForest	0.96	0.97
CART	0.96	0.96
SVMRadial	0.96	0.98
KNN	0.95	0.95

3.3. ***BotNet Detection Framework*** **(BNDF)**

BNDF es un *framework* de detección de *hosts* infectados por Botnets presentado en [4], siendo capaz de operar en una red corporativa. Este *framework* captura y gestiona eventos de tráfico DNS en tiempo real, generando *fingerprints* que en conjunto con técnicas de aprendizaje automático permiten la detección de anomalías. Su arquitectura se presenta en la Figura 3.3, constando de 5 módulos secuenciales.

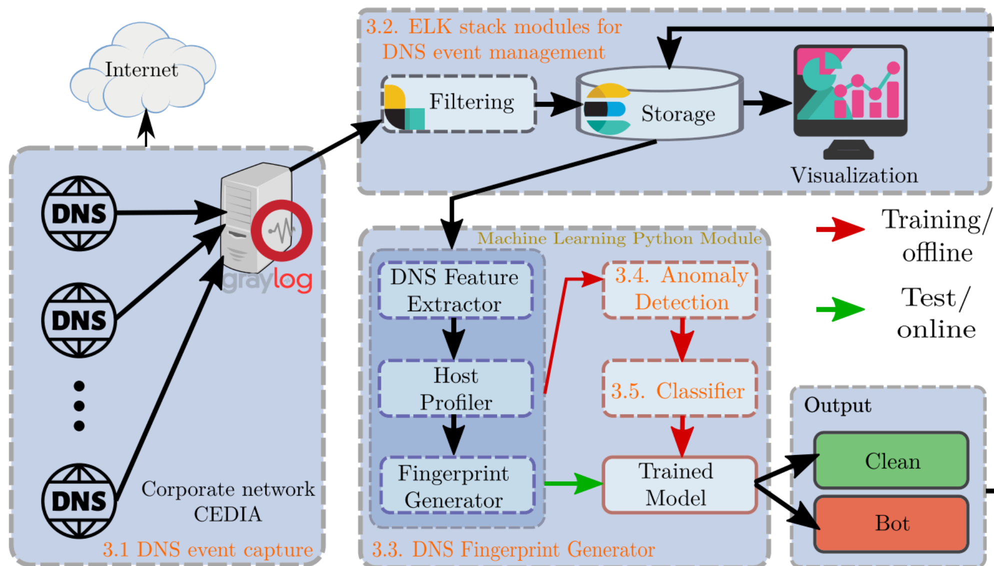


Figura 3.3: Arquitectura de BNDF desde la perspectiva de los procesos [4].

1. **Captura de eventos DNS:** Este módulo es capaz de capturar los registros de varios servidores DNS a través de un único servidor de registro *Graylog*, el cual, permite gestionarlos de manera centralizada. El servidor *Graylog* se encargará a su vez de reenviar los registros DNS hacia el puerto de entrada del siguiente módulo en formato *Graylog Extended Log Format* (GELF).

2. **Pila de módulos ELK para administración de eventos DNS:** Módulo encargado de administrar los eventos recibidos por el servidor *Graylog*, con el fin de adecuar los datos para procesarlos posteriormente. Está conformado por 4 acopladores de datos: *Logstash*, *Elasticsearch*, *Kibana* y *Python*.

   - **Logstash:** Es un procesador de datos que se utiliza para recopilar, filtrar y enrutar datos (generalmente se usa con en conjunto con *Elasticsearch*). Se encargará de extraer los *Top-Level Domain*s (TLDs) y Second-Level Domains (SLDs) de cada registro DNS.

   - **Elasticsearch:** Es un motor de búsqueda que recibe los datos enviados por *Logstash*, y se utiliza para el almacenamiento, consulta y análisis eficiente y rápido de extensos volúmenes de datos. Juega un papel fundamental en la indexación y búsqueda de los datos relacionados con los eventos DNS.

   - **Kibana:** Es una plataforma de visualización y análisis de datos que se utili-

za para presentar los datos almacenados en *Elasticsearch* de manera gráfica. Permite crear gráficos, tablas y paneles interactivos para analizar los eventos DNS de manera intuitiva y efectiva.

- **Python:** es un lenguaje de programación potente con un enfoque efectivo hacia la programación orientada a objetos. En este contexto, *Python* se utiliza para implementar funcionalidades adicionales, personalización y lógica específica requerida en el procesamiento y análisis de los datos DNS.

3. **Generador de *fingerprints* DNS:** Módulo conformado a su vez por 3 instancias diferentes. En primer lugar el “Extractor de Características” de DNS, extrae 13 atributos útiles que permitirán caracterizar el comportamiento de un *host*. A continuación, el “Perfilador de *Hosts*” analiza estos atributos y crea perfiles DNS específicos para cada *host*. Finalmente, el “Generador de *Fingerprints*” crea una huella digital única, la cual, contiene información estadística de cada *host* para determinar si su comportamiento es benigno o malicioso. Estas *fingerprints* son generadas para cada *host* considerando intervalos de una hora durante 10 días. La Tabla 3.7 detalla los 13 atributos considerados para la generación de las *fingerprints*.

Tabla 3.7: Características extraídas para la generación de fingerprints.

Tipo	Característica	Nombre
Basados en solicitudes DNS	P1	Número de solicitudes DNS por hora
	P2	Número de solicitudes DNS distintas por hora
	P3	Mayor número de solicitudes para un solo dominio
	P4	Número promedio de solicitudes por minuto
	P5	Número más alto de solicitudes por minuto
	P6	Número de consultas de registros MX
	P7	Número de consultas de registros PTR
	P8	Número de servidores DNS diferentes consultados
Basados en dominio	P9	Número de TLD diferentes consultados
	P10	Número de SLD diferentes consultados
	P11	Razón de Unicidad (relación entre el número de solicitudes enviadas y el número de solicitudes diferentes enviadas)
Basado en respuestas	P12	Número de consultas fallidas/NXDOMAIN
Basado en mapeo	P13	Razón de Flujo (proporción de las diferentes solicitudes enviadas a las diferentes direcciones IP resueltas)

4. **Detección de anomalías:** Módulo encargado de determinar la presencia de anomalías en la red. Para esto, se toma como premisa el hecho de que el tráfico benigno circundante será mayor que el tráfico malicioso. Dado que en cada red los umbrales de detección de anomalías son diferentes, este módulo ha sido

implementado en base a aprendizaje automático no supervisado, catalogando a cada huella dactilar como Bot o *Clean*, en función de la presencia o ausencia de anomalías, respectivamente.

Para la detección de anomalías, este módulo emplea el algoritmo de aprendizaje automático conocido como *Isolation Forest* (Bosques de Aislamiento) a través de la biblioteca *scikit-learn*, el cual, es ideal para trabajar con conjuntos de muestras altas.

Adicionalmente, sobre las *fingerprints* catalogadas como anómalas se emplea el algoritmo RMA para determinar si las consultas realizadas son hacia dominios generados por DGAs. Esto se basa en el hecho de que los Bots generan una gran cantidad de consultas hacia dominios DGAs. En [4] se menciona específicamente que un trabajo a futuro sería mejorar este algoritmo, siendo este presente documento el resultado de dicho proceso.

5. **Clasificador:** Este módulo tiene como objetivo utilizar un algoritmo de aprendizaje supervisado para detectar futuros *hosts* infectados de manera precisa y en el menor tiempo posible. Para este propósito, es necesario entrenar el algoritmo con el conjunto de datos proporcionado por el detector de anomalías, los cuales han sido etiquetados como Bot o *Clean*. Específicamente, se utiliza el algoritmo *Random Forests*, siendo sus principales conceptos los detallados en la Sección 2.3.3.

Es importante indicar que el *framework* BNDF, primero debe operar de manera *offline* en estado estable, donde se requiere de 10 días para analizar todo el tráfico DNS y generar el respectivo modelo entrenado. Posteriormente, una vez completada esta etapa, el sistema puede ejecutarse de manera *online*, en este caso, las *fingerprints* calculadas son enviadas al modelo entrenado, el cual las clasifica como Bot o *Clean*.

Diseño e implementación

En este capítulo se presenta la metodología utilizada durante la implementación de los algoritmos de detección estudiados, con el fin de ser aplicados en la red de una IES a través del diseño de un módulo de detección temprana en BNDF. En primer lugar, se describen los recursos y herramientas utilizados en la etapa de entrenamiento y evaluación de los modelos de aprendizaje. Para ello, se genera una base de datos equilibrada que incluye 50 familias de mAGDs y dominios legítimos, en la misma proporción. A continuación, se detalla la configuración de los modelos de aprendizaje a través de sus respectivos hiperparámetros. Por último, se seleccionan las métricas de evaluación que se utilizarán para medir el rendimiento de los modelos (Sección 4.1). Luego, en la Sección 4.2 se proporciona un detalle exhaustivo de la implementación de cada uno de los módulos que componen los sistemas de detección de mAGDs propuestos. Por otro lado, es importante conocer el entorno y los recursos necesarios para garantizar la conectividad y el correcto funcionamiento de las contribuciones propuestas al ser aplicadas en la red real de trabajo (Sección 4.3). Finalmente, en la Sección 4.4 se propone una nueva arquitectura para BNDF, la cual incorpora las contribuciones mencionadas anteriormente, con el objetivo de dotarlo de una capacidad de respuesta temprana ante la posible presencia de Bots en la red. Todas las implementaciones realizadas pueden ser encontradas en el repositorio de GitHub `https://github.com/fabianastudillo/dga-detector/`.

4.1. Recursos y herramientas

4.1.1. Base de datos

Como se ha mencionado anteriormente en el Capítulo 3, una metodología estándar y comúnmente utilizada para la detección y eliminación de las Botnets basadas en DGA, es el uso de técnicas de aprendizaje automático. Las soluciones más difundidas en este campo emplean la variante supervisada, explicada en la Sección 2.3. El aprendizaje automático supervisado consta de 2 fases: entrenamiento y prueba, siendo la fase de entrenamiento una etapa crítica, en donde, el modelo generado adquiere la capacidad de aprender los patrones de interés. Debido a esto, contar con una base de

datos de entrenamiento diversa es un paso primordial para cualquier implementación que emplee técnicas de aprendizaje automático supervisado.

Durante el desarrollo de este documento y en concordancia con lo mencionado en [32] y [54], se ha observado un hecho innegable en este campo: la falta de solidaridad y rigurosidad científica en relación con la reproducibilidad de resultados. En la literatura estudiada, ha sido común encontrar soluciones prometedoras pero en su mayoría irreproducibles debido a que las implementaciones propuestas así como los recursos empleados se detallan, pero permanecen disponibles solo para sus autores. [32] afirma que lo antes mencionado implica una falta de comparabilidad entre soluciones. Si bien es posible implementar las soluciones propuestas de manera similar en base a sus descripciones, la falta de un conjunto de entrenamiento estándar dificulta comparar los resultados entre diferentes propuestas. En su defecto, [32] y [54] han propuesto el conjunto de datos denominado como *University of Murcia domain generation algorithm dataset* (UMUDGA) que consta de más de 30 millones de muestras de mAGDs abarcando 50 clases distintas de DGAs junto con 1 millón de muestras benignas debidamente etiquetadas para ser usadas en sistemas basados en aprendizaje automático. Por cada una de las clases de DGAs disponibles, existen al menos 10000 muestras de nombres de dominio, pero aun así, la mayoría de clases poseen un millón de muestras. En la Tabla 4.1 se indican las 50 clases distintas de DGAs disponibles en UMUDGA.

Tabla 4.1: Clases de DGAs disponibles en UMUDGA.

Familia	Ejemplo	Familia	Ejemplo	Familia	Ejemplo
ccleaner	ab1fe3cfc138a.com	chinad	1cfci3lpp2g03jym.ru	proslikefan	ynhjpfre.se
corebot	f3x1n345petw4wp30uhu.ddns.net	cryptolocker	xiddynlbhwwyx.co.uk	pykspa	iyfjsswvrzz.info
dircrypt	yhimkmqdlwioramad.com	dyre	c00043ea763912a7d39e0930ecd10c31e3.to	pykspa_noise	odtvfpacyic.info
locky	fliffmdhwrdjb.org	fobber_v1	vhkintjtksyxgjrzz.net	qakbot	evtlfpcywscjnnrpaylajx.info
necurs	lpyufsvhmypoycie.sc	fobber_v2	drohppbkxj.com	ramdo	usyeeusiuaaayscg.org
pizd	advanceprepare.net	gozi_gpl	readilyafrsourceallowed.ru	ramnit	lyhggcqelikfigrbnxo.com
pushdo	wediljufobil.kz	gozi_luther	tandeclarandodiclinton.com	ranbyus_v1	hatyixfnvkrljy.cc
qadars	dq3whin4lm78.top	gozi_nasa	andhetithewateramission.com	ranbyus_v2	vhobmdkwchobjitcn.me
rovnix	migrationispeopleofof.biz	gozi_rfc4343	labeldetailsintroduction.com	sisron	mtcwndi1otaa.com
shiotob	9g2rdi9uga.net	kraken_v1	ibbwnhgh.mooo.com	suppobox_1	winterpaint.net
simda	puwedyp.info	kraken_v2	iuhqhbmq.yi.org	suppobox_2	storythank.net
symmi	amkovexoroas.ddns.net	matsnu	wedding-muscle.com	suppobox_3	constanceboniface.net
tinba	xuconvvoonqr.us	murofet_v1	xtwscsoqwxobqdcuibtorby.ru	tempedreve	mdcpkpelwb.com
zeus-newgoz	wqfny6pifxk413eiykco7bfd9.org	murofet_v2	krnpyunosomuzom.biz	vawtrak_v1	nglktid.top
alureon	meojytusps.com	murofet_v3	ozezh54gqpvaypvfrlzftmxo51bxnzexas.ru	vawtrak_v2	isernemd.com
banjori	oioberionirkutskagl.com	nymaim	spenthomeless.ad	vawtrak_v3	goraddefong.com
bedep	twaktrgbfvpss3.com	padcrypt	fefekbbcnddedfbk.com		

Este conjunto de datos proporciona una base sólida y diversa para la evaluación y comparación de diferentes soluciones de detección basadas en aprendizaje automático supervisado. La disponibilidad de UMUDGA promueve la reproducibilidad y la

comparabilidad de los resultados en el campo de la detección de Botnets basadas en DGA. En apoyo a la generación de soluciones innovadoras, en UMUDGA además de la gran lista de nombres de dominio, también se han proporcionado 130 características adicionales por cada muestra. Estás características han sido seleccionadas por ser ampliamente usadas en estudios de aprendizaje automático, como por ejemplo: el índice de *Jaccard*, la distancia *Euclidiana*, la distancia de *Manhattan*, la entropía, etc.

Debido a las grandes contribuciones que se han venido mencionando, la presente propuesta ha empleado netamente el conjunto de datos UMUDGA para la implementación de los algoritmos de detección de Botnets seleccionados. Para esto se han seleccionado 5000 muestras aleatorias por cada clase de DGA, escogiendo en la misma proporción un número de muestras benignas, formando así un conjunto de muestras equilibrado. En la Tabla 4.2 se detalla la selección realizada.

Tabla 4.2: Cantidad de muestras seleccionadas en la base de datos.

<table>
<tr><th>Tipo de nombre de dominio</th><th colspan="2">Cantidad</th></tr>
<tr><td>Malicioso</td><td colspan="2">250000</td></tr>
<tr><td>Benigno</td><td colspan="2">250000</td></tr>
<tr><td rowspan="2">Total:</td><td rowspan="2">500000</td><td>Entrenamiento: 90 %</td></tr>
<tr><td>Prueba: 10 %</td></tr>
</table>

De la Tabla 4.2 se puede notar que se ha seleccionado una relación 9:1 entre los datos dirigidos para el entrenamiento y los datos dirigidos para la evaluación de los algoritmos, respectivamente. Se ha optado por reforzar las fases de entrenamiento debido a que los algoritmos seleccionados trabajarán en conjunto con el *framework* BNDF, por lo cual, la prueba de validación de interés es el análisis del rendimiento de los mismos dentro de la red de un IES.

4.1.2. Selección de algoritmos de aprendizaje automático

Por todo lo antes mencionado en la Sección 2.3, no cabe duda que los modelos descritos presentan cualidades deseables para tratar tareas de clasificación y ventajas altamente equiparables entre sí. Es por esto entonces que se ha optado por explorar el rendimiento de los modelos de aprendizaje: KNN, SVM, *Random Forest* y MLP, con

el fin de determinar cual de estos puede ajustarse mejor al conjunto de entrenamiento y logra generalizar las clasificaciones de manera más efectiva.

Para este propósito, se han implementado los 4 modelos de aprendizaje mediante el lenguaje de programación *Python*, ya que, además de ser el lenguaje empleado para el desarrollo del *framework* BNDF, presenta características que a criterio de los presentes autores son consideradas altamente deseables. *Python* se destaca por contener una gran variedad de conjuntos de datos, así como, bibliotecas de herramientas de interés científico. En este contexto la biblioteca de libre distribución orientada al aprendizaje automático, *Scikit-learn*, ha sido empleada en las presentes implementaciones, debido a sus prestaciones a la hora de trabajar con extensas cantidades de datos, ofreciendo un rápido procesamiento, cálculo y visualización de resultados. Específicamente, la librería *Scikit-learn* [55] incluye los algoritmos de aprendizaje automático comúnmente más utilizados y proporciona documentación completa sobre cada uno de ellos. La implementación de dichos modelos puede ser encontrada en `https://github.com/fabianastudillo/dga-detector/blob/main/Modelos%20de%20Aprendizaje/MethodsML.py`.

4.1.2.1. Algoritmo *K-Nearest Neighbors* (KNN)

Sckit-learn [56] tiene un modelo de aprendizaje automático basado en el algoritmo KNN, el cual, puede ser llamado por medio del método “`KNeighborsClassifier()`”. Los conceptos involucrados para ajustar el algoritmo fueron mostrados en la Sección 2.3.1; en la Tabla 4.3 se indican los parámetros que se han seleccionado para el ajuste de los sistemas de detección.

Tabla 4.3: Hiperparámetros relevantes a configurar en el algoritmo KNN.

Hiperparámetro	Nombre	Descripción
`n_neighbors`	Número de vecinos	Número de vecinos más cercanos que se considerarán al realizar una predicción.
`p`	Potencia de Minkowski	Parámetro de potencia para la métrica de Minkowski.
`weights`	Función de peso	Función de peso utilizada en la predicción. Valores posibles: “uniform”, “distance” o definida por el usuario.

4.1.2.2. Algoritmo *Support Vector Machine* (SVM)

Sckit-learn [57] tiene un modelo de aprendizaje automático basado en el algoritmo SVM, el cual, puede ser llamado por medio del método: “`svm.SVC()`”. Los conceptos

involucrados para ajustar el algoritmo fueron mostrados en la Sección 2.3.2; en la Tabla 4.4 se indican los parámetros que se han seleccionado para el ajuste de los sistemas de detección.

Tabla 4.4: Hiperparámetros relevantes a configurar en el algoritmo SVM.

Hiperparámetro	Nombre	Descripción
`C`	Parámetro de regularización	Constante positiva de regularización, indica la penalización de los errores, la fuerza de la regularización es inversamente proporcional a `C`.
`kernel`	Función de kernel	Especifica el tipo de kernel que se usará en el algoritmo. Puede tomar los valores de: “linear”, “poly”, “rbf”, “sigmoid”, “precomputed” o definida por el usuario.
`gamma`	Coeficiente de kernel	Es el coeficiente del kernel cuando toma los valores de “rbf”, “poly” o “sigmoid”.

4.1.2.3. Algoritmo *Random Forests*

Sckit-learn [58] tiene un modelo de aprendizaje automático basado en el algoritmo *Random Forests*, el cual, puede ser llamado por medio del método “`RandomForestClassifier()`”. Los conceptos involucrados para ajustar el algoritmo fueron mostrados en la Sección 2.3.3; en la Tabla 4.5 se indican los parámetros que se han seleccionado para el ajuste de los sistemas de detección.

Tabla 4.5: Hiperparámetros relevantes a configurar en el algoritmo *Random Forest*.

Hiperparámetro	Nombre	Descripción
`n_estimators`	Número de estimadores	Número de árboles de decisión que se utilizarán en el modelo.
`max_features`	Máximo número de características	El número de características a considerar al buscar la mejor división. Puede tomar los valores de: “auto”, “sqrt”, “log2”, enteros o decimales.
`max_depth`	Máxima profundidad	Parámetro que especifica la profundidad máxima de cada árbol de decisión.

4.1.2.4. Algoritmo *Multi-Layer Perceptron* (MLP)

Sckit-learn [59] tiene un modelo de aprendizaje automático basado en el algoritmo MLP, el cual, puede ser llamado por medio del método “`MLPClassifier()`”. Los conceptos involucrados para ajustar el algoritmo fueron mostrados en la Sección 2.3.4; en la Tabla 4.6 se indican los parámetros que se han seleccionado para el ajuste de los sistemas de detección.

Tabla 4.6: Hiperparámetros relevantes a configurar en el algoritmo MLP.

Hiperparámetro	Nombre	Descripción
`activation`	Función de activación	Función de activación que se utiliza para la capa oculta. Valores posibles: “relu”, “tanh”, “logistic” o “identity”.
`hidden_layer_sizes`	Tamaño de capas ocultas	Número de capas ocultas y el número de neuronas en cada capa oculta.
`solver`	Solucionador	Especifica el algoritmo de optimización utilizado para ajustar los pesos de la red neuronal durante el entrenamiento. Valores posibles: “lbfgs”, “sgd” o “adam”.

4.1.2.5. Ajuste de hiperparámetros

La elección de los hiperparámetros es una parte crucial en la implementación de cualquier modelo de aprendizaje automático, ya que influye directamente en su rendimiento. Para facilitar esta tarea, *Sckit-learn* implementa distintas herramientas para el ajuste de los hiperparámetros, como por ejemplo "`GridSearchCV`" [60], que permite encontrar los mejores hiperparámetros para un modelo mediante una búsqueda exhaustiva en un conjunto predefinido de valores. Cabe mencionar que debido a que la búsqueda se realiza sobre un conjunto de valores fijos, el ajuste de los hiperparámetros representa una solución pseudo-óptima.

El funcionamiento de `GridSearchCV` implica definir un modelo de aprendizaje específico junto con un diccionario que contiene los diferentes valores posibles para cada hiperparámetro a ajustar. Posteriormente, realiza una evaluación exhaustiva de todas las combinaciones posibles, ajustando el modelo para cada una de ellas y determinando cuál proporciona el mejor rendimiento. Esta evaluación se realiza mediante validación cruzada, lo que ayuda a evitar el sobreajuste y proporciona una estimación más robusta del rendimiento del modelo en diferentes conjuntos de datos. Aunque el ajuste de hiperparámetros con `GridSearchCV` puede ofrecer una solución pseudo-óptima debido a la limitación del conjunto de valores fijos, sigue siendo una herramienta valiosa para mejorar el desempeño de los modelos de aprendizaje automático. En la Tabla 4.7 se indican los parámetros de entrada necesarios para el funcionamiento de `GridSearchCV`.

Tabla 4.7: Parámetros de entrada de *GridSearchCV*

Parámetro	Nombre	Descripción
estimator	Estimador	Modelo de aprendizaje automático
param_grid	Cuadrícula de hiperparámetros	Diccionario con posibles valores para cada hiperparámetro a ajustar
cv	Validación cruzada	Cantidad de folds para la validación cruzada

4.1.3. Selección de métricas de evaluación

Para evaluar el rendimiento de los modelos de aprendizaje se emplearán las siguientes métricas, las cuales, son usadas en la mayoría de implementaciones de aprendizaje automático orientadas a la detección de Botnets.

- **Matriz de confusión:** Es una tabla que resume el rendimiento de un modelo

de clasificación al comparar las predicciones con los valores reales. En el campo del aprendizaje automático enfocado en la detección de Bots, una matriz de confusión puede mostrar cuántos casos fueron correctamente o incorrectamente clasificados por el modelo. A partir de la matriz de confusión, se pueden calcular otras métricas que evalúan el rendimiento del modelo, como *Accuracy*, *Precision*, *Recall* y *F1-score*. Es una tabla de dos dimensiones que muestra la cantidad de predicciones realizadas por el modelo para cada clase y cómo se comparan con las clases reales, tal como se muestra en la Tabla 4.8.

Tabla 4.8: Matriz de confusión genérica.

Clase Real	**Clase Predicha**	
	Maliciosos	**Benignos**
Maliciosos	True Positive (TP)	False Negative (FN)
Benignos	False Positive (FP)	True Negative (TN)

Donde:

- **Verdaderos positivos (*True Positives*, TP):** Es el número de nombres de dominio maliciosos clasificados correctamente como maliciosos.
- **Falsos positivos (*False Positives*, FP):** Es el número de nombres de dominios benignos clasificados incorrectamente como maliciosos (falsas alarmas).
- **Verdaderos negativos (*True Negatives*, TN):** Es el número de nombres de dominios benignos clasificados correctamente como benignos.
- **Falsos negativos (*False Negatives*, FN):** Es el número de nombres de dominios maliciosos clasificados incorrectamente como benignos (casos omitidos).

- ***Accuracy* (Exactitud):** Es una medida general de qué tan bien el modelo clasifica correctamente los casos positivos y negativos. Se calcula dividiendo el número de predicciones correctas por el número total de predicciones según la Ecuación (4.1).

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{4.1}$$

- ***Precision*** **(Precisión):** Es una medida de qué tan preciso es el modelo al predecir los casos positivos. Se calcula dividiendo el número de verdaderos positivos entre la suma de verdaderos positivos y falsos positivos. Indica la proporción de casos positivos que el modelo clasifica correctamente según la Ecuación (4.2).

$$Precision = \frac{TP}{TP + FP} \tag{4.2}$$

- ***Recall*** **(Recuperación o Sensibilidad):** Es una medida de qué tan bien el modelo es capaz de encontrar todos los casos positivos. Se calcula dividiendo el número de verdaderos positivos entre la suma de verdaderos positivos y falsos negativos. Proporciona información sobre la capacidad del modelo para identificar correctamente los casos positivos según la Ecuación (4.3).

$$Recall = \frac{TP}{TP + FN} \tag{4.3}$$

- ***F1-score*****:** Es una métrica que combina la *Precision* y el *Recall* en una única medida que proporciona una evaluación más completa del rendimiento del modelo. Se calcula como la media armónica de ambos parámetros. Es útil cuando se desea tener un equilibrio entre la *Precision* y el *Recall*, especialmente en casos donde hay un desequilibrio en las clases o cuando se busca minimizar tanto los falsos positivos como los falsos negativos según la Ecuación (4.4).

$$F1 - Score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{4.4}$$

- **Validación cruzada:** La generalización de un modelo de aprendizaje automático indica la capacidad de un modelo para entregar clasificaciones precisas en cualquier escenario. En este contexto, el conjunto de datos de entrenamiento juega un papel muy importante. Si el modelo se entrena con un conjunto de datos estático, surge la posibilidad de obtener un rendimiento sobrestimado, en el cual, incluso las métricas de evaluación pueden dar resultados mejores de los que en realidad el modelo entrega para casos prácticos. Por este motivo, la técnica de “Validación Cruzada” es atractiva, ya que permite a las métricas de evaluación

entregar resultados más confiables.

Esta técnica consiste en dividir los datos en múltiples K subconjuntos llamados *folds* o pliegues, aproximadamente del mismo tamaño. Posteriormente, se selecciona uno de los pliegues como conjunto de prueba y los $K-1$ pliegues restantes como conjunto de entrenamiento. Entonces, se entrena el modelo utilizando el conjunto respectivo y se evalúa su rendimiento utilizando el conjunto de prueba. Este proceso se repite K veces, utilizando un pliegue diferente para la prueba en cada iteración y los demás para el entrenamiento. Al final, se promedian los resultados obtenidos en cada iteración para obtener una medida más robusta del rendimiento del modelo. Ya que se utilizan todos los datos disponibles tanto para el entrenamiento como para la evaluación, la validación cruzada permite una evaluación más precisa del modelo.

Emplear la técnica de la validación cruzada es una práctica recomendada en la detección de Botnets para obtener estimaciones más confiables del rendimiento del modelo y evitar problemas de sobreajuste o subajuste al evaluar su capacidad de generalización.

4.2. Implementación de algoritmos de detección de mAGDs

A continuación se describe la metodología empleada para la implementación de los algoritmos mostrados en las Secciones 3.1 y 3.2.

4.2.1. Algoritmo *MaldomDetector*

Las bases teóricas acerca del funcionamiento del algoritmo *MaldomDetector* fueron presentadas en la Sección 3.1, acorde al estudio realizado en [3]. La implementación del algoritmo sigue la arquitectura mostrada en la Figura 3.1 y se estructura en los siguientes módulos secuenciales:

1. **Módulo de preparación de datos:** Este módulo consta de 3 etapas: *Data Preprocessing*, *Feature Extraction Unit*, y *Randomness Measurement Algorithm* (RMA). La etapa *Data Preprocessing* se encarga de extraer los nombres de dominio so-

licitados por los *hosts* de la red a partir de los paquetes de solicitud DNS. Sin embargo, ya que el algoritmo *MaldomDetector* será implementado dentro del *framework* BNDF, esta etapa se omite, debido a que este último la implementa por sí mismo.

En cuanto a las etapas *Feature Extraction Unit* y RMA, se ha optado por implementarlas de manera conjunta, generando así una sola etapa capaz de obtener tanto las características básicas como las derivadas, es decir, el vector V, de manera directa. La implementación en *Python* de esta etapa conjunta se ha realizado a través de la función `DataPreparation` presentada en `https://github.com/fabianastudillo/dga-detector/blob/main/MaldomDetector/DataPreparation.py`. Esta función permite obtener el vector V con las 12 características determinadas mediante el análisis de correlación de los componentes principales desarrollado en [3], por lo cual, las características F9, F11, F13 y F14 no son calculadas. La lista de características seleccionadas es mostrada en la Tabla 3.1.

Cabe mencionar que para las características F8 y F15 existen casos en donde se pueden generar indeterminaciones, específicamente, cuando el nombre de un dominio no contiene vocales. Por defecto el algoritmo *MaldomDetector* no contempla está situación, por lo cual, para abordar este problema se ha definido en base a múltiples experimentaciones que las características F6 y F3 (responsables de las indeterminaciones), en lugar de contener un valor de 0, contengan un valor de 0.1. La elección de este valor ha demostrado ser efectiva al mejorar significativamente las métricas de evaluación, particularmente las métricas F8 y F15, que han aumentado en un factor de 10. Esto ha permitido resaltar la ausencia de vocales en los dominios, una característica mayoritariamente presente en los dominios mAGDs.

Para automatizar la obtención de los vectores de características V para cada nombre de dominio de la base de datos, se ha generado el *script* `AutomationDP`, adjunto en `https://github.com/fabianastudillo/dga-detector/blob/main/MaldomDetector/AutomationDP.py`. Este *script* genera como salida el archivo “`FeaturesDatabaseMaldom.xlsx`”, el cual, deberá ser modificado agregando una

columna adicional, denominada como "Etiqueta", de tal manera de que cada nombre de dominio benigno posea un valor de 0, y cada dominio malicioso, un valor de 1.

2. **Módulo de toma de decisiones:** Este módulo implementa los modelos de aprendizaje mencionados en la Sección 4.1.2, generando clasificadores de nombres de dominio. En la etapa de entrenamiento se utiliza el archivo "`FeaturesDatabaseMaldom.xlsx`", dividiendo los datos aleatoriamente en un 90 % para entrenamiento y un 10 % para pruebas, esto acorde a la Tabla 4.2. Así mismo, se ha seleccionado un valor de $K = 10$, para efectuar el proceso de validación cruzada.

   Para ajustar los hiperparámetros mencionados en las tablas de la Sección 4.1.2 para cada modelo de aprendizaje, se utilizó la herramienta mencionada en la Sección 4.1.2.5. A través del script presentado en `https://github.com/fabianastudillo/dga-detector/blob/main/Modelos%20de%20Aprendizaje/GridSearch.py` se ha logrado optimizar los modelos de aprendizaje con los valores de las Tablas 4.9, 4.10, 4.11, 4.12.

Tabla 4.9: Ajuste de los hiperparámetros en el algoritmo KNN (*MaldomDetector*).

Hiperparámetro	**Nombre**	**Valor**
`n_neighbors`	Número de vecinos	100
`p`	Potencia de Minkowski	10
`weights`	Función de peso	uniform

Tabla 4.10: Ajuste de los hiperparámetros en el algoritmo SVM (*MaldomDetector*).

Hiperparámetro	**Nombre**	**Valor**
`C`	Parámetro de regularización	300
`kernel`	Función de kernel	linear

Tabla 4.11: Ajuste de los hiperparámetros en el algoritmo *Random Forests* (*MaldomDetector*).

Hiperparámetro	**Nombre**	**Valor**
`n_estimators`	Número de estimadores	100
`max_features`	Máximo número de características	sqrt
`max_depth`	Máxima profundidad	100

Tabla 4.12: Ajuste de los hiperparámetros en el algoritmo MLP (*MaldomDetector*).

Hiperparámetro	**Nombre**	**Valor**
`activation`	Función de activación	relu
`hidden_layer_sizes`	Tamaño de capas ocultas	64
`solver`	Solucionador	adam

Los ajustes de los hiperparámetros para cada modelo de aprendizaje pueden ser encontrados en `https://github.com/fabianastudillo/dga-detector/blob/main/MaldomDetector/Maldom-Train.py`.

4.2.2. Algoritmo basado en N-gramas enmascarados

Las bases teóricas del funcionamiento de este algoritmo fueron presentadas en la Sección 3.2. En nuestra implementación, se ha optado por desarrollar los siguientes módulos secuenciales:

1. **Módulo de extracción de características:** Este módulo es el encargado de obtener las 44 características léxicas y estadísticas para cada nombre de dominio consultado en la red a través de solicitudes DNS. La implementación en *Python* de este módulo se desarrolló a través de la función `maskgrams` que puede ser encontrada en `https://github.com/fabianastudillo/dga-detector/blob/main/N-gramas%20enmascarados/maskgrams.py`. Este módulo permite obtener las características presentadas en la Tabla 3.4 a partir de un nombre de dominio. Para las características F5-F8, se utilizan las Ecuaciones (4.5), (4.6) y (4.7).

$$\mu_x = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{4.5}$$

$$\sigma_x^2 = \frac{\sum_{i=1}^{n}(x_i - \mu_x)^2}{n-1} \tag{4.6}$$

$$\sigma_x = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \mu_x)^2}{n-1}} \tag{4.7}$$

Donde:

- x_i: Es la frecuencia de ocurrencia de cada i-ésimo N-gramas distinto dentro del nombre de dominio.
- n: Es la cantidad de N-gramas distintos en el nombre de dominio.
- μ_x: Es la media de N-gramas.

- σ_x^2: Es la varianza de N-gramas.
- σ_x: Es la desviación estándar de N-gramas.

Para automatizar que la función *maskgrams* sea capaz de obtener las características de cada nombre de dominio de la base de datos, se ha generado el *script* presentado en `https://github.com/fabianastudillo/dga-detector/blob/main/N-gramas%20enmascarados/AutomationMG.py`. Este *script* genera como salida el archivo "`FeaturesDatabaseMG.xlsx`", el cual, deberá ser modificado agregando una columna adicional, denominada como "Etiqueta", de tal manera de que cada nombre de dominio benigno posea un valor de 0, y cada dominio malicioso, un valor de 1.

2. **Clasificador:** Este módulo implementa los modelos de aprendizaje mencionados en la Sección 4.1.2, para generar clasificadores de nombres de dominio. En la etapa de entrenamiento se utiliza el archivo "`FeaturesDatabaseMG.xlsx`", dividiéndolo aleatoriamente en un 90 % para entrenamiento y un 10 % para pruebas, esto acorde a la Tabla 4.2. Así mismo, se ha seleccionado un valor de $K = 10$, para efectuar el proceso de validación cruzada.

   Para ajustar los hiperparámetros mencionados en las tablas de la Sección 4.1.2 para cada modelo de aprendizaje, se utilizó la herramienta mencionada en la Sección 4.1.2.5. A través del script presentado en `https://github.com/fabianastudillo/dga-detector/blob/main/Modelos%20de%20Aprendizaje/GridSearch.py` se ha logrado optimizar los modelos de aprendizaje con los valores de las Tablas 4.13, 4.14, 4.15, 4.16.

Tabla 4.13: Ajuste de los hiperparámetros en el algoritmo KNN (N-gramas enmascarados).

Hiperparámetro	Nombre	Valor
`n_neighbors`	Número de vecinos	25
`p`	Potencia de Minkowski	1
`weights`	Función de peso	distance

Tabla 4.14: Ajuste de los hiperparámetros en el algoritmo SVM (N-gramas enmascarados).

Hiperparámetro	Nombre	Valor
`C`	Parámetro de regularización	300
`kernel`	Función de kernel	linear

Tabla 4.15: Ajuste de los hiperparámetros en el algoritmo *Random Forests* (N-gramas enmascarados).

Hiperparámetro	Nombre	Valor
n_estimators	Número de estimadores	100
max_features	Máximo número de características	sqrt
max_depth	Máxima profundidad	50

Tabla 4.16: Ajuste de los hiperparámetros en el algoritmo MLP (N-gramas enmascarados).

Hiperparámetro	Nombre	Valor
activation	Función de activación	logistic
hidden_layer_sizes	Tamaño de capas ocultas	64
solver	Solucionador	adam

Los ajustes de los hiperparámetros para cada modelo de aprendizaje pueden ser encontrados en https://github.com/fabianastudillo/dga-detector/blob/main/N-gramas%20enmascarados/MG-Train.py.

4.3. Escenario de trabajo

En primer lugar, se realizó la instalación de BNDF utilizando la infraestructura de red de la “Universidad de Cuenca”, siendo esta una Institución de Educación Superior (IES) del Ecuador. Esto se debe a las necesidades inherentes del alcance de la implementación, ya que, es necesario trabajar con una gran cantidad de *hosts* para entrenar el modelo de manera inicial e identificar anomalías en las solicitudes DNS.

Debido a lo expuesto anteriormente, se ha solicitado un servidor dedicado para la recepción de todas las solicitudes DNS generadas en la Universidad de Cuenca, con el fin de implementar BNDF en conjunto con las contribuciones del presente documento. Para tener acceso al servidor de manera remota, ha sido necesario configurar una *Virtual Private Network* (VPN) con credenciales de usuario y contraseña proporcionadas por la IES. Finalmente, una vez dentro de la red de la IES se accederá al servidor proporcionado usando SSH. En la Tabla 4.17 se detallan las principales características del servidor, así como los parámetros relacionados a la configuración de la VPN.

Tabla 4.17: Características del servidor proporcionado por la IES.

Parámetro	**Valor**
Sistema Operativo	Ubuntu 20.04.1 LTS
Almacenamiento	48GB
Memoria RAM	4GB
Dominio de la red de la universidad (VPN)	redufw.ucuenca.edu.ec
Puerto remoto del *gateway* (VPN)	20443
Dirección IP interna	10.0.x.x
Puerto de recepción de solicitudes DNS	10000

Para el acceso a la red se ha empleado el cliente VPN, "*FortiClient VPN*", el cual, permite establecer conexiones seguras y encriptadas a través de una red pública, protegiendo la comunicación entre el dispositivo y la red remota. En la Figura 4.1, se muestra la configuración del cliente VPN con los datos mostrados en la Tabla 4.17, mientras que en la Figura 4.2 se ilustra el establecimiento de la conexión.

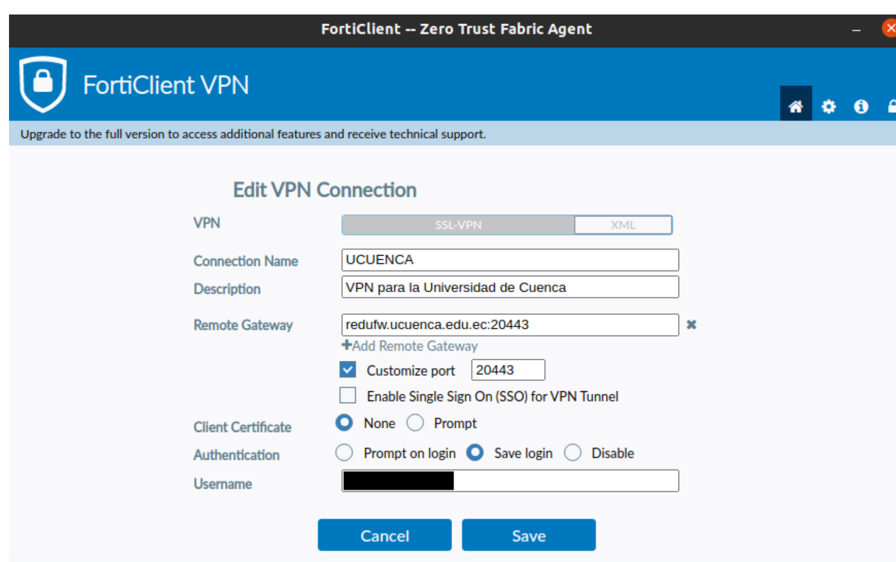


Figura 4.1: Configuración del cliente VPN.

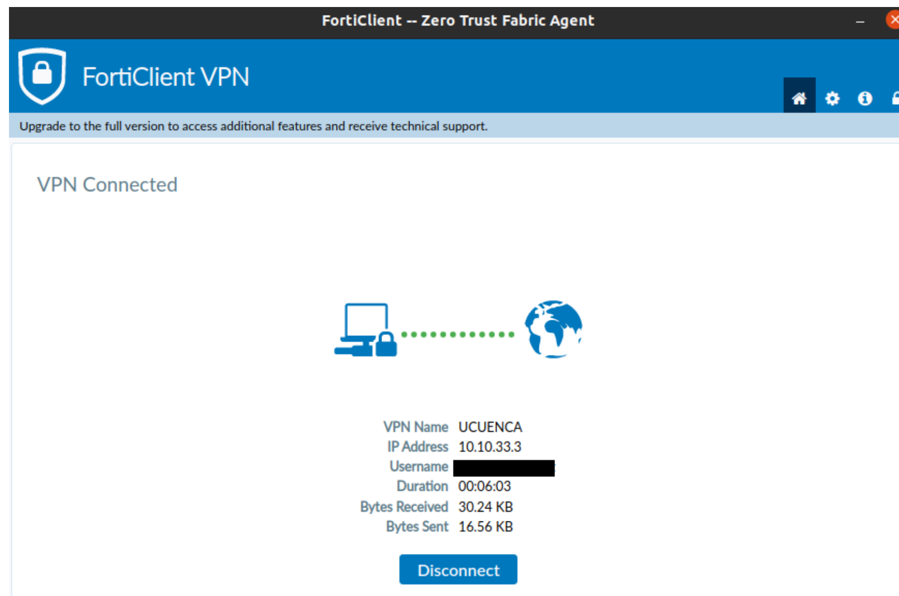


Figura 4.2: Conexión establecida a través de la VPN.

Una vez dentro de la red interna de la IES, se accede al servidor proporcionado mediante el protocolo SSH. Para este propósito, a través del sistema criptográfico RSA se han generado las llaves públicas de los ordenadores de los autores, para posteriormente utilizar las llaves privadas correspondientes como credenciales de acceso al servidor. El ingreso exitoso al servidor se muestra en la Figura 4.3.

```
adrian@adrian-VirtualBox:~$ sudo ssh           @          -i /home/adrian/.ssh/id_rsa
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-144-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Tue Jun 13 03:27:17 UTC 2023

  System load:  0.17               Users logged in:                1
  Usage of /:   31.6% of 47.92GB   IPv4 address for br-ea48c8cfe2ab: 172.18.0.1
  Memory usage: 84%                IPv4 address for docker0:       172.17.0.1
  Swap usage:   74%                IPv4 address for ens160:
  Processes:    265

  => There are 3 zombie processes.

 * Strictly confined Kubernetes makes edge and IoT secure. Learn how MicroK8s
   just raised the bar for easy, resilient and secure K8s cluster deployment.

   https://ubuntu.com/engage/secure-kubernetes-at-the-edge

Expanded Security Maintenance for Applications is not enabled.

74 updates can be applied immediately.
1 of these updates is a standard security update.
To see these additional updates run: apt list --upgradable

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status

New release '22.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


*** System restart required ***
Last login: Tue Jun 13 03:26:34 2023 from
          @server:~$
```

Figura 4.3: Acceso al servidor mediante el protocolo SSH.

4.4. Arquitectura propuesta

Dado que el presente documento es una extensión del trabajo presentado en [4], es necesario realizar la descarga, instalación y uso de todos los recursos empleados para su desarrollo. Esto es posible gracias a que el *framework* es de código abierto, siendo desarrollado en el sistema operativo *Ubuntu* 20.04.1 LTS, empleando mayoritariamente *Python*. Todos los recursos necesarios para su uso están disponibles en la plataforma de desarrollo colaborativo *GitHub* en el siguiente enlace: `https://github.com/fabianastudillo/bndf`.

4.4.1. Requerimientos previos

Para llevar a cabo la instalación de BNDF, es necesario contar con las dependencias que se mencionan a continuación. Los procedimientos de instalación de estas dependencias se detallan en la Sección A.1.1.

- *Docker 17.06.0:* Es una plataforma de contenedores de código abierto que permite empaquetar, distribuir y ejecutar aplicaciones dentro de contenedores. El proceso de instalación puede encontrarse en la Sección A.1.1.1.
- *Docker-Compose 1.27.0:* Es una herramienta que permite definir y administrar aplicaciones multi-contenedor utilizando archivos de configuración YAML. Proporciona una forma sencilla de definir la estructura de una aplicación compuesta por múltiples servicios, sus dependencias y la configuración necesaria para ejecutarlos. El proceso de instalación puede encontrarse en la Sección A.1.1.2.
- *Git:* Es un sistema de control de versiones distribuido utilizado para gestionar el desarrollo de proyectos de *software*, facilitando el trabajo colaborativo y el seguimiento de cambios en el código fuente. El proceso de instalación puede encontrarse en la Sección A.1.1.3.
- *Curl:* Es una herramienta de línea de comandos utilizada para transferir datos a través de varios protocolos de red, como *Hypertext Transfer Protocol* (HTTP), *Hypertext Transfer Protocol Secure* (HTTPS), entre otros. El proceso de instalación puede encontrarse en la Sección A.1.1.4.
- *Time:* Es herramienta que se utiliza para obtener información sobre el tiempo transcurrido durante la ejecución de un programa, incluyendo el tiempo de CPU utilizado y otros detalles relacionados con el rendimiento. El proceso de instalación puede encontrarse en la Sección A.1.1.5.

Cabe mencionar que la máquina sobre la cual se ejecutará BNDF debe estar habilitada para recibir el reenvió de todas las solicitudes DNS de la red de análisis, específicamente en su puerto 10000.

4.4.2. Instalación de BNDF

El proceso de instalación de BNDF en el servidor mencionado, se describe en la Sección A.1.2. En caso de realizar los procedimientos de manera correcta, se podrá observar como los *dockers*: “*logstash*”, “*elasticsearch*”, “*kibana*” y “*python*”, se inicializan, como se muestra en la Figura 4.4.

```
@server:~$ docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED        STATUS                  PORTS                                                                              NAMES
9b1584338645   python-uc              "/bin/sh -c 'cron &&…"   2 months ago   Up 2 months                                                                                                python
0dc8ff97be8a   kibana:7.13.1          "/bin/tini -- /usr/l…"   2 months ago   Up 2 months (healthy)   0.0.0.0:5601->5601/tcp, :::5601->5601/tcp                                          kibana
b28c76507af2   logstash-uc:7.13.1     "/usr/local/bin/dock…"   2 months ago   Up 2 months (healthy)   5044/tcp, 9600/tcp, 10000/tcp, 0.0.0.0:10000->10000/udp, :::10000->10000/udp   logstash
fe1fdeed5629   elasticsearch:7.13.1   "/bin/tini -- /usr/l…"   2 months ago   Up 2 months (healthy)   0.0.0.0:9200->9200/tcp, :::9200->9200/tcp, 9300/tcp                                elasticsearch
@server:~$
```

Figura 4.4: *Dockers* inicializados exitosamente.

4.4.3. Contribuciones

Al inicializar el *framework* BNDF, los 5 módulos presentados en la Figura 3.3 comienzan a trabajar simultáneamente. El funcionamiento del sistema consiste inicialmente en capturar los registros DNS de toda la red, en un servidor de registro *Graylog*. Posteriormente, estos son reenviados hacia la etapa de filtrado para su adecuación y posterior almacenamiento. Es entonces cuando comienza el proceso de generación de las *fingerprints*, en base a la extracción de las 13 características mencionadas en la Tabla 3.7. A partir de este punto, el sistema cuenta con 2 modos de operación: *offline* y *online*.

- ***Offline:*** El sistema se mantiene en estado estable en este modo durante los primeros 10 días de ejecución. En este periodo de tiempo se utilizan las *fingerprints* generadas para detectar anomalías en las solicitudes DNS mediante un modelo de aprendizaje automático no supervisado de árboles de aislamiento. Este modelo clasifica cada *fingerprint* como Bot o *Clean*, con el objetivo de generar una base de datos etiquetada. Finalmente, se procede a entrenar un modelo de aprendizaje supervisado de bosques aleatorios utilizando esta base de datos.

- ***Online:*** Este modo es ejecutado una vez se finalice el periodo de los 10 días. A partir de este momento, se emplea el modelo entrenado de bosques aleatorios junto con las nuevas *fingerprints* calculadas para su clasificación correspondiente. Como proceso adicional, se realiza un análisis de nombres de dominio por medio del algoritmo RMA, con el objetivo de determinar si cada nombre de dominio consultado pertenece a un mAGD o a un dominio legítimo.

BNDF presenta una limitación en su diseño en cuanto a la generación de resultados en tiempo real. Aunque el análisis de la red se realiza de manera inmediata, la interpretación de los eventos DNS ocurre después de cada hora. A partir de estos últimos,

se obtienen las *fingerprints* involucradas durante el proceso de clasificación de un *host* como Bot. Por lo tanto, el tiempo necesario para la clasificación está compuesto por el período de una hora requerido para la obtención de los eventos DNS, así como el tiempo de procesamiento de las *fingerprints*, y de la clasificación como tal. En conclusión, el diseño del *framework* no cuenta con un módulo de detección temprana de Bots, lo cual supone un riesgo importante para la red debido a la naturaleza efímera de la comunicación de CC de las Botnets.

Debido a que la limitación expuesta está en contraposición a los objetivos propuestos en este trabajo, se ha optado por generar un módulo de detección temprana, independiente de los módulos de generación de *fingerprints*, detección de anomalías y clasificación de *hosts*.

La arquitectura propuesta se ilustra en la Figura 4.5, en donde, a diferencia de la arquitectura original (Figura 3.3), se ha incluido un módulo de detección temprana que consta de los dos algoritmos de detección de mAGDs estudiados (3.1 y 3.2), así como del algoritmo RMA, con el objetivo de comparar sus rendimientos. Si bien los algoritmos propuestos han sido entrenados con una base de datos de 50 familias de DGAs distintas (Tabla 4.1), en un escenario real como el de una red de una IES, las DGAs presentes pueden pertenecer únicamente a un subconjunto de familias, o incluso a categorías no antes vistas. Debido a esta naturaleza heterogénea de las redes, los modelos de aprendizaje deben ser lo más generalizados posibles. Sin embargo, al no estar especializados en una familia específica, es posible que aumente la tasa de falsos positivos detectados. Además, es fundamental tener en cuenta que el conjunto de nombres de dominio legítimo utilizado para entrenar los algoritmos puede no representar con precisión la recurrencia y las características de todos los dominios benignos presentes en las redes reales. Por estos motivos, ha sido necesario emplear una *whitelist* que contenga los nombres de dominio legítimos más frecuentes en la red de análisis. Esta inclusión permite reducir la tasa de falsos positivos y evitar el análisis y tratamiento repetitivo de los nombres de dominio presentes en ella.

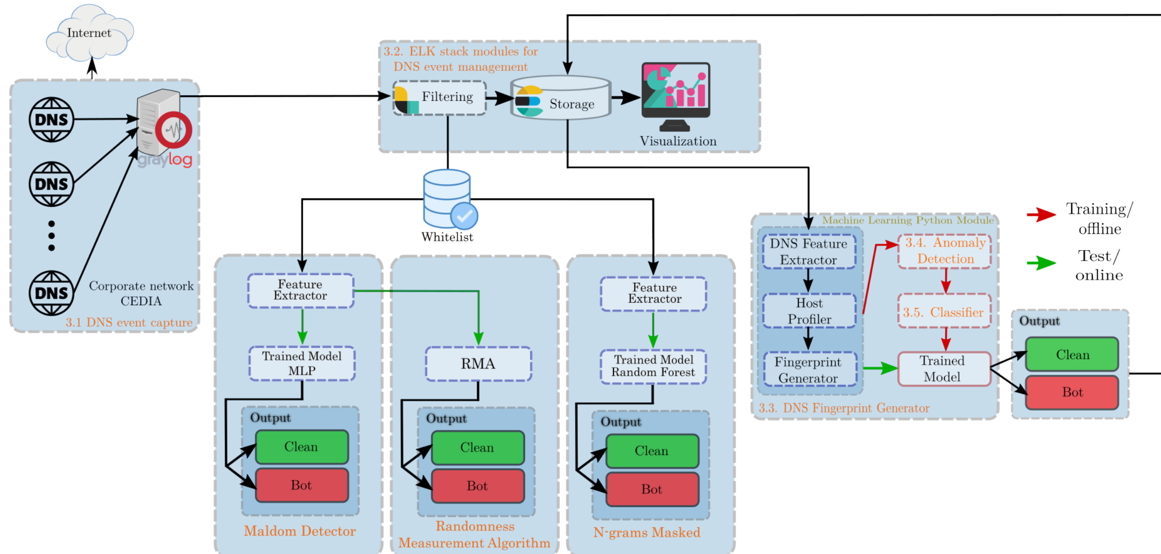


Figura 4.5: Arquitectura propuesta desde la perspectiva de los procesos.

El modulo de detección temprana ha sido desarrollado a través del *script* presentado en el repositorio de GitHub `https://github.com/fabianastudillo/dga-detector/blob/main/Detector.py`. Para su funcionamiento se utiliza un *socket* configurado en modo pasivo a través del puerto 9999. A través de dicho puerto se reciben solicitudes DNS previamente filtradas, en base a los campos `_ip`, `_name` y `_rescode`. Cuando un nombre de dominio es catalogado como un mAGD por cualquiera de los 3 algoritmos de detección, se registra la IP solicitante, así como el código de respuesta (`_rescode`) de la solicitud. Este último es de gran importancia debido a que la gran mayoría de Bots amparados bajo el DGA realizan una gran cantidad de solicitudes fallidas antes de establecer una comunicación exitosa con el servidor *Command and Control* (CC), por lo cual, este parámetro es de interés para una detección precisa de los Bots.

Resultados

En este capítulo se plantean y detallan distintos escenarios de evaluación para los algoritmos de detección estudiados (*MaldomDetector* y N-gramas enmascarados) en conjunto con RMA. El objetivo general es evaluar el rendimiento de la clasificación y la eficiencia en el uso de los recursos computacionales. Para llevar a cabo estas evaluaciones, se emplean diversas métricas, como *Accuracy*, *Precision*, *Recall* y *F1-score*, así como el tiempo de procesamiento por dominio y el almacenamiento ocupado por los modelos de aprendizaje automático entrenados. Estas métricas permiten obtener una visión integral del desempeño de los algoritmos y facilitan la identificación de la mejor solución en los diferentes contextos planteados. En el primer escenario de evaluación (Sección 5.1), se realiza una evaluación individual de los diferentes algoritmos de aprendizaje automático utilizados en cada algoritmo de detección. El objetivo es determinar cuál de estos algoritmos se ajusta mejor al enfoque de extracción de características propuesto por los algoritmos de detección. Una vez seleccionados los algoritmos de aprendizaje automático con mejor desempeño, se pasa al siguiente escenario de evaluación (Sección 5.2), donde se comparan sus rendimientos junto con el algoritmo RMA. Para realizar esta comparación, se utiliza una base de datos de 50000 muestras equilibradas y aisladas. El objetivo de esta prueba es evaluar cómo se desempeñan los algoritmos en condiciones realistas y determinar cuál de ellos ofrece el mejor rendimiento en términos de clasificación de dominios. En la Sección 5.3 se plantea un escenario de evaluación que involucra a la red de la IES en funcionamiento. El objetivo principal consiste en inferir la capacidad de detección de mAGDs que cada algoritmo presenta al procesar solicitudes DNS reales en un período de siete días. Se analiza detalladamente la cantidad de dominios maliciosos detectados por cada algoritmo y se examina la relación existente entre ellos, identificando los mAGDs únicos y coincidentes. Además, se emplea el parámetro `_rescode` disponible en las solicitudes DNS para identificar el comportamiento de los mAGDs detectados por cada algoritmo. Finalmente, en la Sección 5.4 se aborda una evaluación enfocada en el uso de los recursos computacionales utilizados por los dos algoritmos de detección estudiados.

5.1. Evaluación de los algoritmos de aprendizaje automático

En esta sección, se evalúan los algoritmos de aprendizaje automático previamente estudiados y configurados para cada uno de los algoritmos de detección implementados. El objetivo principal es analizar y comparar las métricas de evaluación del rendimiento, como *Accuracy*, *Precision*, *Recall* y *F1-Score*, con el fin de identificar el modelo que presente el mejor desempeño en cada uno de los algoritmos.

Para este propósito, como se mencionó en la Sección 4.1.1, se utilizarán 500000 muestras de nombres de dominio distribuídas equilibradamente entre mAGDs y dominios legítimos. Se mantendrá una relación de 9:1 entre las muestras de entrenamiento y las muestras de evaluación, respectivamente. Además, siguiendo la práctica empleada en las implementaciones estudiadas (Sección 4.2.1 y Sección 4.2.2), se realizará una validación cruzada con $K = 10$ para evitar resultados sobreestimados.

5.1.1. Algoritmo MaldomDetector

Al emplear el enfoque de extracción de características planteado por algoritmo de detección *MaldomDetector*, en conjunto con los algoritmos de aprendizaje automático, KNN, SVM, *Random Forests* y MLP, se obtuvieron los resultados que se ilustran tanto en la Figura 5.1 como en la Tabla 5.1.

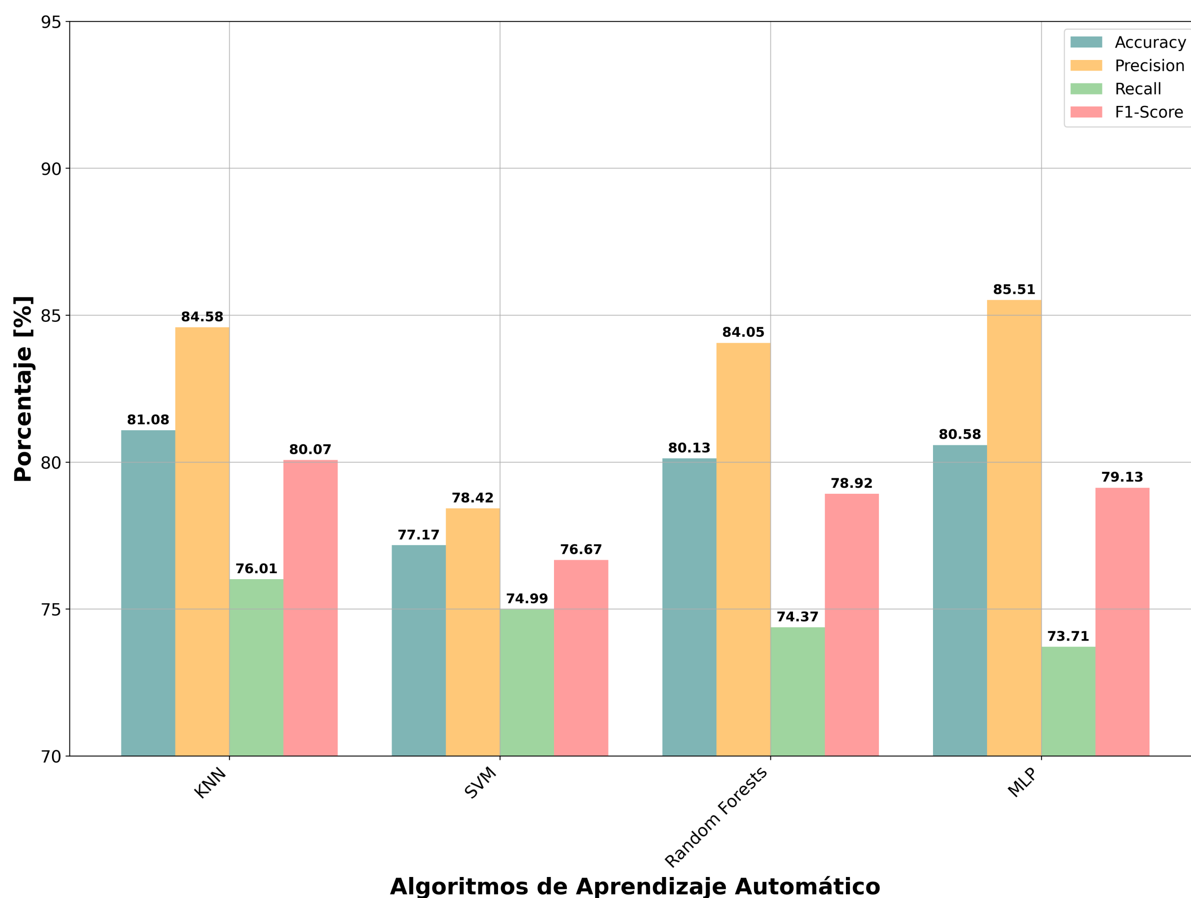


Figura 5.1: Evaluación del rendimiento de los algoritmos de aprendizaje automático bajo el enfoque de *MaldomDetector*.

Tabla 5.1: Evaluación del rendimiento de los algoritmos de aprendizaje automático bajo el enfoque de *MaldomDetector*.

Algoritmo de Aprendizaje	**Accuracy [%]**	**Precision [%]**	**Recall [%]**	**F1-Score [%]**
KNN	81.08	84.58	76.01	80.07
SVM	77.17	78.42	74.99	76.67
Random Forests	80.13	84.05	74.37	78.92
MLP	80.58	85.51	73.71	79.13

Como se observa en la Figura 5.1 y se confirma en la Tabla 5.1, los algoritmos KNN, *Random Forests* y MLP obtienen valores de *Accuracy* superiores al 80 %. Entre ellos, el algoritmo KNN se destaca con un valor del 81.08 %. Esto indica que los 3 algoritmos presentan una exactitud global aceptable, implicando que las predicciones correctas de los casos legítimos y maliciosos, son altas. En resumen, estos algoritmos han logrado una alta tasa de clasificación correcta en general.

Por otro lado al analizar la métrica *Precision*, nuevamente los algoritmos KNN, *Random Forests* y MLP obtienen valores similares, superando el 84 %. Específicamente, el algoritmo MLP obtuvo un rendimiento levemente superior con un valor de 85.51 %. Estos resultados indican que los 3 algoritmos logran generar una baja tasa de falsos positivos, lo cual significa que la gran mayoría de las predicciones de los dominios clasificados como maliciosos son correctas. Se concluye que estos algoritmos minimizan los casos en los que se clasifican incorrectamente dominios legítimos como maliciosos.

Al analizar la métrica *Recall*, se observa que los valores varían entre el 73.71 % y el 76.01 %, siendo la métrica más baja para cada algoritmo. El algoritmo KNN se posiciona con el mejor rendimiento, alcanzando el 76.01 %. Esto significa que dicho algoritmo tiene la capacidad más alta entre los algoritmos evaluados para identificar la mayoría de los dominios maliciosos y, por lo tanto, genera la tasa más baja de falsos negativos de los 4 algoritmos.

Finalmente, para la métrica *F1-Score*, los algoritmos KNN y MLP alcanzan rendimientos cercanos al 80 %. Sin embargo, nuevamente el algoritmo KNN se destaca al obtener el mejor resultado con un valor de 80.07 %. Este resultado indica que el algoritmo KNN logra la tasa más baja tanto de falsos positivos como de falsos negativos en comparación con los otros algoritmos evaluados.

En conclusión, el algoritmo KNN se destaca por tener el rendimiento más alto en 3 de las 4 métricas de evaluación (*Accuracy*, *Recall* y *F1-Score*). Por lo tanto, se lo ha seleccionado como el modelo de aprendizaje automático para ejecutar el algoritmo de detección *MaldomDetector* dentro de BNDF.

5.1.2. Algoritmo N-gramas enmascarados

Al emplear el enfoque de extracción de características planteado por algoritmo de detección de N-gramas enmascarados, en conjunto con los algoritmos de aprendizaje automático, KNN, SVM, *Random Forests* y MLP, se obtuvieron los resultados que se ilustran tanto en la Figura 5.2 como en la Tabla 5.2.

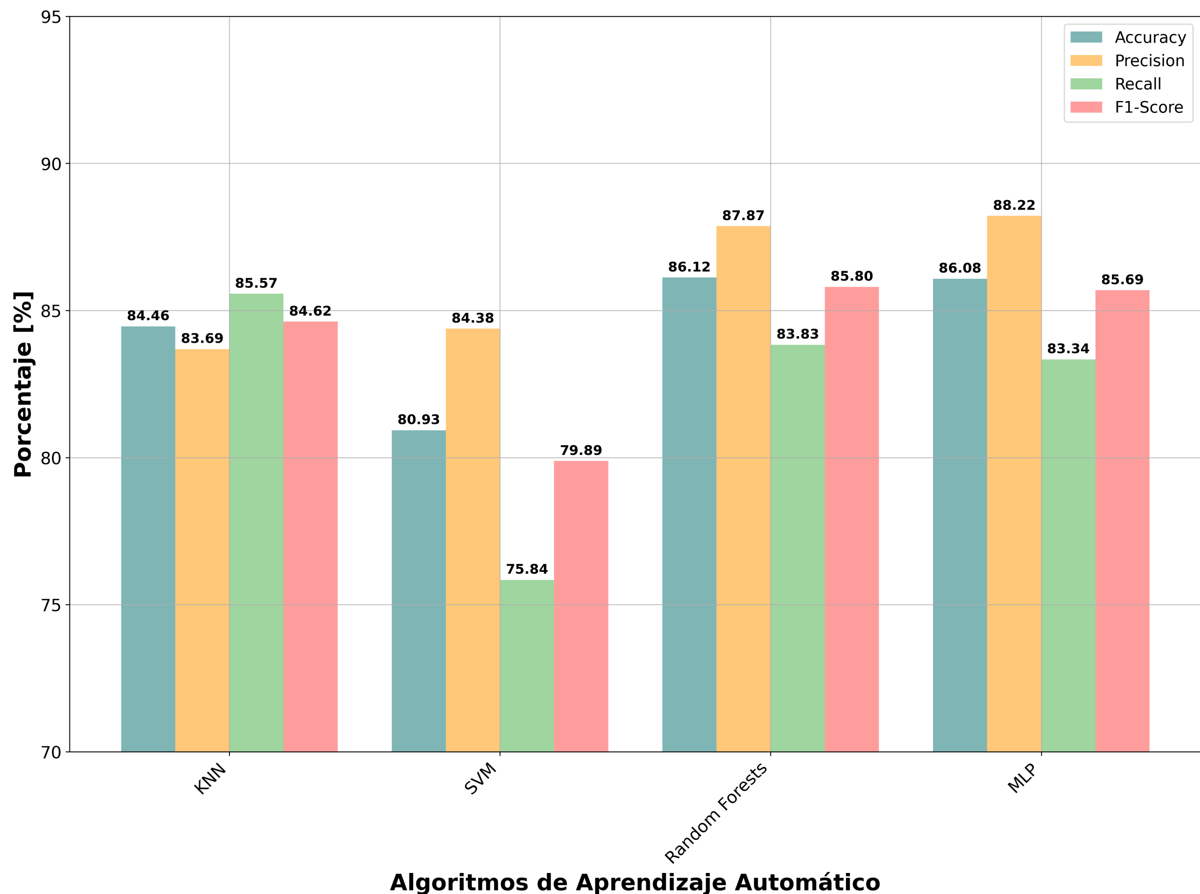


Figura 5.2: Evaluación del rendimiento de los algoritmos de aprendizaje automático bajo el enfoque de N-gramas enmascarados.

Tabla 5.2: Evaluación del rendimiento de los algoritmos de aprendizaje automático bajo el enfoque de N-gramas enmascarados.

Algoritmo de Aprendizaje	Accuracy [%]	Precision [%]	Recall [%]	F1-Score [%]
KNN	84.46	83.69	85.57	84.62
SVM	80.93	84.38	75.84	79.89
Random Forests	86.12	87.87	83.83	85.80
MLP	86.08	88.22	83.34	85.69

Según los resultados obtenidos en la Figura 5.2 y en la Tabla 5.2, en este caso, los algoritmos *Random Forests* y MLP obtienen valores de *Accuracy* superiores al 86 %. Entre ellos, el algoritmo *Random Forests* se destaca como el mejor con un valor del 86.12 %. Esto indica que los 4 algoritmos presentan una exactitud global notable, implicando que las predicciones correctas de los casos legítimos y maliciosos, son altas. En general, todos los algoritmos han logrado alcanzar una alta tasa de clasificación correcta.

Al analizar la métrica *Precision*, únicamente los algoritmos *Random Fortest* y MLP superan el 87 %. Aun así, el algoritmo MLP obtuvo el rendimiento superior, con un valor de 88.22 %. Estos resultados indican que ambos algoritmos logran generar una baja tasa de falsos positivos, implicando que la gran mayoría de las predicciones de los dominios clasificados como maliciosos son correctas. Se puede interpretar que estos algoritmos son los que tienen la mejor capacidad para minimizar los casos en los que se clasifican incorrectamente dominios legítimos como maliciosos.

Por otra parte, la métrica *Recall* presenta sus valores más altos para el algoritmo KNN y *Random Forests* con valores del 85.57 % y 83.83 %, respectivamente. Esto significa que ambos algoritmos tienen la capacidad más alta entre los algoritmos evaluados para identificar la mayoría de los dominios maliciosos y, por lo tanto, generan las tasas más bajas de falsos negativos de los 4 algoritmos.

Finalmente, para la métrica *F1-Score*, el algoritmo *Random Forests* obtiene el mejor rendimiento alcanzando el valor de 85.80 %. Este resultado indica que este algoritmo logra la tasa más baja tanto de falsos positivos como de falsos negativos en comparación con los otros algoritmos evaluados.

En conclusión, el algoritmo *Random Forests* se destaca por tener el rendimiento más alto en 2 de las 4 métricas de evaluación (*Accuracy* y *F1-Score*). Por lo tanto, se lo ha seleccionado como el modelo de aprendizaje automático para ejecutar el algoritmo de detección de N-gramas enmascarados dentro de BNDF.

5.2. Escenario de evaluación *offline*

Una vez se han seleccionado los modelos de aprendizaje automático que presentan el mejor rendimineto en conjunto con las características propuestas por los algoritmos de detección *MaldomDetector* y N-gramas enmascarados, se realiza la evaluación y comparación de los mismos con respecto a RMA. Para esto se utiliza una base de datos equilibrada de 50000 muestras que incluyen mAGDs y dominios legítimos nunca antes vistos en las etapas de entrenamiento de los modelos de aprendizaje. El objetivo de esta evaluación adicional, es identificar al algoritmo de detección que presente el mejor desempeño en base a las métricas de evaluación de *Accuracy*, *Precision*, *Recall*

y *F1-score*. En la Figura 5.3 y la Tabla 5.3 se presentan los resultados obtenidos.

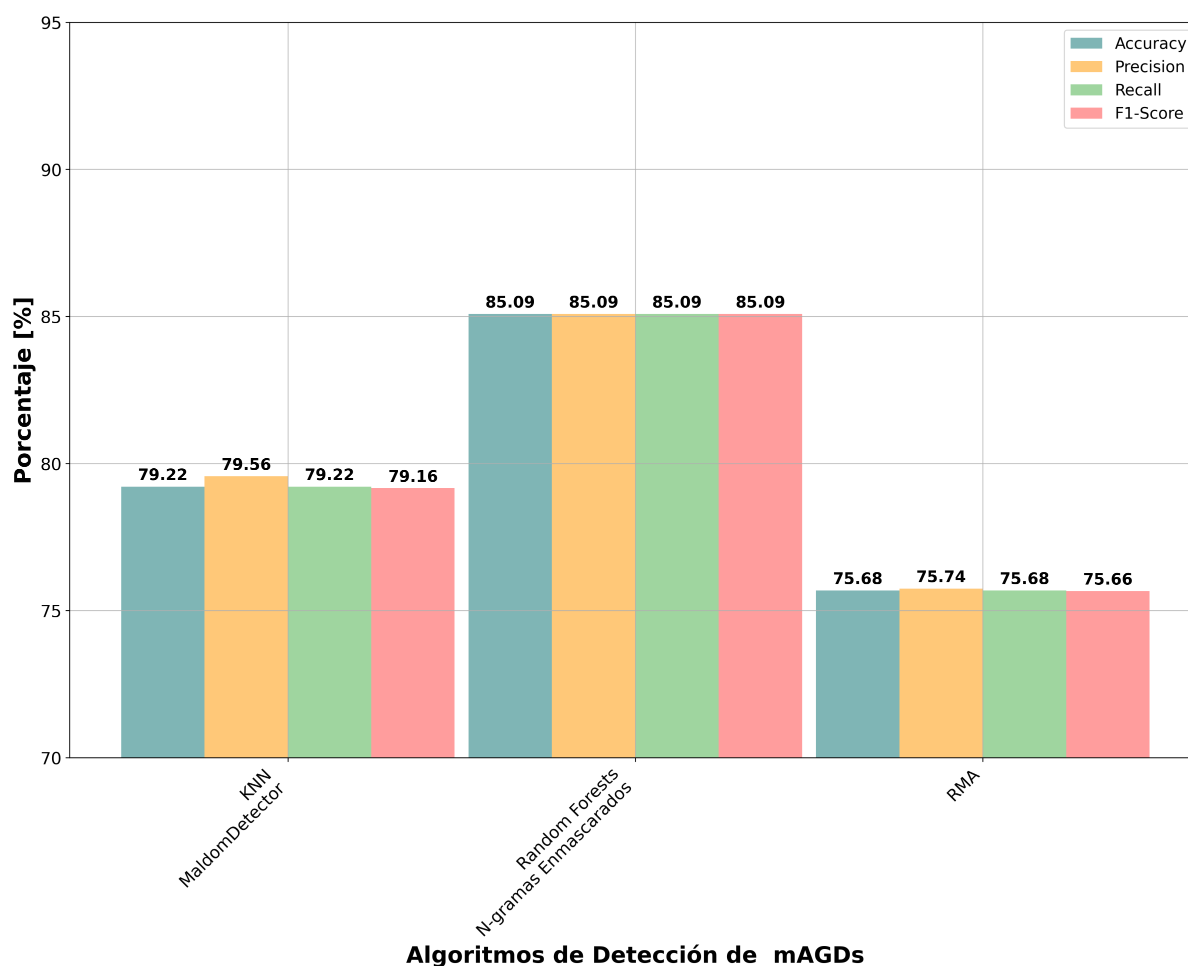


Figura 5.3: Evaluación del rendimiento de los algoritmos *MaldomDetector*, N-gramas enmascarados y RMA.

Tabla 5.3: Evaluación del rendimiento de los algoritmos *MaldomDetector* y N-gramas enmascarados y RMA.

Algoritmo de Detección	Accuracy [%]	Precision [%]	Recall [%]	F1-Score [%]
MaldomDetector	79.22	79.56	79.22	79.16
N-gramas Enmascarados	85.09	85.09	85.09	85.09
RMA	75.68	75.74	75.68	75.66

Como se observa en la Figura 5.3 y se confirma en la Tabla 5.3, el algoritmo de detección de N-gramas enmascarados supera en cada una de las métricas evaluadas, los resultados obtenidos por el algoritmo *MaldomDetector* y por RMA, con aproximadamente un 5 % y 10 %, respectivamente. Analizando el algoritmo de N-gramas enmascarado, se puede notar que este presenta un valor del 85.09 % en cada una

de las métricas de evaluación. Esto implicaría que el algoritmo de N-gramas enmascarados es capaz de clasificar una notable cantidad de nombres de dominio correctamente, además la gran mayoría de las predicciones de mAGDs realizadas han sido correctas, de igual manera posee una capacidad notable de detección hacia todos los mAGDs globalmente, y logra la tasa más baja conjunta de falsos positivos y falsos negativos.

En conclusión, el algoritmo de detección de N-gramas enmascarados presenta un rendimiento superior en todas las métricas de evaluación analizadas, posicionándose como la mejor opción en términos de clasificación para su implementación en redes reales.

5.3. Escenario de evaluación *online*

Debido a que en un escenario real no se tiene la certeza absoluta sobre la procedencia de todas las solicitudes DNS circundantes, es imposible contrastar de manera directa los resultados obtenidos en el escenario de evaluación *offline* (Sección 5.2) con el rendimiento de los algoritmos de detección en una red en funcionamiento. Aun así, es posible identificar resultados que indiquen una capacidad de detección más precisa entre algoritmos. En este contexto, se realizará un análisis de las solicitudes DNS junto con las predicciones obtenidas por los algoritmos de detección, con el fin de inferir si dichas predicciones son verdaderamente mAGDs.

Para esta evaluación ha sido necesario realizar el despliegue del módulo de detección temprana presentado en la Figura 4.5 en la red de la IES de interés. Las pruebas propuestas consisten principalmente en contabilizar y analizar los mAGDs detectados por cada algoritmo. Para este propósito, se ha definido un período de análisis de 7 días, dando inicio en el día martes 20 de junio a las 12:00pm y finalizando el día lunes 26 de junio a las 12:00am. Durante este periodo se ha realizado la captura de las solicitudes DNS de toda la red en formato GELF. Además, sobre cada una de estas solicitudes se ha ejecutado un proceso de filtrado para obtener los parámetros `_ip`, `_name` y `_rescode`.

El parámetro `_ip` idealmente permitiría identificar los *hosts* infectados responsables

de las consultas malintencionadas. Sin embargo, cabe mencionar que en el presente caso, esto no fue posible debido a que las solicitudes DNS capturadas se encontraban enmascaradas a través de 15 servidores DNS locales distribuidos en las distintas instalaciones de la IES. Por otro lado, el parámetro `_name` fue empleado como entrada por los algoritmos de detección, dado que contiene el FQDN al cual se busca acceder. El uso de este parámetro permite a los algoritmos de detección, clasificar el tráfico de la red entre benigno y malicioso. Finalmente, a través del parámetro `_rescode` fue posible realizar inferencias sobre la procedencia de la solicitudes DNS, siendo de utilidad para el análisis de la capacidad de detección de cada algoritmo. Este último parámetro puede adoptar los siguientes valores:

- **`NOERROR`:** Indica que la solicitud se completó correctamente y se encontró una respuesta. En el ámbito de la detección de Botnets, este `_rescode` puede presentarse en solicitudes DNS catalogadas erróneamente como tráfico malicioso (falsos positivos), o en el caso de que se haya establecido la comunicación con el servidor CC con éxito.
- **`NXDOMAIN`:** Indica que el nombre de dominio consultado no existe. La presencia de este valor en las consultas DNS puede atribuirse al intento de establecer recurrentemente la comunicación con el servidor CC. Sin embargo, también es posible identificarlas en casos accidentales debido a errores de tipeo por parte de los *hosts*, aunque en menor medida.
- **`SERVFAIL`:** Indica que el servidor DNS no pudo procesar la solicitud. Los motivos pueden incluir problemas de conectividad, configuración o sobrecargas del servidor. Este tipo de `_rescode` no es determinante en la detección de Botnets.
- **`REFUSED`:** Indica que el servidor DNS rechazó la solicitud. Al igual que el caso anterior, este valor no desempeña un papel importante en la detección de Botnets.

Cabe mencionar que existen solicitudes DNS en donde ocurre la ausencia de este parámetro. Esto indicaría que no se registró explícitamente un código de respuesta específico en la solicitud DNS. Este comportamiento es común en las consultas DNS responsables de iniciar el proceso de comunicación con el servidor DNS. Debido a lo antes expuesto, en los posteriores análisis se ha optado por no considerar las solicitu-

des DNS que no incluyan el parámetro `_rescode`. En la Figura 5.4 se detalla la cantidad de solicitudes DNS capturadas durante el periodo de análisis.

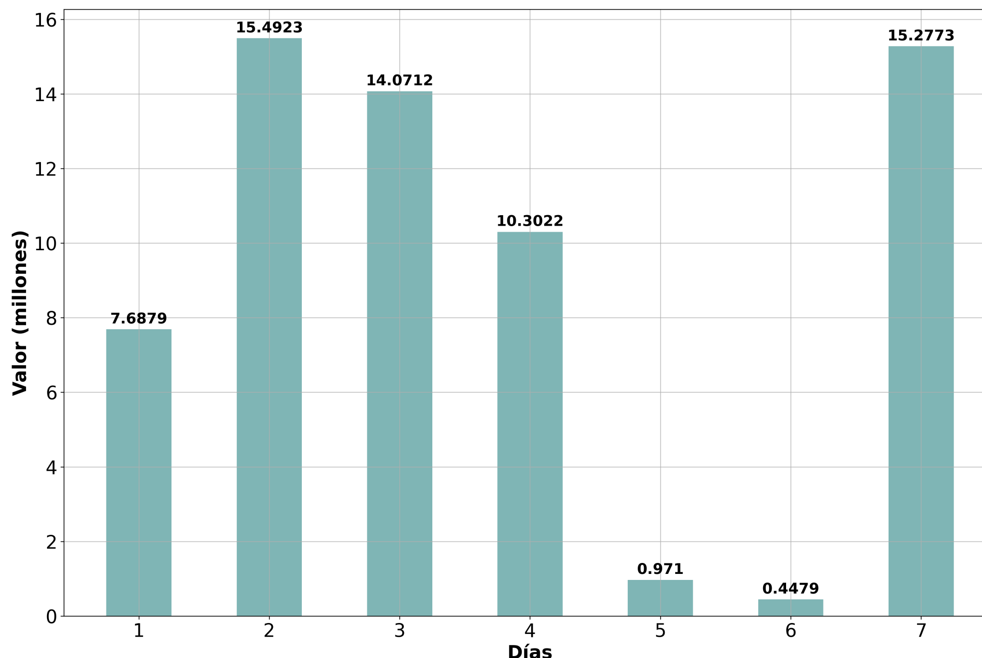
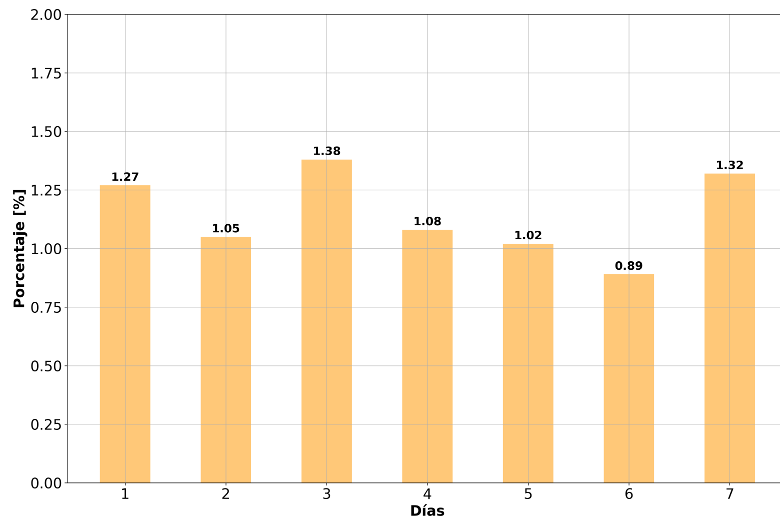
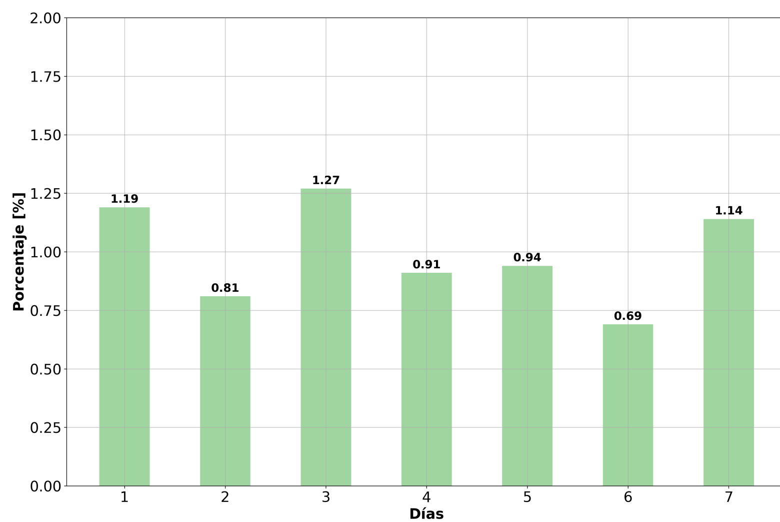


Figura 5.4: Cantidad de solicitudes DNS capturadas por día.

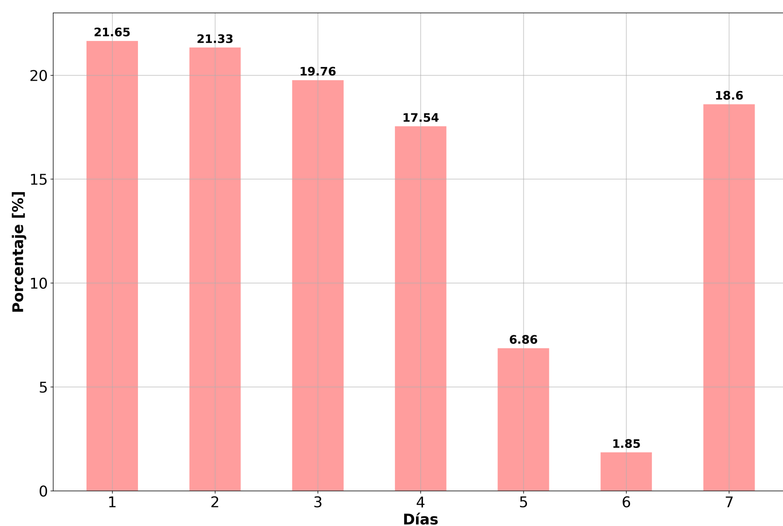
En la Figura 5.4, se puede observar un cambio drástico en la cantidad de solicitudes capturadas durante los días 5 y 6. Este cambio se justifica por la baja afluencia de *hosts* en la IES en ambos días. Durante todo el período analizado, se ejecutaron simultáneamente los algoritmos de detección *MaldomDetector*, N-gramas enmascarados y RMA. Por otro lado, la Figura 5.5 muestra el porcentaje de los mAGDs detectados por cada uno de los algoritmos en relación con las solicitudes DNS capturadas por día. Así mismo, la Tabla 5.4 detalla las cantidades absolutas de los mAGDs detectados por día por cada uno de los algoritmos.



(a) MaldomDetector.



(b) N-gramas Enmascarados.



(c) RMA.

Figura 5.5: Cantidad de nombres de dominio catalogados como mAGDs por cada algoritmo de detección.

Tabla 5.4: Número de mAGDs detectados por cada algoritmo.

Algoritmos de Detección	**Día 1**	**Día 2**	**Día 3**	**Día 4**	**Día 5**	**Día 6**	**Día 7**
MaldomDetector	97308	163252	194238	111644	9889	3997	201078
N-gramas enmascarados	91400	125326	179296	94160	9147	3082	174057
RMA	1664756	3304103	2780650	1806558	66570	8291	2841404
Solicitudes DNS Totales	7687915	15492323	14071196	10302240	971032	447915	15277323

Al observar la Figura 5.5 y la Tabla 5.4, se puede apreciar que tanto el algoritmo *MaldomDetector* como el de N-gramas enmascarados muestran una tasa de detección similar, a pesar de que utilizan enfoques de extracción de características diferentes. En la mayoría de los días analizados, ambos algoritmos han dictaminado que aproximadamente el 1 % del tráfico circundante en la red es generado por DGAs. Aun así, es necesario mencionar que el algoritmo de N-gramas enmascarados presenta consistentemente la menor tasa de detección de mAGDs. Por otro lado, el algoritmo RMA muestra una tasa de detección mucho más alta, indicando que una proporción mucho mayor del tráfico es malicioso en comparación con los algoritmos anteriores. RMA detecta que en los días con mayor afluencia de tráfico, más del 17 % de este es malicioso.

Partiendo del hecho de que en cualquier red de datos la mayoría del tráfico circundante es benigno, se puede inferir que RMA generaría la mayor cantidad de falsos positivos en comparación con los algoritmos *MaldomDetector* y N-gramas enmascarados. La inferencia anterior ha sido confirmada a través de una inspección visual realizada sobre un subconjunto de mAGDs determinados por cada algoritmo. La considerable variación en la cantidad de mAGDs detectados se puede atribuir al hecho de que RMA es un algoritmo determinista y carece de la capacidad de identificar patrones específicos presentes en cada familia DGA, a diferencia de los algoritmos *MaldomDetector* y N-gramas enmascarados, que adquieren esta capacidad mediante el uso de aprendizaje automático.

De manera complementaria para analizar el comportamiento global de cada algoritmo, en la Tabla 5.5 se presenta la cantidad total de mAGDs detectados y su porcentaje correspondiente, con respecto a la cantidad total de solicitudes DNS generadas en la red durante el periodo mencionado.

Tabla 5.5: mAGDs totales detectados por cada algoritmo.

Algoritmos de Detección	**Total de mAGDs Detectados**	**Porcentaje de Detección Total [%]**
MaldomDetector	781406	1.22
N-gramas enmascarados	676468	1.05
RMA	12472332	19.41
Solicitudes DNS Totales		
64249944		

A través de la Tabla 5.5, se confirma nuevamente que el RMA sería el algoritmo más propenso a generar falsos positivos, ya que cataloga aproximadamente el 20 % del tráfico circundante como malicioso. Por otra parte, los algoritmos *MaldomDetector* y de N-gramas enmascarados solo clasifican el 1 % como tráfico malicioso, aproximadamente.

Con el fin de analizar la relación existente entre los conjuntos de mAGDs detectados por cada uno de los algoritmos, es necesario identificar la cantidad de predicciones únicas y coincidentes entre estos. Esto cobra relevancia dado que si las predicciones obtenidas por el algoritmo *MaldomDetector* o por el de N-gramas enmascarados resultaran ser un subconjunto de las predicciones realizadas por el RMA, fuera posible plantear una nueva arquitectura en cascada. Esto con el objetivo de que los algoritmos basados en aprendizaje automático eviten el análisis de dominios innecesarios que no han sido detectados por RMA. La Figura 5.6 ilustra las predicciones únicas y coincidentes totales entre todos los algoritmos, proporcionando una representación visual de esta relación.

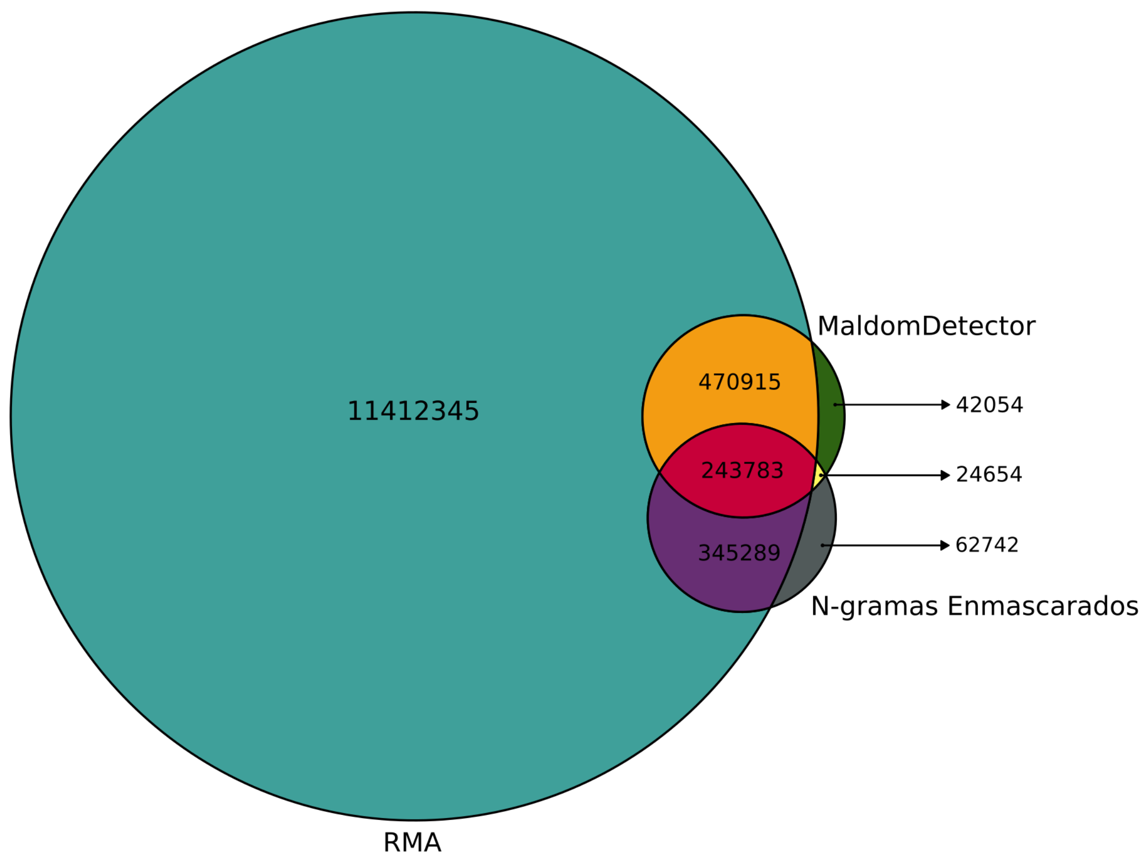


Figura 5.6: Cantidad de mAGDs únicos y coincidentes detectadas por cada algoritmo.

La Figura 5.6 muestra que la mayoría de las predicciones de mAGDs realizadas por los algoritmos MaldomDetector y de N-gramas enmascarados también han sido identificadas por el algoritmo RMA. Sin embargo, existen algunos mAGDs que solo son detectados por sus respectivos algoritmos, lo que descarta la implementación de la arquitectura en cascada mencionada anteriormente. Siempre y cuando los recursos computacionales lo permitan, la implementación de los algoritmos *MaldomDetector* y de N-gramas enmascarados permitiría obtener tasas de falsos positivos más bajas.

Dado que los algoritmos DGAs tienen un comportamiento extremadamente repetitivo y reconocible, se puede emplear el campo `_rescode` disponible en las solicitudes DNS como una prueba de validación sobre las mAGDs detectadas por cada algoritmo. Como se mencionó en la Sección 2.1.2, el comportamiento típico de un DGA implica

generar múltiples solicitudes DNS en busca de establecer la comunicación con el servidor de CC. Durante este proceso, el DGA consultará servidores DNS repetidamente utilizando en mayor medida dominios inexistentes. Por cada una de estas consultas, el servidor DNS responderá utilizando el parámetro `_rescode` con el valor `NXDOMAIN`, lo cual, es de suma importancia en el presente análisis. Este enfoque es válido únicamente antes de que el DGA logre establecer la comunicación con el servidor de CC, dado que al establecerse, el servidor DNS ajustará el parámetro `_rescode` con el valor `NOERROR`. Por lo tanto, se espera que la cantidad de solicitudes DNS catalogadas como maliciosas y que tengan el parámetro `_rescode` con el valor `NXDOMAIN` sea alta, mientras que las solicitudes con el valor `NOERROR` sean bajas.

En la Figura 5.7, se ilustra el porcentaje de solicitudes DNS por día que han sido catalogadas como maliciosas y que tienen el parámetro `_rescode` con el valor `NOERROR`. De manera similar, la Figura 5.8 muestra el mismo caso, pero para las solicitudes que tienen el parámetro `_rescode` con el valor `NXDOMAIN`.

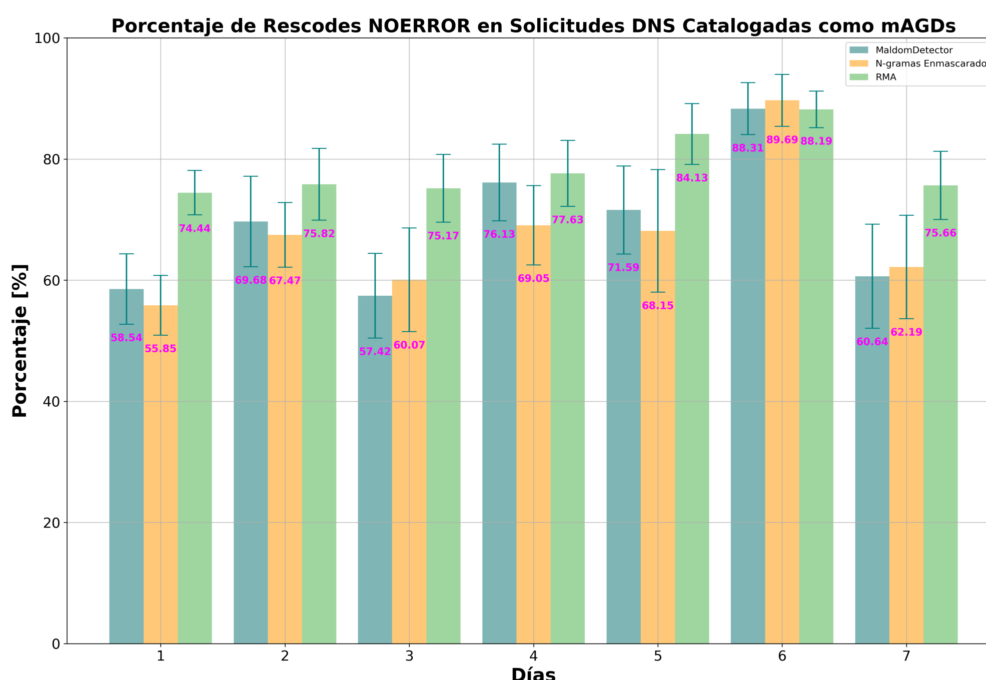


Figura 5.7: Porcentaje de solicitudes DNS catalogadas como mAGDs con `_rescode:NOERROR`.

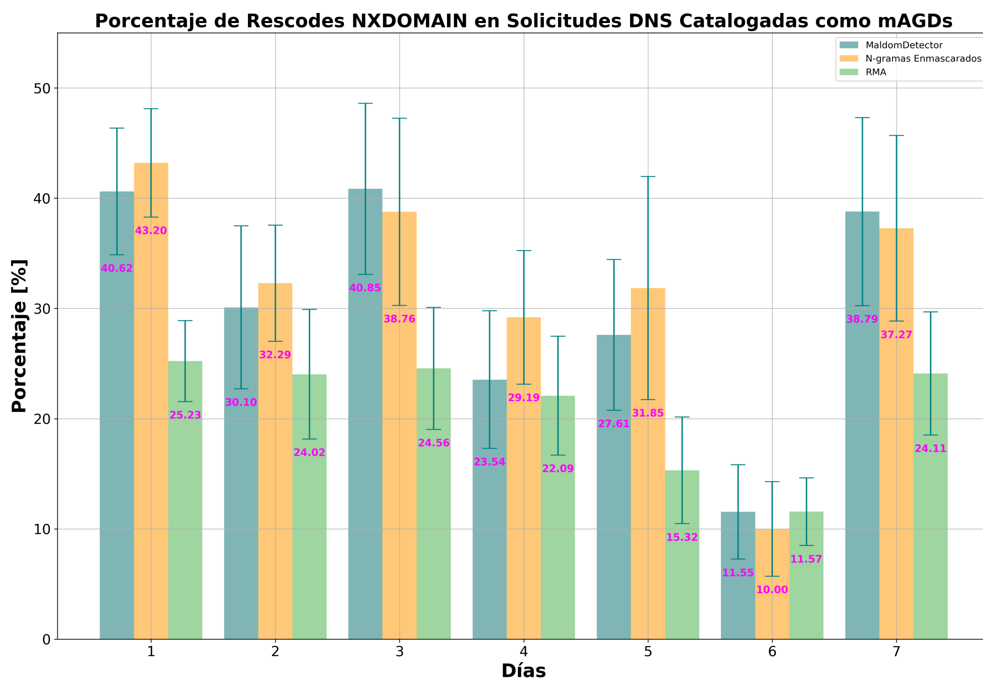


Figura 5.8: Porcentaje de solicitudes DNS catalogadas como mAGDs con `_rescode:NXDOMAIN`.

Al analizar las Figuras 5.7 y 5.8, se confirma la inferencia realizada sobre la tasa de detección del RMA. Se observa que los mAGDs detectados por este algoritmo presentan, en general, una mayor cantidad de parámetros `_rescode` con el valor `NOERROR` y una menor cantidad de parámetros `_rescode` con el valor `NXDOMAIN`, en comparación con los otros dos algoritmos analizados. Estos resultados sugieren que el RMA tiene una tasa de falsos positivos más alta. Por lo tanto, se concluye que el uso del algoritmo de detección *MaldomDetector* o el de N-gramas enmascarados en su lugar representaría una mejora en las capacidades de detección de mAGDs para el *framework* BNDF.

Además, al realizar el mismo análisis para los algoritmos basados en aprendizaje automático, se observa que durante el período de análisis, el algoritmo *MaldomDetector* detecta una mayor tasa de mAGDs con el parámetro `_rescode` en `NOERROR`, así como la menor tasa de mAGDs con el parámetro `_rescode` en `NXDOMAIN`. Esto sugiere que el algoritmo de N-gramas enmascarados tiene una capacidad de detección más confiable que la del algoritmo *MaldomDetector*.

5.4. Evaluación del uso de recursos computacionales

Si bien es importante conocer el rendimiento de la clasificación por parte de los algoritmos de detección para la elección e implementación de una solución en el contexto de una red real, existen otros factores igualmente relevantes a considerar. Entre ellos el tamaño de almacenamiento del modelo de aprendizaje entrenado y el tiempo de procesamiento, también pueden ser determinantes al seleccionar una solución, especialmente cuando se trabaja con recursos computacionales limitados. En la Tabla 5.6 se presentan varias mediciones de los recursos computacionales usados por los algoritmos *MaldomDetector* y *N-gramas enmascarados*.

Tabla 5.6: Evaluación de los recursos computacionales ocupados por los algoritmos.

Métricas	**MaldomDetector**	**N-gramas enmascarados**
Tiempo promedio de extracción de características por dominio	0.02 ms	0.37 ms
Tiempo promedio de predicción por dominio	1.35 ms	1.86 ms
Tiempo promedio total de procesamiento por dominio	1.38 ms	2.23 ms
Tamaño del modelo entrenado	91 MB	982 MB

Para la obtención de los resultados detallados en la Tabla 5.6, se ha utilizado una base de datos equilibrada de 50000 muestras que incluyen mAGDs y dominios legítimos nunca antes vistos en las etapas de entrenamiento de los modelos de aprendizaje. Para obtener un valor representativo de todas las muestras, se ha calculado el promedio sobre todas las predicciones individuales para cada métrica. A partir de dichos resultados, se concluye que el algoritmo *MaldomDetector* tiene una ventaja en el tiempo de procesamiento en comparación con el algoritmo de N-gramas enmascarados. *MaldomDetector* logra extraer las 12 características necesarias en un tiempo de 0.02 ms, mientras que el algoritmo de N-gramas enmascarados requiere 0.37 ms para extraer las 44 características requeridas. Además, en la predicción de un nombre de dominio específico, *MaldomDetector* es aproximadamente 0.5 ms más rápido, por lo tanto, este algoritmo presenta una ventaja en situaciones donde se requiera de respuestas rápidas en la clasificación de dominios. En cuando al uso de almacenamiento, nuevamente el algoritmo *MaldomDetector* se posiciona como ganador, debido a que su implementación basada en KNN, emplea únicamente 91 MB, mientras que el algoritmo de N-gramas enmascarados con su implementación de *Random Forest*, requiere

casi 10 veces más de almacenamiento, con 982 MB.

En base a la interpretación de los resultados, se puede concluir que el algoritmo *MaldomDetector* presenta un rendimiento superior en términos de uso de recursos computacionales, por lo cual, en este contexto resulta ser la mejor opción para su implementación en redes donde los recursos sean limitados.

Conclusiones y trabajos futuros

En esta sección, se analizarán y discutirán los hallazgos más relevantes, se destacarán las contribuciones realizadas y se evaluará el cumplimiento de los objetivos planteados inicialmente. En la Sección 6.1 se consolidarán las implementaciones propuestas (Sección 4) así como los resultados obtenidos (Sección 5), con el objetivo de generar una visión integral y concluyente de todo el trabajo realizado. En la Sección 6.2 se discutirán los inconvenientes presentados durante las implementaciones propuestas, junto con las consideraciones que se deben tener en cuenta para la obtención de resultados válidos y de interés científico. Finalmente, en la Sección 6.3 se detallan enfoques de implementación que podrían incrementar el rendimiento de los sistemas de detección de mAGDs, trascendiendo el alcance del presente trabajo.

6.1. Conclusiones

A través de una exhaustiva revisión del estado del arte ha sido posible explorar distintas soluciones actuales, novedosas y con resultados altamente prometedores en el campo de la detección de Botnets. Entre ellas, se destacan dos soluciones de particular interés que han sido implementadas a lo largo de este documento: *MaldomDetector* y N-gramas enmascarados. Ambos algoritmos basan su funcionamiento en el uso de aprendizaje automático supervisado, para lo cual, proponen un enfoque de extracción de características léxicas y estadísticas a partir a los nombres de dominio disponibles en las solicitudes DNS. Para la obtención de resultados cada autor definió una base de datos con la que idealmente estos algoritmos lograron alcanzar un valor de *accuracy* del 98 % para el caso de *MaldomDetector*, mientras que en el caso de N-gramas enmascarados fue del 97.04 %.

Lamentablemente, debido a la falta de solidaridad científica por parte de los autores de los algoritmos mencionados, gran parte de los recursos utilizados en sus implementaciones no se encuentran disponibles al público. Por esta razón se ha optado por utilizar una base de datos diversa, extensa y de libre distribución denominada UMUDGA con el fin de obtener resultados más generalizables y replicables en diferentes contextos e implementaciones. Aun así, con esta base de datos se han obtenido resultados que

difieren a los presentados por los autores de cada uno de los algoritmos. Específicamente en el presente caso se alcanzó un valor máximo de *accuracy* del 79.22 % para *MaldomDetector*, mientras que para N-gramas enmascarados del 85.09 %.

Durante las pruebas iniciales realizadas, se han utilizado 4 algoritmos de aprendizaje automático supervisado: KNN, SVM, *Random Forests* y MLP. Esto se realizó con objetivo de identificar el algoritmo de aprendizaje que mejor se adapte a los algoritmos de detección y a sus respectivos enfoques de extracción de características. En el caso de *MaldomDetector* se obtuvieron los mejores resultados con el algoritmo KNN mientras que para N-gramas enmascarados, *Random Forests* presentó el mejor desempeño. Después de las evaluaciones realizadas de manera *offline*, se concluye categóricamente que el algoritmo de N-gramas enmascarados presenta el mejor rendimiento en términos de detección a diferencia de los algoritmos *MaldomDetector* y RMA.

Luego de realizar un exhaustivo análisis acerca de la arquitectura y funcionamiento de BNDF, se concluyó que la mejor contribución posible sería dotar al *framework* con la capacidad de detectar mAGDs en tiempo real. Para esto se diseñó un módulo de detección temprana independiente de los módulos nativos, conformado por los algoritmos de detección *MaldomDetector*, N-gramas enmascarados y RMA. Su implementación permite la ejecución simultanea de dichos algoritmos, con el objetivo principal de emitir alertas inmediatas ante la presencia de posibles mAGDs, pero también determinar el algoritmo con el mejor desempeño durante el periodo de análisis.

Una vez implementado el módulo de detección temprana se llevaron a cabo diferentes escenarios de evaluación, con el objetivo de inferir el rendimiento de detección de los 3 algoritmos mencionados. En este contexto, utilizar las métricas de evaluación *Accuracy*, *Precision*, *Recall* y *F1-Score*, resulta inviable. Se puede concluir que es incorrecto asumir que el rendimiento obtenido a través de las métricas de evaluación en el escenario *offline* representará con exactitud el rendimiento de los algoritmos en un escenario en tiempo real. Sin embargo, estas métricas proporcionan un buen punto de partida para seleccionar la mejor configuración de los algoritmos.

Mediante las pruebas en tiempo real (escenario *online*) realizadas en la red de la IES, se concluyó que el parámetro `_rescode` disponible en las solicitudes DNS es útil para

determinar la validez de las predicciones realizadas por los algoritmos de detección. Específicamente, al analizar los `_rescode` se esperaría que la cantidad de solicitudes DNS catalogadas como maliciosas y con el valor `_rescode:NXDOMAIN` sean altas, mientras que con el valor `_rescode:NOERROR` sean bajas. En base a este análisis se pudo concluir que el RMA fue el algoritmo de detección que generaba la mayor cantidad de falsos positivos, haciéndolo poco deseable. Por otro lado, el algoritmo de N-gramas enmascarados, a pesar de poseer un comportamiento similar al algoritmo *Maldom-Detector*, presentó la tasa de falsos positivos más baja de todas y por lo tanto, se posicionó como la mejor opción a implementar en la red de la IES, concordando con la tendencia mostrada en las pruebas del escenario *offline*.

Finalmente, en la evaluación del uso de recursos computacionales de los algoritmos de detección basados en aprendizaje automático, se pudo determinar que el algoritmo *MaldomDetector* posee el tiempo de procesamiento promedio por dominio más bajo, con un valor de 1.38 ms, en comparación con los 2.23 ms del algoritmo de N-gramas enmascarados. Este resultado cobra sentido al analizar la cantidad de características empleadas por cada algoritmo: *MaldomDetector* procesa solo 12 características, mientras que N-gramas enmascarados utiliza 44. Así mismo, el algoritmo de aprendizaje automático KNN empleado por *MaldomDetector* requiere únicamente un espacio de almacenamiento de 91 MB, mientras que el algoritmo *Random Forests* empleado por N-gramas enmascarados requiere de 982 MB de almacenamiento. De los datos presentados, se puede concluir que en el caso en donde los recursos computacionales sean limitados, el algoritmo *MaldomDetector* puede ser una solución viable a implementar a pesar de poseer un rendimiento de detección menor en comparación al algoritmo de N-gramas enmascarados.

6.2. Recomendaciones

- Durante la etapa de entrenamiento de los algoritmos de aprendizaje automático supervisado, se recomienda utilizar la técnica de validación cruzada. Esto ayuda a evitar resultados sobreestimados que indiquen un rendimiento superior al real en términos de clasificación.

- El uso de una base de datos equilibrada contribuirá a que el algoritmo de aprendizaje entrenado sea más generalizado. Esto a su vez permitirá que las métricas de evaluación, como la *Precision* y *Recall*, presenten una proporcionalidad directa en relación con la cantidad de falsos positivos y falsos negativos, respectivamente. Esto debido a que estas métricas son sensibles al desequilibrio de clases.
- Es posible que la base de datos utilizada en el entrenamiento de los algoritmos de aprendizaje no contenga nombres de dominio que se consulten de manera recurrente en una red real específica. Por lo tanto, se recomienda actualizar de forma continua la *Whitelist* implementada en el módulo de detección temprana. Esto ayudará a evitar el procesamiento de dominios repetitivos y, en consecuencia, reducir la tasa de falsos positivos
- Aunque los algoritmos de detección permiten identificar mAGDs en el tráfico de red, para determinar el host específico que ha sido infectado, es necesario conocer la dirección IP privada correspondiente. En el presente caso, esta información se ha visto limitada por las políticas de seguridad del ISP que proporcionó el reenvío del tráfico analizado. Sin embargo, un profesional de seguridad con los permisos adecuados no debería tener dificultades para acceder a esta información.

6.3. Trabajos Futuros

- Durante la revisión del estado del arte, se identificaron diversos enfoques de detección que difieren de las soluciones propuestas en este trabajo. Estos enfoques consideran no solo el SLD, sino también el nombre de dominio completo (FQDN) para la extracción de características. Por lo tanto, un trabajo futuro podría analizar la influencia del uso del nombre de dominio completo en la tasa de detección, comparándola con la utilización exclusiva del SLD.
- Aunque los enfoques propuestos permiten identificar mAGDs pertenecientes a 50 familias de DGAs diferentes, no abordan específicamente la identificación de la familia generadora. La capacidad de identificar las familias responsables de

generar mAGDs es relevante en escenarios reales donde existe un desequilibrio de clases. Identificar las clases mayoritarias de DGAs presentes en una red real permitiría enfocar los esfuerzos en la detección y bloqueo de estas familias específicas.

- Aunque la arquitectura presentada en este documento se centra en la detección en tiempo real realizada por cada uno de los algoritmos, al analizar la arquitectura original de BNDF, también es posible incorporar estos algoritmos después de la etapa de detección de anomalías. Por lo tanto, un trabajo futuro interesante sería analizar el rendimiento de cada algoritmo cuando se ubican en una etapa posterior a la detección de anomalías.
- Tras revisar la literatura existente, se ha observado que varios enfoques de detección proponen el uso de un sistema de votación para clasificar los nombres de dominio. A partir de este enfoque, resulta interesante considerar la implementación de un sistema de detección conjunto que combine los algoritmos *Maldom-Detector*, N-gramas enmascarados y un tercer algoritmo de detección adicional. Este enfoque de votación tiene el potencial de generar una tasa de detección más confiable en comparación con la utilización individual de los algoritmos.

Referencias

[1] Mathworks, “Support Vector Machine (SVM) - MATLAB & Simulink.” [En línea]. Disponible: https://la.mathworks.com/discovery/support-vector-machine.html

[2] Aurélien Géron, *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems*, 2da ed., 2019. [En línea]. Disponible: https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/

[3] A. O. Almashhadani, M. Kaiiali, D. Carlin, y S. Sezer, “Maldomdetector: A system for detecting algorithmically generated domain names with machine learning,” *Computers & Security*, vol. 93, p. 101787, 2020. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S0167404820300729

[4] V. Quezada, F. Astudillo-Salinas, L. Tello-Oquendo, y P. Bernal, “Real-time bot infection detection system using dns fingerprinting and machine-learning,” *Computer Networks*, vol. 228, p. 109725, 2023. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S1389128623001706

[5] R. Kemmerer, “Cybersecurity,” in *25th International Conference on Software Engineering, 2003. Proceedings.*, 2003, pp. 705–715.

[6] A. Sundaram, “An introduction to intrusion detection,” *XRDS*, vol. 2, num. 4, p. 3–7, apr 1996. [En línea]. Disponible: https://doi.org/10.1145/332159.332161

[7] A. J. P., “Computer security threat monitoring and surveillance,” *Technical Report, James P. Anderson Company*, 1980. [En línea]. Disponible: https://cir.nii.ac.jp/crid/1573950399661362176

[8] M. H. Bhuyan, D. K. Bhattacharyya, y J. K. Kalita, “Network anomaly detection: Methods, systems and tools,” *IEEE Communications Surveys & Tutorials*, vol. 16, num. 1, pp. 303–336, 2014.

[9] T.-S. Wang, H.-T. Lin, W.-T. Cheng, y C.-Y. Chen, “Dbod: Clustering and detecting dga-based botnets using dns traffic analysis,” *Computers & Security*, vol. 64, pp. 1–15, 2017. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S0167404816301250

[10] M. Singh, M. Singh, y S. Kaur, "Detecting bot-infected machines using dns fingerprinting," *Digital Investigation*, vol. 28, pp. 14–33, 2019. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S174228761830272X

[11] S. Cole, "An analysis of the evolution of botnets," *Iowa State University*, p. 15, 2019. [En línea]. Disponible: https://dr.lib.iastate.edu/entities/publication/8c171389-caae-46f6-bc6b-8c585fa74918

[12] Securelist by Kaspersky, "Bots and botnets in 2018," 2019. [En línea]. Disponible: https://securelist.com/bots-and-botnets-in-2018/90091/

[13] Spamhaus Malware Team, "Spamhaus Botnet Threat Update: Q1-2021," 2021. [En línea]. Disponible: https://www.spamhaus.org/news/article/809/spamhaus-botnet-threat-update-q1-2021

[14] X. H. Vu, X. D. Hoang, y T. H. H. Chu, "A novel model based on ensemble learning for detecting dga botnets," in *2022 14th International Conference on Knowledge and Systems Engineering (KSE)*, 2022, pp. 1–6.

[15] P. M. Anand, T. G. Kumar, y P. S. Charan, "An ensemble approach for algorithmically generated domain name detection using statistical and lexical analysis," *Procedia Computer Science*, vol. 171, pp. 1129–1136, 2020, third International Conference on Computing and Network Communications (CoCoNet'19). [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S1877050920310991

[16] T. Wang, L.-C. Chen, y Y. Genc, "A dictionary-based method for detecting machine-generated domains," *Information Security Journal: A Global Perspective*, vol. 30, num. 4, pp. 205–218, 2021. [En línea]. Disponible: https://doi.org/10.1080/19393555.2020.1834650

[17] T. A. Tuan, H. V. Long, y D. Taniar, "On detecting and classifying dga botnets and their families," *Computers & Security*, vol. 113, p. 102549, 2022. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S0167404821003734

[18] A. M. Manasrah, T. Khdour, y R. Freehat, “Dga-based botnets detection using dns traffic mining,” *Journal of King Saud University - Computer and Information Sciences*, vol. 34, num. 5, pp. 2045–2061, 2022. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S1319157822000726

[19] Z. Wang, Y. Guo, y D. Montgomery, “Machine learning-based algorithmically generated domain detection,” *Computers and Electrical Engineering*, vol. 100, p. 107841, 2022. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S0045790622001331

[20] V. Hanh, D. Hoang, y T. Chu, “A novel model based on ensemble learning for detecting dga botnets,” 10 2022.

[21] A. Cucchiarelli, C. Morbidoni, L. Spalazzi, y M. Baldi, “Algorithmically generated malicious domain names detection based on n-grams features,” *Expert Systems with Applications*, vol. 170, p. 114551, 2021. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S0957417420311957

[22] C. Xu, J. Shen, y X. Du, “Detection method of domain names generated by dgas based on semantic representation and deep neural network,” *Computers & Security*, vol. 85, pp. 77–88, 2019. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S0167404818312938

[23] H. Shahzad, A. R. Sattar, y J. Skandaraniyam, “From real malicious domains to possible false positives in dga domain detection,” in *2021 IEEE 13th International Conference on Computer Research and Development (ICCRD)*, 2021, pp. 6–10.

[24] White Opps, “The Methbot Operation The Methbot Operation,” pp. 1–30, 2016. [En línea]. Disponible: https://www.whiteops.com/hubfs/Resources/WO_Methbot_Operation_WP.pdf

[25] A. Karim, R. B. Salleh, M. Shiraz, S. A. A. Shah, I. Awan, y N. B. Anuar, “Botnet detection techniques: review, future trends, and issues,” *Journal of Zhejiang University SCIENCE C*, vol. 15, num. 11, pp. 943–983, 2014. [En línea]. Disponible: https://doi.org/10.1631/jzus.C1300242

[26] M. Singh, M. Singh, y S. Kaur, “Detecting bot-infected machines using dns fingerprinting,” *Digital Investigation*, vol. 28, pp. 14–33, 2019. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S174228761830272X

[27] K. Alieyan, A. ALmomani, A. Manasrah, y M. M. Kadhum, “A survey of botnet detection based on DNS,” *Neural Computing and Applications*, vol. 28, num. 7, pp. 1541–1558, 2017. [En línea]. Disponible: https://doi.org/10.1007/s00521-015-2128-0

[28] M. Grill, I. Nikolaev, V. Valeros, y M. Rehak, “Detecting dga malware using netflow,” in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2015, pp. 1304–1309.

[29] X. D. Hoang y X. H. Vu, “An improved model for detecting dga botnets using random forest algorithm,” *Information Security Journal: A Global Perspective*, vol. 31, num. 4, pp. 441–450, 2022. [En línea]. Disponible: https://doi.org/10.1080/19393555.2021.1934198

[30] J. Liang, S. Chen, Z. Wei, S. Zhao, y W. Zhao, “Hagdetector: Heterogeneous dga domain name detection model,” *Computers & Security*, vol. 120, p. 102803, 2022. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S0167404822001973

[31] K. Thakur, H. Alqahtani, y G. Kumar, “An intelligent algorithmically generated domain detection system,” *Computers & Electrical Engineering*, vol. 92, p. 107129, 2021. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S0045790621001336

[32] M. Zago, M. Gil Pérez, y G. Martínez Pérez, “Umudga: A dataset for profiling dga-based botnet,” *Computers & Security*, vol. 92, p. 101719, 2020. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S0167404820300067

[33] D. Yan, H. Zhang, Y. Wang, T. Zang, X. Xu, y Y. Zeng, “Pontus: A linguistics-based dga detection system,” in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.

[34] V. Hanh y D. Hoang, “A novel machine learning-based approach for detecting word-based dga botnets,” *Journal of Theoretical and Applied Information Technology*, vol. 99, 12 2021.

[35] C. Chio y D. Freeman, *Machine Learning and Security: Protecting Systems with Data and Algorithms*, 1ra ed. O’Reilly Media, 2018.

[36] A. C. Müller y S. Guido, *Introduction to Machine Learning with Python*, 1ra ed., D. Schanafelt, Ed. O’Reilly Media, 2022.

[37] K. L. Priddy y P. E. Keller, *Artificial Neural Networks: An Introduction*. SPIE - The International Society for Optical Engineering, 2009.

[38] M. Kuhn y K. Johnson, *Applied predictive modeling*, 1ra ed. New York: Springer, 2013.

[39] A. Mammone, M. Turchi, y N. Cristianini, “Support vector machines,” *WIREs Computational Statistics*, vol. 1, num. 3, pp. 283–289, 2009. [En línea]. Disponible: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wics.49

[40] S. Russell y P. Norvig, *Artificial Intelligence A Modern Approach*, 3ra ed., 2010, vol. 4.

[41] M. Singh, M. Singh, y S. Kaur, “Issues and challenges in dns based botnet detection: A survey,” *Computers & Security*, vol. 86, pp. 28–52, 2019. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S0167404819301117

[42] J. Peck, C. Nie, R. Sivaguru, C. Grumer, F. Olumofin, B. Yu, A. Nascimento, y M. De Cock, “Charbot: A simple and effective method for evading dga classifiers,” 05 2019.

[43] D.-T. Truong y G. Cheng, “Detecting domain-flux botnet based on dns traffic features in managed network,” *Security and Communication Networks*, vol. 9, num. 14, pp. 2338–2347, 2016. [En línea]. Disponible: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1495

[44] A. Satoh, Y. Nakamura, D. Nobayashi, y T. Ikenaga, “Estimating the randomness of domain names for dga bot callbacks,” *IEEE Communications Letters*, vol. 22, num. 7, pp. 1378–1381, 2018.

[45] C. Xu, J. Shen, y X. Du, “Detection method of domain names generated by dgas based on semantic representation and deep neural network,” *Computers & Security*, vol. 85, pp. 77–88, 2019. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S0167404818312938

[46] V. G. Quezada Pauta, “Sistema autónomo de detección de botnets en la red de la Universidad de Cuenca basado en el comportamiento anómalo del tráfico DNS,” Bachelor Thesis, Universidad de Cuenca, 2021. [En línea]. Disponible: http://dspace.ucuenca.edu.ec/handle/123456789/36031

[47] D. Plohmann, K. Yakdan, M. Klatt, J. Bader, y E. Gerhards-Padilla, “A comprehensive measurement study of domain generating malware.” in *USENIX Security Symposium*, vol. 10, num. 3241094.3241115, 2016.

[48] Amazon Wen Services, “Alexa top 1 million.” [En línea]. Disponible: s3.amazonaws.com/alexa-static/top-1m.csv.zip

[49] J. Selvi, R. J. Rodríguez, y E. Soria-Olivas, “Detection of algorithmically generated malicious domain names using masked n-grams,” *Expert Systems with Applications*, vol. 124, pp. 156–163, 2019. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S0957417419300648

[50] M. Kursa y W. Rudnicki, “Feature selection with boruta package,” *Journal of Statistical Software*, vol. 36, pp. 1–13, 09 2010.

[51] Netlab, “DGA - Netlab OpenData Project,” 2019. [En línea]. Disponible: https://data.netlab.360.com/dga/

[52] Amazon, “Alexa- static top 1 million data,” 2019. [En línea]. Disponible: http://s3.amazonaws.com/alexa-static/top-1m.csv.zip

[53] M. Anand, “DGA-Proj,” 2019. [En línea]. Disponible: https://github.com/PMohanAnand/DGA-Proj

[54] M. Zago, M. Gil Pérez, y G. Martínez Pérez, “Umudga: A dataset for profiling algorithmically generated domain names in botnet detection,” *Data in Brief*, vol. 30, p. 105400, 2020. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S2352340920302948

[55] “scikit-learn: machine learning in Python — scikit-learn 1.2.2 documentation.” [En línea]. Disponible: https://scikit-learn.org/stable/

[56] “sklearn.neighbors.KNeighborsClassifier — scikit-learn 1.2.2 documentation.” [En línea]. Disponible: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn-neighbors-kneighborsclassifier

[57] “sklearn.neighbors.KNeighborsClassifier — scikit-learn 1.2.2 documentation.” [En línea]. Disponible: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn-neighbors-kneighborsclassifier

[58] “sklearn.ensemble.RandomForestClassifier — scikit-learn 1.2.2 documentation.” [En línea]. Disponible: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier

[59] “sklearn.neural_network.MLPClassifier — scikit-learn 1.2.2 documentation.” [En línea]. Disponible: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier

[60] “sklearn.model_selection.GridSearchCV — scikit-learn 1.3.0 documentation.” [En línea]. Disponible: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html#sklearn.model_selection.GridSearchCV

Anexo A

A.1. Instalación de *BotNet Detection Framework* (BNDF)

Todos los procedimientos mostrados a continuación son realizados usando la distribución de Linux *Ubuntu* 20.04.1 LTS. Cabe mencionar que la máquina sobre la cual se ejecutará BNDF debe estar habilitada para recibir el reenvío de todas las solicitudes *Domain Name System* (DNS) de la red de análisis, específicamente en su puerto 10000.

A.1.1. Requerimientos previos

Previo a la instalación de las dependencias siguientes, es necesario actualizar la lista de paquetes disponibles en los repositorios configurados en el sistema. Para esto, se utilizan los comandos mostrados en el Extracto de código A.1.

```
1 sudo apt-get update
2 sudo apt-get upgrade
```

Extracto de código A.1: Actualización de dependencias.

El comando *Advanced Package Tool* (APT) es usado para referirse al sistema de administración de paquetes utilizado en sistemas operativos basados en la distribución Linux Debian.

A.1.1.1. Instalación de *docker 17.06.0*

Para instalar esta dependencia, es necesario descargar el paquete desde la página oficial, por lo tanto, en primera instancia se debe permitir que APT utilice repositorios HTTPS a través del comando mostrado en el Extracto de código A.2.

```
1 sudo apt-get install apt-transport-https ca-certificates curl gnupg
    lsb-release
```

Extracto de código A.2: Inclusión de repositorios HTTPS en APT.

Agregar la clave GPG oficial de *Docker* mediante el comando presente en el Extracto de código A.3.

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg
   --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

Extracto de código A.3: Descarga de la clave oficial GPG de *Docker*.

Añadir el repositorio *Docker* al sistema usando el comando del Extracto de código A.4.

```
echo \ "deb [arch=amd64
   signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
   https://download.docker.com/linux/ubuntu \ $(lsb_release -cs)
   stable" | sudo tee /etc/apt/sources.list.d/docker.list >
   /dev/null
```

Extracto de código A.4: Inclusión de *Docker* en los repositorios del sistema.

Actualizar el índice de paquetes APT nuevamente, con el comando presente en el Extracto de código A.5.

```
sudo apt-get update
```

Extracto de código A.5: Actualización de dependencias.

Instalar la última versión de *Docker* mediante el comando en el Extracto de código A.6.

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Extracto de código A.6: Instalación de *Docker*.

Para comprobar que *Docker* se ha instalado correctamente, se puede ejecutar el comando mostrado en el Extracto de código A.7.

```
sudo docker run hello-world
```

Extracto de código A.7: Comprobación del funcionamiento de *Docker*.

Este comando descargará una imagen de prueba y la ejecutará en un contenedor. Si todo funciona correctamente, debería ver un mensaje que indica que *Docker* está funcionando correctamente.

A.1.1.2. Instalación de *docker-compose* 1.27.0

Descargar la última versión estable de *Docker Compose* e instalarla en la ubicación "*/usr/local/bin/docker-compose*", mediante el comando mostrado en el Extracto de código A.8.

```
sudo curl -L
"https://github.com/docker/compose/releases/download/1.29.2/docker-
compose -$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Extracto de código A.8: Instalacion de *Docker Compose*.

Asignar al archivo "*docker-compose*" permisos de ejecución mediante el comando mostrado en el Extracto de código A.9.

```
sudo chmod +x /usr/local/bin/docker-compose
```

Extracto de código A.9: Asignación de permisos de ejecución.

Para comprobar que *Docker Compose* se ha instalado correctamente, ejecutar el comando del Extracto de código A.10.

```
docker-compose --version
```

Extracto de código A.10: Comprobación del funcionamiento de *Docker Compose*.

A.1.1.3. Instalación de *git*

Instalar *Git*, mediante el comando mostrado en el Extracto de código A.11.

```
sudo apt-get install git
```

Extracto de código A.11: Instalación de *Git*.

Para comprobar que *Git* se ha instalado correctamente, ejecutar el comando del Extracto de código A.12.

```
git --version
```

Extracto de código A.12: Comprobación del funcionamiento de *Git*.

A.1.1.4. Instalación de *curl*

Instalar *Curl*, mediante el comando mostrado en el Extracto de código A.13.

```
sudo apt-get install curl
```

Extracto de código A.13: Instalación de *Curl*.

Para comprobar que *Curl* se ha instalado correctamente, ejecutar el comando del Extracto de código A.14.

```
curl --version
```

Extracto de código A.14: Comprobación del funcionamiento de *Curl*.

A.1.1.5. Instalación de *time*

Instalar *time*, mediante el comando mostrado en el Extracto de código A.15.

```
sudo apt-get install time
```

Extracto de código A.15: Instalación de *Time*.

Para comprobar que *time* se ha instalado correctamente, ejecutar el comando del Extracto de código A.16.

```
time --version
```

Extracto de código A.16: Comprobación del funcionamiento de *Time*.

A.1.2. Instalación

Como primera instancia, se debe descargar el repositorio completo del proyecto, el cual fue presentado en [4], y está disponible en `https://github.com/fabianastudillo/bndf.git`, por medio del comando mostrado en el Extracto de código A.17.

```
git clone https://github.com/fabianastudillo/bndf.git
```

Extracto de código A.17: Descarga del repositorio completo de BNDF.

Como resultado de la descarga, se tendrá una carpeta en el sistema denominada "*bndf*". Para poner en funcionamiento el *framework*, es necesario ubicarse en esta carpeta e iniciar y ejecutar los contenedores que emplea BNDF para su funcionamiento, por medio de los comandos presentes en el Extracto de código A.18.

```
cd bndf/
docker-compose build
docker-compose up
docker-compose up -d python
```

Extracto de código A.18: Compliación y ejecución de BNDF.

Se puede comprobar que el *framework* está ejecutándose correctamente cuando los contenedores se encuentren en estado "*healthy*". Esto puede verificarse por medio del comando presente en el Extracto de código A.19. Sin embargo, cabe mencionar que este proceso no es inmediato, por lo cual, es necesario esperar que el levantamiento de los contenedores finalice.

```
docker ps
```

Extracto de código A.19: Estado de los contenedores de BNDF.

Finalmente, para ingresar a los distintos contenedores generados ("*logstash*", "*elasticsearch*", "*kibana*" y "*python*"), se debe emplear el comando mostrado en el Extracto de código A.20.

```
docker exec -it [container-name] bash
```

Extracto de código A.20: Ingreso a un contenedor específico en ejecución.