



Trabajo Fin de Grado

Herramienta de identificación e informe de activos y servicios de red basado en análisis de tráfico y huella de sistemas operativos.

Tool for identification and report of network assets and services based on traffic analysis and operating systems fingerprint.

Autor

Javier Ortega Palacios

Director

Álvaro Alesanco Iglesias

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2020

AGRADECIMIENTOS

A mi familia, pareja y amigos, por estar conmigo en estos cuatro años apoyándome.
Sin vosotros no habría sido posible.

A Álvaro Alesanco y Antonio Sanz por la oportunidad.

Lista de Acrónimos

TIC Tecnologías de la información y la comunicación

CMDB Configuration Management DataBase

IP Internet Protocol

ITIL Information Technology Infrastructure Library

OGC Office of Government Commerce

ISO International Organization for Standardization

IEC International Electrotechnical Commission

TADDM Tivoli Application Dependency Discovery Manager

SCCM System Center Configuration Manager

IDE Integrated Development Environment

JSON JavaScript Object Notation

HTTP Hypertext Transfer Protocol

HTML HyperText Markup Language

CSV Comma Separated Values

GEXF Graph Exchange XML Format

CI Configuration Item

UID Unique Identifier

ECMA European Computers Manufacturers Association

IETF Internet Engineering Task Force

TFG Trabajo Fin de Grado

TCP Transmission Control Protocol

UDP User Datagram Protocol

OS/SO Sistema Operativo

TTL Time To Live

SSH Secure Shell Protocol

FTP File Transfer Protocol

SSL Secure Socket Layer

TLS Transport Layer Secure

IDS Intrusion Detection System

SPAN Switched Port Analyzer

ICMP Internet Control Message Protocol

GUI Graphical User Interface

RDP Remote Desktop Protocol

USMA United States Military Academy

NSA National Security Agency

PCAP Packet Capture

RESUMEN

En todas las organizaciones, la gestión de activos TIC es una tarea complicada. Tener estos activos actualizados y contabilizados es una tarea tanto tediosa como imprescindible para mantener los niveles de seguridad adecuados y mitigar o evitar posibles ataques que se puedan recibir. Para este propósito, se intenta mantener una denominada CMDB (*Configuration Management Database*), en la que se debe reflejar tanto los activos que hay en la empresa en forma de dirección IP (dirección del *Internet Protocol*) como los servicios de red que usen, la versión de los sistemas operativos y software que está instalado como la versión de este. De esta manera, cuando un administrador de la organización en cuestión quiera aplicar una mejora o un parche a los equipos, sabrá sobre qué equipos debe actuar y en qué dirección de red se encuentra. En la CMDB es importante tener también el responsable o responsables de cada activo y el lugar físico donde se encuentra en la empresa. En la base de datos se pueden añadir muchas más características de los equipos como el coste, las mejoras realizadas y el tiempo de vida del propio dispositivo, aunque nos centraremos únicamente en la parte de red.

En este trabajo se ha creado una herramienta capaz de crear esta CMDB con información extraída de una captura de tráfico de red. Este, es una primera aproximación de lo que puede ser un software profesional de gestión de activos, basado en captura de tráfico pasiva, sin interferir en la red ni afectar a su rendimiento, y sin instalar ningún tipo de software en los activos. A día de hoy, no hay un software que haga esto de manera automática y usando tráfico de red, así que este trabajo puede servir para mejorar la automatización de este proceso en las organizaciones y la mejora del software ya existente para este propósito.

Índice

1. Introducción	1
1.1. Gestión de los activos: Bases de datos de gestión de configuración (CMDB)	1
1.2. Objetivos del trabajo	2
1.3. Materiales utilizados	2
1.4. Organización de la memoria	4
2. Conceptos previos	5
2.1. CIs (Configuration Items)	5
2.2. JSON	5
2.3. JSON Schema	6
2.4. Servicios de red	6
2.5. Huella de Sistemas Operativos (<i>OS Fingerprinting</i>)	7
2.6. Huella JA3	7
3. Análisis de Zeek	9
3.1. Introducción y arquitectura de Zeek	9
3.2. Programación en Zeek. Fichero local.zeek	12
3.3. Ficheros log generados por Zeek	12
4. Diseño y desarrollo de la herramienta	15
4.1. Diseño y programación de la herramienta principal	15
4.1.1. Recolección de datos y análisis	15
4.1.2. Generación del informe web	21
4.1.3. Generación del grafo de red	22
4.2. Desarrollo del script de instalación	23
4.3. Uso de Docker. Desarrollo del archivo Dockerfile	24
5. Evaluación y mejoras	27
5.1. Efectividad	27
5.1.1. Evaluación utilizando dataset	27

5.1.2. Evaluación en una red real	29
5.2. Rendimiento	31
5.3. Futuras mejoras de la herramienta	34
5.3.1. Mejoras para la detección	34
5.3.2. Otras mejoras	34
6. Conclusiones	37
Bibliografía	39
Lista de Figuras	41
Lista de Tablas	43
Anexos	44
A. Código de la disección de los logs de Zeek	47
B. Código de la página web del informe	55
C. Código del ejecutable de instalación	57
D. Manejo de GEPHI para extraer información de red	59
E. Código del Dockerfile	65
F. Ejemplo de resultados	67
F.1. Archivo data.json	67
F.2. Informe HTML	67
F.3. Grafo de red	69

Capítulo 1

Introducción

1.1. Gestión de los activos: Bases de datos de gestión de configuración (CMDB)

Las CMDB o Bases de datos de configuración es un concepto aparecido a finales de los años 80 del siglo pasado, y fue desarrollado para la tercera versión de la *Information Technology Infrastructure Library* (ITIL), una librería de buenas prácticas para la gestión de distintos aspectos de las TIC creada por la *Office of Government Commerce*(OGC), división del Ministerio de Hacienda Británico. Esta librería fue la precursora de otros documentos de buenas prácticas como son las normas ISO/IEC 20000, estándares internacionales de gestión de las TIC. Para el final de los años 90, el crecimiento exponencial de las tecnologías de la información en las organizaciones llevó a tomar en serio el uso de este tipo de bases de datos, ya que se habían convertido en una parte importante del proceso de negocio [18]. A día de hoy, cada vez es más crítico el tener una de estas bases de datos en las empresas, ya que no solo ayudan en el propio proceso de negocio, sino que tienen una implicación directa en la seguridad de los datos de las organizaciones, una de las cosas más importantes actualmente.

Una de las principales actividades en un departamento de microinformática de una organización es la actualización de software y de sistemas operativos, al igual que la gestión y control de las direcciones IP. Esto, alineado con los departamentos de seguridad informática y con el de sistemas, constituyen una pieza clave en la seguridad, gestión y continuidad de la actividad. Esta gestión es difícil de realizar, y cada vez más complicado por el gran número de dispositivos que hay en nuestra red. Se prevé que para el 2025 habrá más de 41.6 billones de dispositivos conectados a Internet [10], lo que complicará cada vez más esta tarea de mantenimiento de activos. Además de mantener la gestión de los activos para proteger la seguridad de la organización sabiendo qué hay conectado a la red corporativa, esta gestión puede servir para descubrir ataques o amenazas potenciales si se descubre un equipo que no debería estar conectado y

generando tráfico, como ocurrió en la NASA en 2019 [20].

En esta línea, hay empresas tecnológicas que han desarrollado herramientas para mantener esta base de datos de los activos. Estas herramientas suelen ser de pago y su mantenimiento es una tarea compleja y tediosa. Entre estas herramientas destacan Tivoli Application Dependency Discovery Manager (TADDM) de la empresa IBM, System Center Configuration Manager (SCCM) de Microsoft, Altiris de Symantec, KACE de Dell o OCS Inventory. La gran mayoría de ellas usan un aplicación instalada en los propios dispositivos, que guarda todo lo que hay instalado en este e información que pueda ser relevante para mantener el equipo seguro, pero ninguna de ellas usa tráfico de red o busca saber si hay algo más conectado a su red. Es por ello, que este trabajo puede ser una ayuda complementaria, que no sustitutiva, a este tipo de software de gestión, de manera que aumentaría la visibilidad de estas herramientas.

1.2. Objetivos del trabajo

El objetivo del trabajo es desarrollar una herramienta que de forma automática genere un informe de activos partiendo de una captura de tráfico realizada por ejemplo con la herramienta WireShark [8]. Este informe será únicamente basado en activos de tipo lógico (de la red), lo que no quiere decir que sea un informe completo, sino que es complementario y de apoyo. Los objetivos más detallados serían los siguientes:

- Familiarización y análisis de Zeek (antiguamente denominada Bro), software para extraer información de las capturas de tráfico de red en un formato apropiado para analizar.
- Programación de la herramienta en Python que diseccione los archivos creados por Zeek y recupere información relevante.
- Programación de la generación del informe de activos final.
- Familiarización con GEPHI y uso para extracción de información.

1.3. Materiales utilizados

Este trabajo al ser puramente software no se ha usado más material físico que un ordenador con una tarjeta de red, pero se ha necesitado el siguiente software gratuito:

- **Máquina Virtual Ubuntu 18.04 LTS:** Sistema operativo en el que se ha instalado Python, Zeek y se ha desarrollado y probado la herramienta.

- **Python 3.6:** Lenguaje de programación utilizado, que es de tipo interpretado y de software libre. Se usa la versión 3.6 de Python ya que es una versión estable y en la que son compatibles todas las librerías que se usan para el desarrollo.
- **PyCharm Community 2020.1 EAP:** Entorno de desarrollo (IDE) utilizado para programar la herramienta y probarla.
- **Librerías de Python utilizadas:**
 - **NumPy:** Librería para utilizar de manera sencilla matrices y vectores en Python.
 - **JSON:** Librería para manejar archivos en formato JSON.
 - **JSONSchema:** Librería para utilizar el formato JSON Schema en Python, usado para validar y comprobar un correcto JSON.
 - **Requests:** Librería para hacer peticiones HTTP de forma sencilla.
 - **Flask:** Librería para implementar un motor de páginas web en Python.
 - **JSON2HTML:** Librería que convierte un objeto JSON en una tabla HTML de manera sencilla.
- **Zeek:** Herramienta de análisis de tráfico de red (ver Capítulo 3).
- **p0f:** Herramienta de detección pasiva de sistemas operativos basado en huella de red de estos.
- **JA3 para Zeek:** Librería para Zeek de detección y recopilación de huella JA3 del protocolo SSL (ver Capítulo 2).
- **Brassfork:** Herramienta escrita en Go que extrae información de una captura de tráfico y lo convierte a dos archivos de nodos y uniones entre ellos [17].
- **CSVtoGEXF:** Herramienta escrita en Python que convierte los archivos CSV en formato GEXF para ser representada en Python con los campos que se le indique [16].
- **Capturas de tráfico de red:** Capturas de tráfico de red reales, extraídas de la página web de NETRESEC [5]. Además de estas, se han utilizado capturas de tráfico generadas artificialmente por mí para evaluar la herramienta.
- **VMWare Workstation:** Entorno de virtualización gratis de la empresa VM-Ware.

1.4. Organización de la memoria

La memoria está organizada como se indica a continuación:

- **Capítulo 1: Introducción.** En este capítulo se realiza una introducción al TFG, así como los objetivos y materiales utilizados.
- **Capítulo 2: Conceptos previos.** En este capítulo se va a explicar conceptos y herramientas que se usan junto a Zeek para la creación de la herramienta.
- **Capítulo 3: Análisis de Zeek.** En este capítulo se explica el funcionamiento de la herramienta Zeek y los archivos que este genera a partir de una captura de tráfico. Además, se explicarán los módulos utilizados y qué mejoras aportan.
- **Capítulo 4: Diseño y desarrollo de la herramienta.** En este capítulo se hace el diseño y se explica cómo se desarrollaron las distintas partes de la herramienta, además de los problemas y limitaciones que se han encontrado.
- **Capítulo 5: Evaluación y mejoras.** En este capítulo se analizan distintos aspectos de la herramienta, para ver su rendimiento y se proponen mejoras de cara al futuro y el desarrollo de la herramienta.
- **Capítulo 6: Conclusiones.** En este capítulo se realizan unas conclusiones sobre el trabajo, así como los pros y contras de la herramienta.
- **Anexo A: Código de la disección de los logs de Zeek.**
- **Anexo B: Código de la página web del informe.**
- **Anexo C: Código del ejecutable de instalación.**
- **Anexo D: Manejo de GEPHI para extraer información de red.**
- **Anexo E: Código del Dockerfile.**
- **Anexo F: Ejemplo de resultados.**

Capítulo 2

Conceptos previos

2.1. CIs (Configuration Items)

Los CIs (*Configuration Items*) o Objetos de Configuración son cada una de las entradas de la CMDB. Estos objetos son normalmente activos TIC como ordenadores, móviles, tabletas o equipos finales (*Endpoints*), equipos de red como son routers, firewalls, servidores u otro tipo de equipos. Estos Objetos de Configuración no solamente indican el tipo de equipo que es, sino que deben indicar los servicios que implementan, las acciones o cambios que se han realizado sobre ellos, las relaciones con otros CIs, los roles de acceso a este, el responsable o responsables del CI, etc. Dentro de nuestra herramienta cada uno de estos CIs se diferencian con un identificador único (*UID*), que en nuestro caso es la dirección IPv4 o IPv6. Además de la dirección IP, cada activo tiene distintos campos definidos en el JSON Schema como son los servicios de red, el software instalado, el posible sistema operativo, el JA3 y el posible User-Agent del equipo, pero únicamente la IP es un campo obligatorio en cada activo.

2.2. JSON

JavaScript Object Notation o Notación de Objetos de JavaScript es un formato de intercambio de datos, de manera sencilla de escribir para personas y fácilmente entendible por máquinas, que es independiente del lenguaje de programación que se utilice, ya que es un estándar de la ECMA Internacional (*European Computers Manufacturers Association*) [15]. Para seguir correctamente el lenguaje JSON, se deben seguir las instrucciones que se indican en la RFC 8259 [12]. Es esta, se definen las directrices para crear los Objetos, Arrays y Valores acordes al lenguaje JSON. Estos deben cumplir:

- **Valores:** Los valores del JSON deben ser un objeto, array, número, *string* (Cadena de texto), o los valores *false*, *true* o *null*.

- **Objetos:** Los objetos JSON son conjuntos de pares nombre-valor. Para cada nombre hay un valor del tipo indicado en el punto anterior. Cada uno de los conjuntos se separa con comas (,) y cada par se indica colocando dos puntos (:) entre ellos, y al principio y final se deben colocar llaves({}), quedando de la siguiente manera:

{nombre1:valor1, nombre2:valor2, ... nombreN:valorN}

- **Arrays:** Los array de JSON se deben crear de manera que comiencen y finalicen con corchetes ([]) y se coloquen objetos JSON entre ellos, separados por comas (,) quedando de la siguiente manera:

[{Objeto1}, {Objeto2}, ... {ObjetoN}]

2.3. JSON Schema

El JSON Schema es una forma de descripción de JSON, en la cual se especifica el tipo de dato y valor que se va a introducir en un objeto JSON. También se definen valores que deben ser obligatorios y valores que no, además de restricciones a estos mismos (por ejemplo, que un entero sea mayor que un entero concreto o que un string sea de una longitud determinada). Cuando se usa JSON Schema, se debe hacer una validación de este antes de añadirlo a un array de JSON, para verificar que está cumplido el esquema, y que los datos introducidos son válidos. A día de hoy no es un estándar, pero está en proceso de estandarización por el *Internet Engineering Task Force* (IETF), y se puede encontrar como borrador [22].

2.4. Servicios de red

Se entienden por servicios de red a los servicios utilizados por los dispositivos de red para comunicarse y compartir recursos e información, además de posibles configuraciones y otras funcionalidades. Cada servicio de red se identifica por un protocolo concreto, que es la forma de comunicarse entre dos dispositivos, y cada uno de estos protocolos se identifica con un puerto TCP o UDP que utiliza por defecto. Los servicios de red se ubican en la capa de Transporte del modelo TCP/IP. Los protocolos y puertos más comunes son el 20 y 21 (*File Transfer Protocol*), el 22 (*Secure Shell Protocol*), el 53 (*Domain Name Protocol*), el 80 (*Hypertext Transfer Protocol*), el 443 (*Secure Socket Layer/Transport Layer Secure*) y muchos otros.

2.5. Huella de Sistemas Operativos (*OS Fingerprinting*)

La detección de huella de Sistemas Operativos o *OS Fingerprinting* se refiere al conjunto de técnicas que basándose en parámetros de la comunicación TCP/IP son capaces de detectar el Sistema Operativo de un elemento de red, normalmente equipos finales. Los parámetros que se suelen observar son campos de TCP/IP como son el tamaño máximo de paquete, el TTL (*Time To Live*), el tamaño de ventana y otros parámetros. La técnica puede hacerse de forma activa usando herramientas como nmap, forzando la comunicación o de forma pasiva con herramientas como p0f, la que se usa en este trabajo. También, el análisis del User-Agent de la comunicación HTTP, SSH y FTP son otras de las fuentes de información del posible Sistema Operativo. En la siguiente tabla se detallan distintos valores de parámetros de la comunicación TCP/IP y su correspondiente Sistema Operativo:

Operating System (OS)	IP Initial TTL	TCP window size
Linux (kernel 2.4 and 2.6)	64	5840
Google's customized Linux	64	5720
FreeBSD	64	65535
Windows XP	128	65535
Windows 7, Vista and Server 2008	128	8192
Cisco Router (IOS 12.4)	255	4128

Tabla 2.1: Parámetros usados para detectar el Sistema Operativo y sus valores, obtenida de la web de NETRESEC [4]

2.6. Huella JA3

JA3 es un método para identificar el tipo de aplicación o usuario que está realizando una conexión a través del protocolo SSL/TLS. Este protocolo, por definición, está encriptado y mantiene la confidencialidad de la información entre cliente y servidor. La comunicación a través de SSL se comienza con un denominado *SSL Handshake*, en el que el cliente y el servidor negocian los parámetros de la comunicación como la versión de SSL, el cifrado, la curva elíptica utilizada y otros parámetros. Al transmitirse esta negociación en claro y ser la única parte del protocolo que no está encriptada, es posible identificar un tipo de cliente basándose únicamente en estos parámetros. Por

simplicidad, se usa el hash MD5 de los parámetros y de esta manera se identifica de manera unívoca a un tipo de usuario. Este método se ha estandarizado [19] y se utiliza principalmente en la búsqueda y detección de programas maliciosos que intentan utilizar SSL para ocultarse. Este estándar está basado en la investigación y posterior herramienta TLSFingerprint creada por Lee Brotherston [13].

Capítulo 3

Análisis de Zeek

3.1. Introducción y arquitectura de Zeek

Zeek [9] es una herramienta de código abierto que se utiliza para monitorización de la seguridad de una red, lo que se denomina Sistema de detección de intrusos o *Intrusion Detection System (IDS)*, aunque puede servir para más aplicaciones como la que se ha usado en este proyecto. Zeek está desarrollado principalmente por programadores de la empresa Corelight, así como por investigadores de Universidades como la de Indiana, Berkeley y otras. Además, al ser de código abierto, ha desarrollado una comunidad alrededor en la que los usuarios crean y añaden funcionalidades y módulos para mejorar sus detecciones y rendimiento. Esta herramienta era conocida anteriormente como Bro y lleva en continuo desarrollo desde 1994.

Zeek está programada en C++ y está disponible en diferentes sistemas operativos como macOS, sistemas basados en GNU/Linux y FreeBSD. La arquitectura de Zeek está basado en 3 componentes básicos que interactúan entre ellos. Los tres elementos son la Red (*Network*), el Motor de eventos (*Event Engine*) y el Intérprete de reglas programadas (*Policy Script Interpreter*).

- **Red:** La red o *network* en la arquitectura de Zeek se refiere a un flujo de paquetes de red proveniente de una interfaz o tarjeta de red de un dispositivo que esté capturando dicho tráfico. Zeek también es capaz de leer flujos de paquetes provenientes de una captura de tráfico anterior, realizada con la herramienta Wireshark o tcpdump, con extensión de archivo .pcap. Estos paquetes se introducen en el siguiente elemento de la arquitectura, el motor de eventos. Una de las formas más comunes de usar Zeek es a través de un puerto espejo o SPAN que capture todo el tráfico que va a través de un segmento de red.
- **Motor de eventos:** El motor de eventos o *Event Engine* en la arquitectura de Zeek reduce los paquetes a eventos de alto nivel. Estos eventos simplifican la

manera de ver el tráfico de red ya que extraen la parte fundamental de estos, reflejando qué han visto, pero no el por qué. Por ejemplo, una comunicación sobre SSH va a disparar diferentes eventos como son *ssh_auth_attempted* (evento de intento de autenticación), *ssh_auth_successful* (evento de autenticación correcta), *ssh_encrypted_packet* (evento que se dispara por cada paquete cifrado), y muchos otros. Estos eventos con la información recolectada se envían al Intérprete de reglas programadas, el encargado de interpretar el por qué de los eventos y correlazarlos entre ellos.

- **Intérprete de reglas programadas:** El intérprete de reglas programadas o *Policy Script Interpreter* es el cerebro de Zeek, el encargado de ejecutar notificaciones o escribir en los registros (*logs*) de los eventos que han sucedido. Este intérprete puede ser programado en un lenguaje propio de Zeek, basado en C++ para que ejecute programas externos y efectúe una respuesta inmediata ante un evento anómalo, además de ser capaz de guardar todo lo que ocurra.

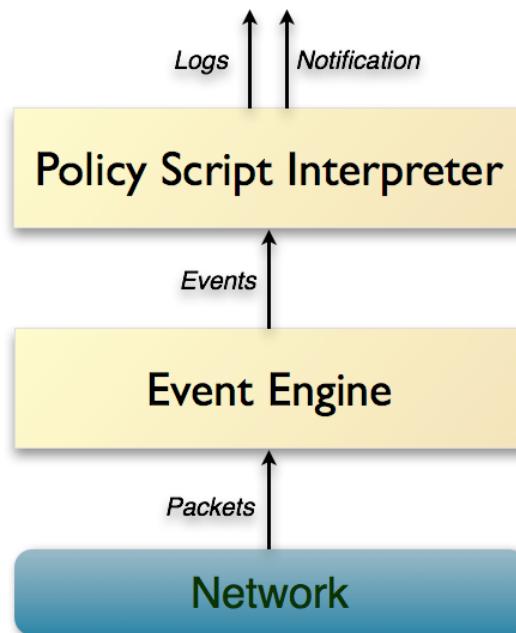


Figura 3.1: Arquitectura de Zeek [9]

La arquitectura y flujo de Zeek es más compleja que la que se representa en la Figura 3.1, ya que el Motor de eventos es un gestor muy complejo, además de haber un elemento llamado BiF (Built-in Functions) que ayuda a la programación de la salida de Zeek y de algoritmos de tipo general como manejo de cadenas, funciones matemáticas, filtros de paquetes, comunicación de procesos y más. Esta arquitectura está explicada con más detalle en la charla “Following the Packets: A Walk Through Bro’s Internal Processing Pipeline” realizada por Robin Sommer [21]. En la Figura 3.2 se representa en detalle el flujo de información dentro de Zeek.

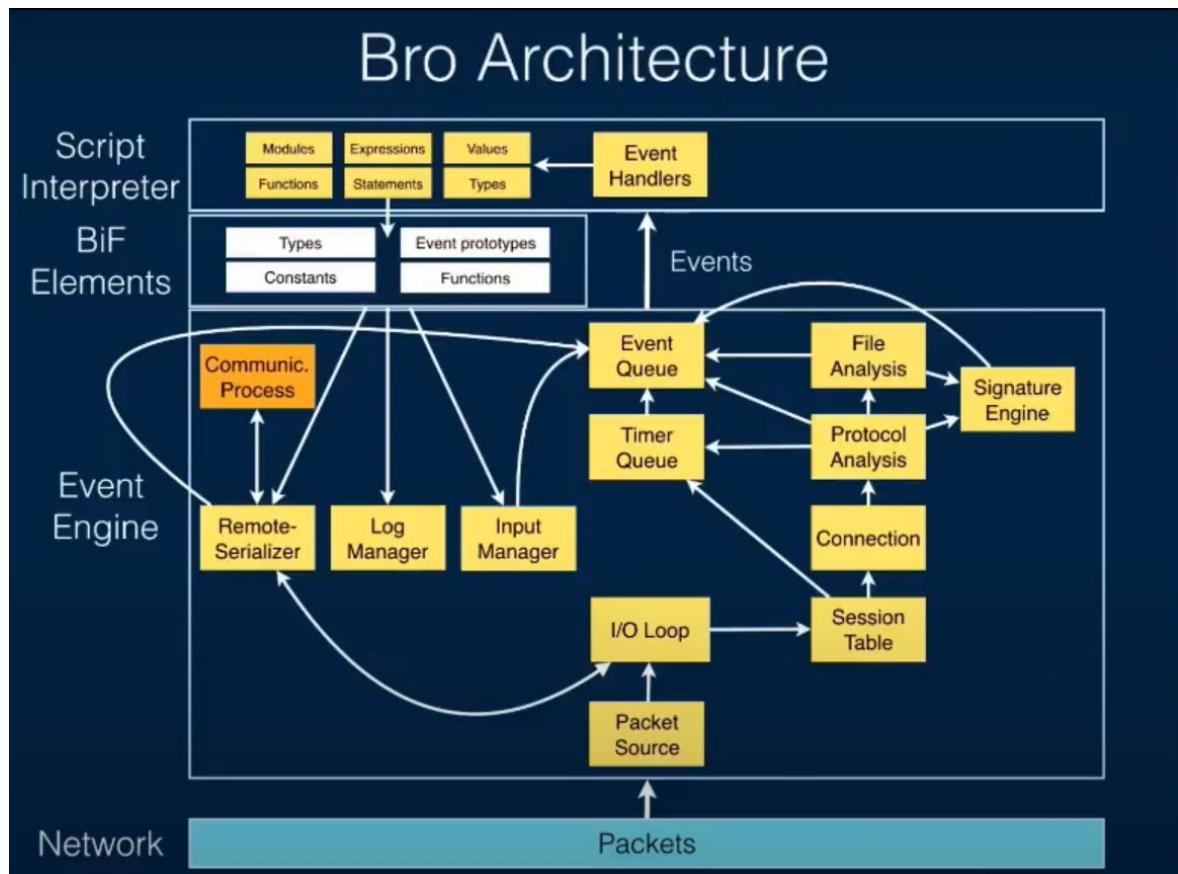


Figura 3.2: Arquitectura detallada de Zeek [21]

3.2. Programación en Zeek. Fichero local.zeek

Como ya se ha comentado en la sección 3.1 Zeek tiene su propio lenguaje de programación basado en C/C++ y que comparte varias características con este. Una de las funciones básicas de Zeek son las declaraciones de librerías externas, las cuales se hacen usando “@load” e indicando después la librería a utilizar. En esta directiva, se añade la ruta del script a cargar desde la librería base de Zeek. Una vez añadida, ya puedes utilizar los objetos y funciones que haya dentro de ella.

Dentro de los ficheros de Zeek, hay un fichero que no se sobrescribe cuando hay actualizaciones y mejoras de Zeek denominado “fichero de personalización de sitio local”. Este fichero, indica los scripts cargados en Zeek que se van a usar para el análisis, además de las redes locales que se quieren analizar usando la directiva “Site::local_nets”. En el caso de la herramienta creada para el TFG, no se utiliza esta directiva ya que se presupone que se está capturando en un segmento de red adecuado y que no hace falta definir tus redes locales, ya que puedes buscar en el informe posteriormente. Sin embargo, hay varias directivas “@load” añadidas, que no están por defecto en el fichero local.zeek como son:

- **tuning/json-logs:** Script utilizado para guardar todos los logs como archivos JSON, en una única lista. Se utiliza para facilitar el posterior manejo de los datos.
- **tuning/track-all-assets:** Script utilizado para monitorizar y registrar los servicios, software y los nombres de host que puedan tener los activos analizados.
- **./ja3:** Script utilizado para generar la huella JA3 de los User-Agents que se encuentren en la captura de tráfico analizada.

3.3. Ficheros log generados por Zeek

Zeek genera múltiples ficheros en formato *.log*, indicando que son ficheros de registro de información. En estos ficheros se indican distintos aspectos de las comunicaciones y eventos que se han extraído gracias al motor de eventos.

Dentro de los propios ficheros log de Zeek hay distintas categorías, ya que no solo se puede usar Zeek para analizar los protocolos, sino que se puede utilizar para extraer más información que enriquezca la visibilidad de la red como pueden ser los ficheros que se han enviado, los ejecutables, los hosts identificados, el software que corre en un equipo, estadísticas del tráfico, etc. Los logs que genera Zeek se pueden dividir en siete categorías diferenciadas:

- **Protocolos de red:** Zeek analiza distintos protocolos de red separándolos por tipo. Para cada uno de estos, se genera un archivo log distinto, teniendo así ficheros de comunicación TCP/UDP e ICMP (*conn.log*), fichero para el protocolo HTTP (*http.log*), fichero para el protocolo SSH (*ssh.log*), y muchos otros.
- **Archivos:** Tipos de logs asociados a ficheros. Se realiza desde análisis de los metadatos de los ficheros, de ejecutables y de certificados enviados a través de la red monitorizada.
- **NetControl:** Son los ficheros asociados a NetControl, el framework de Zeek para gestionar diferentes dispositivos al mismo tiempo.
- **Detección:** Son los ficheros relacionados con el framework de inteligencia asociado a Zeek, de alarmas configuradas y de detección del uso de traceroute.
- **Observaciones de red:** Son los ficheros asociados a los certificados SSL, a la identificación de hosts, uso de servicios conocidos, etc.
- **Miscelánea:** Son los ficheros relacionados con las anomalías de estadísticas y de red, además de la integración con otras herramientas como Snort.
- **Estado de Zeek:** Son los ficheros asociados al propio sistema de Zeek. Muestra la lista de logs que se deben generar, los mensajes de información o advertencia sobre el sistema, la pérdida de paquetes, etc.

Capítulo 4

Diseño y desarrollo de la herramienta

4.1. Diseño y programación de la herramienta principal

4.1.1. Recolección de datos y análisis

La herramienta de detección de activos, la pieza principal del TFG, se ha llamado “Active-mapper” y será denominada así a lo largo de los próximos capítulos. Esta herramienta, tiene como objetivo la creación automática de una CMDB a partir de un archivo de captura de tráfico, es decir, de forma pasiva (no intrusiva) en la red. Para ello se usa un flujo de información que se detalla en la Figura 4.1

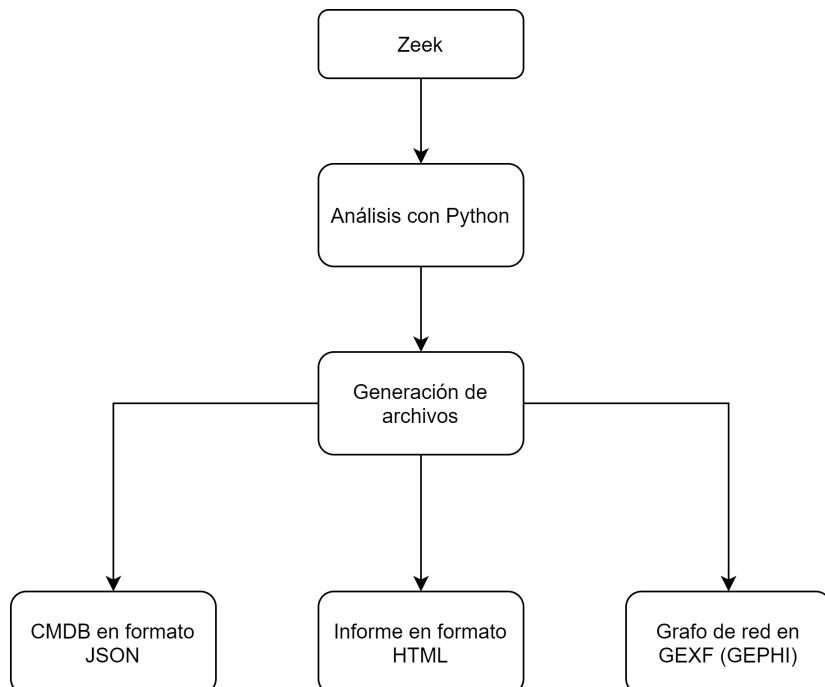


Figura 4.1: Diagrama de flujo de la información

Para definir cada uno de los activos de la CMDB (CIs), se utiliza un JSON Schema que guarda en un formato concreto la IP del dispositivo, el Sistema Operativo, los servicios de red, el Software corriendo en el equipo, la huella JA3 del dispositivo y el posible User-Agent del dispositivo, basándonos en la huella JA3 y utilizando la base de datos de ja3er [2].

El JSON Schema utilizado define que la IP es el único atributo obligatorio para cada activo, y que el resto de atributos pueden ser un array, ya que se puede descubrir más de un servicio o software en un mismo dispositivo. El JSON Schema para el proyecto es el siguiente:

Listing 4.1: JSON Schema de un activo

```
1  {
2    "type" : "object",
3    "properties":{
4      "IP": {"type" : "string"},
5      "OS": {"type": ["string", "array"]},
6      "Services": {"type": ["string", "array"]},
7      "Software": {"type": ["string", "array"]},
8      "JA3_Fingerprint": {"type": ["string", "array"]},
9      "Possible User-Agent": {"type": ["string", "array"]}
10     },
11    "required": ["IP"]
12 }
```

A la hora de que la herramienta genere el activo usando el esquema, se crea un diagrama de flujo partiendo de distintos ficheros log generados por Zeek, y poblando los campos del CI que luego se agrega a la CMDB. Los archivos log de los que se extrae la información para generar cada activo son los siguientes:

- **conn.log:** El fichero conn.log almacena todas las conexiones TCP, UDP e ICMP que se han llevado a cabo en la red. Además, guarda las IP, puertos origen y destino y otras muchas características de la comunicación. Es uno de los ficheros esenciales del análisis del tráfico. En nuestro caso, se utiliza para almacenar una lista de todas las direcciones IP que han intercambiado información en nuestra red monitorizada.
- **known-services.log:** El fichero known-services.log almacena el protocolo detectado en una sesión que ha establecido un intercambio de datos a través de TCP, además de las IP y puertos utilizados entre origen y destino. En nuestra herramienta, se utiliza para detectar los servicios de red asociados a una dirección IP concreta.

- **software.log:** El fichero software.log guarda una lista de software asociado con una dirección IP. Puede detectar por ejemplo el tipo de servidor HTTP o SSH corriendo en una máquina, gracias a los datos intercambiados a través de la red. En nuestro caso, almacena dicho software en una lista.
- **p0f.log:** El fichero p0f.log es el único que no se genera usando Zeek, ya que se genera usando la herramienta p0f. En este fichero, se almacena el sistema operativo que un dispositivo está usando gracias a técnicas de OS Fingerprinting (ver Tabla 2.1).
- **ja3.log:** El fichero ja3.log almacena una lista de huellas JA3 detectadas en los intercambios de SSL dentro de nuestra red. En nuestro caso, se utiliza para descubrir el posible User-Agent de los equipos de la red.

La herramienta se ha programado de la manera más eficiente posible, evitando las listas (arrays) de Python en todas las ocasiones que se ha podido, de manera que se usen diccionarios, mucho más rápidos en cuanto a la realización de búsquedas, que es la principal operación que se realiza en la herramienta. Esto sucede porque los diccionarios de Python usan tablas de hashes, de manera que almacenan elementos sin ordenar por índice, sino que no siguen una ordenación y usan estructuras del tipo clave:valor, en el que la clave es un identificador único para cada valor. En Active-Mapper, la clave es la dirección IP, ya que es un valor que identifica de manera única a un activo, y el valor va a ser el propio objeto del activo en Python. De esta manera, la complejidad temporal de una búsqueda no será $O(n)$, sino $O(1)$, permitiendo que el tiempo de ejecución del programa sea mucho menor [6].

A la hora de hacer la lectura de los ficheros, el primero que se lee es el fichero conn.log, del que se extraen las direcciones IP y se crea el diccionario de IPs y objetos. Al estar todas las comunicaciones realizadas en la red en el fichero conn.log, nos aseguramos que en el resto de ficheros la clave de cada una de las direcciones IP va a existir, y por tanto se van a realizar las búsquedas correctamente en la lectura de todos los ficheros. Además del diccionario de IP y objeto de activo, se crea un segundo diccionario en el que se anota como clave el hash JA3 y como valor los User-Agents asociados a este hash. Esto permitirá más adelante hacer las búsquedas de los User-Agents de manera mucho más rápida a como se realizaría haciendo peticiones HTTP a la página de JA3er.

Los pasos que realiza Python para analizar cualquier fichero de captura de tráfico son los siguientes:

1. Se lanza la herramienta desde ventana de comandos.

2. Figlet muestra el nombre de la herramienta en la ventana de comandos.
3. Se pide al usuario que indique el directorio en que se encuentra el archivo de captura de tráfico, si desea que se realice el grafo de red y si desea validar el JSON siguiendo el JSON Schema definido en el Listing 4.1.
4. Se lanza Zeek y p0f para generar los archivos que posteriormente se van a procesar.
5. Se crea el diccionario de hashJA3:User-Agents usando la base de datos actualizada de JA3er.
6. Se lee el fichero conn.log y se genera el diccionario de IP:Activo, con los campos del activo vacíos.
7. Se lee el fichero known-services.log y se añaden a los activos correspondientes los servicios de red encontrados.
8. Se lee el fichero software.log y se añaden a los activos correspondientes los programas y versiones encontrados. Además, si uno de los software contiene la palabra “Windows” o “Linux” lo anota como Sistema Operativo posible.
9. Se lee el fichero ssl.log y se guardan los hashes JA3 y JA3S para cada uno de los activos.
10. Se lee el fichero p0f.log y se añaden los Sistemas Operativos descubiertos a los activos correspondientes.
11. Eliminamos posibles duplicados en los campos de los activos usando la función unique que se ha creado previamente. Esta función utiliza sets de Python para no ralentizar el tiempo de ejecución.
12. Se hace la búsqueda de User-Agents en el diccionario de hashes JA3 y se guardan en cada activo correspondiente.
13. Se genera el JSON siguiendo el JSON Schema para cada uno de los activos y se valida y añade a una lista si se ha decidido así previamente. Si no se desea validar, se añade a la lista directamente.
14. Se escribe la lista en el archivo data.json y se copia el contenido a la raíz donde se encuentran los archivos de Flask para su posterior utilización.
15. Si se ha elegido crear el grafo de red, se ejecutan secuencialmente brassfork y CSV-toGEXF para generar el grafo y se guarda en el directorio inicial como out.gexf (ver Figura 4.6).

16. Por último, se ejecuta Flask junto con JSON2HTML para generar el informe de activos final basándose en el archivo data.json (ver Figura 4.4).

La información procesada y las acciones realizadas dentro de Python se pueden resumir en el siguiente diagrama de flujo:

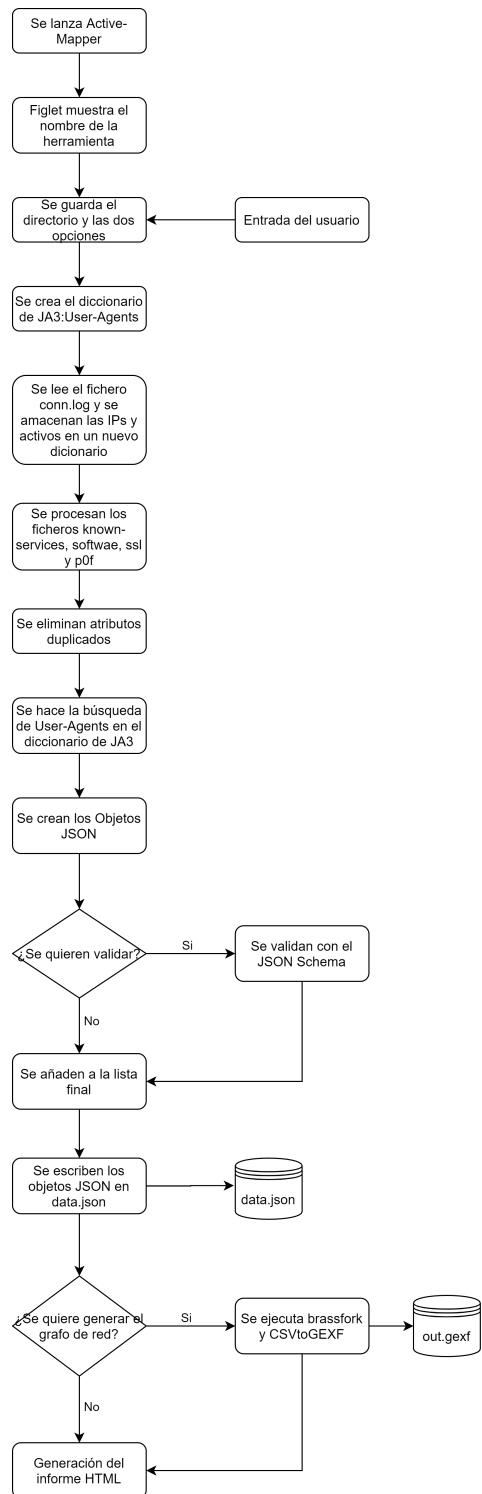


Figura 4.2: Diagrama de flujo del funcionamiento de Python

Para formar cada Configuration Item o activo de red, se puede resumir el diagrama 4.2 en la siguiente figura:

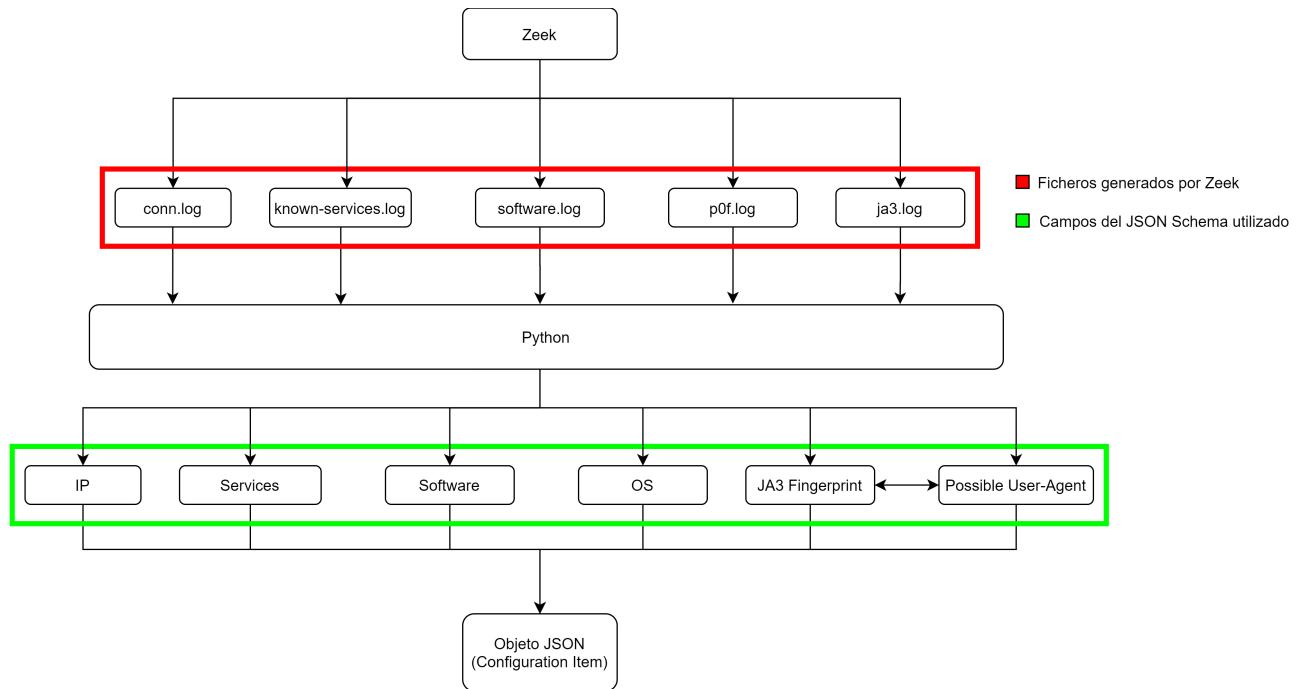


Figura 4.3: Diagrama de flujo de construcción de un activo

4.1.2. Generación del informe web

El informe es una de las piezas clave de la herramienta, ya que se necesita una interfaz que sea útil para el usuario a la hora de ver qué resultados ha obtenido gracias al análisis de la captura con Python. Este informe se genera en HTML (*HyperText Markup Language*), y usando como framework web Flask, framework basado en Python y adecuado para nuestro proyecto.

Para generar el informe, lo primero que se debe hacer es ejecutar Zeek y la herramienta de Python que genere el archivo “data.json”. Una vez generado, se generan dos rutas en Flask, una para el informe web y otra ruta para el propio JSON. A la hora de generar el archivo HTML con el informe reflejado, se utiliza la librería JSON2HTML, capaz de generar una tabla sencilla HTML a partir de un texto en formato JSON como es nuestra CMDB. Además de esto, se crea un pequeño script escrito en JavaScript con una función que sea capaz de realizar una búsqueda en la tabla y poder así filtrar por activo, basándonos en su IP, facilitando mucho el uso del informe.

El diagrama de flujo de la creación del informe se ve reflejado en la siguiente figura:

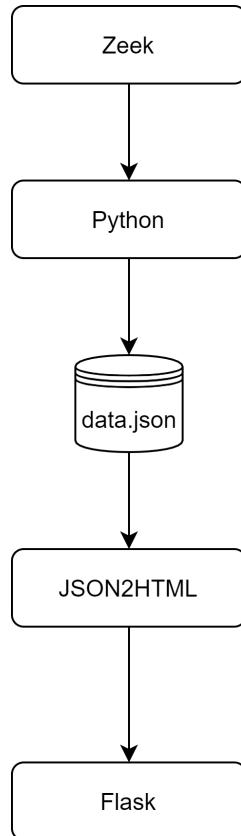


Figura 4.4: Diagrama de flujo de la creación de informes

La búsqueda que se realiza en el informe web se realiza basándose en el comienzo de la dirección IP. En la figura 4.5 se muestra un ejemplo de cómo se realiza esta búsqueda y los resultados que arroja.

The screenshot shows a web browser window with the URL 'localhost:5000' in the address bar. The page displays a table with the following data:

IP	OS	Services	Software	JA3_Fingerprint	Possible User-Agent
10.11.11.200	• Windows 7 or 8		• Mozilla/5.0 (Windows NT 6.1 WOW64 Trident/7.0 rv:11.0) like Gecko	• 4d7a28d6f2263ed61de88ca66eb011e3	<ul style="list-style-type: none"> • Mozilla/5.0 (Windows NT 6.1 WOW64 Trident/7.0 rv:11.0) like Gecko • Mozilla/5.0 (compatible MSIE 10.0 Windows NT 6.1 Trident/6.0 MATMJS) • Mozilla/5.0 (compatible MSIE 10.0 Windows NT 6.1 Trident/6.0 Touch MALNJS)

Figura 4.5: Funcionamiento de la búsqueda por IP

Los campos del activo aparecen en las columnas, y dependiendo de si es un string o un array aparecen como una lista con puntos en cada uno de los atributos de la lista.

4.1.3. Generación del grafo de red

El grafo de red basado en la captura de tráfico se realiza utilizando dos herramientas de terceros llamadas Brassfork y CSVtoGEXF.

- **Brassfork:** Brassfork es una herramienta de código abierto desarrollada en lenguaje de programación Go, que es capaz de obtener dos archivos .csv (*Comma sepparated values* o Archivo separado por comas) de nodos y enlaces a partir de un archivo de captura de tráfico en formato pcap. Brassfork guarda el protocolo, número de bytes enviados, el origen y destino, el tipo de paquete TCP (SYN, ACK, Datos, etc) y muchos otros detalles de las comunicaciones. Brassfork no extrae ninguna información adicional, tan solo refleja el pcap y las comunicaciones en los archivos csv. Utiliza la librería GoPackets de Google y formato JSON para manejar la información.
- **CSVtoGEXF:** CSVtoGEXF es una herramienta de código abierto, que como su nombre indica, convierte dos archivos csv de nodos y enlaces en un archivo GEXF (*Graph Exchange XML Format*), archivo estándar para representación de grafos con la herramienta GEPHI [11]. En el Anexo D de este trabajo se hace una guía para la utilización de GEPHI para extraer información de red. Esta herramienta debe utilizar, además de los dos archivos csv, un tercer archivo de texto de definiciones, que describa el cómo está representada la información de dichos csv.

En la figura 4.6 se representa el diagrama de flujo de la creación del grafo basado en la captura de tráfico.

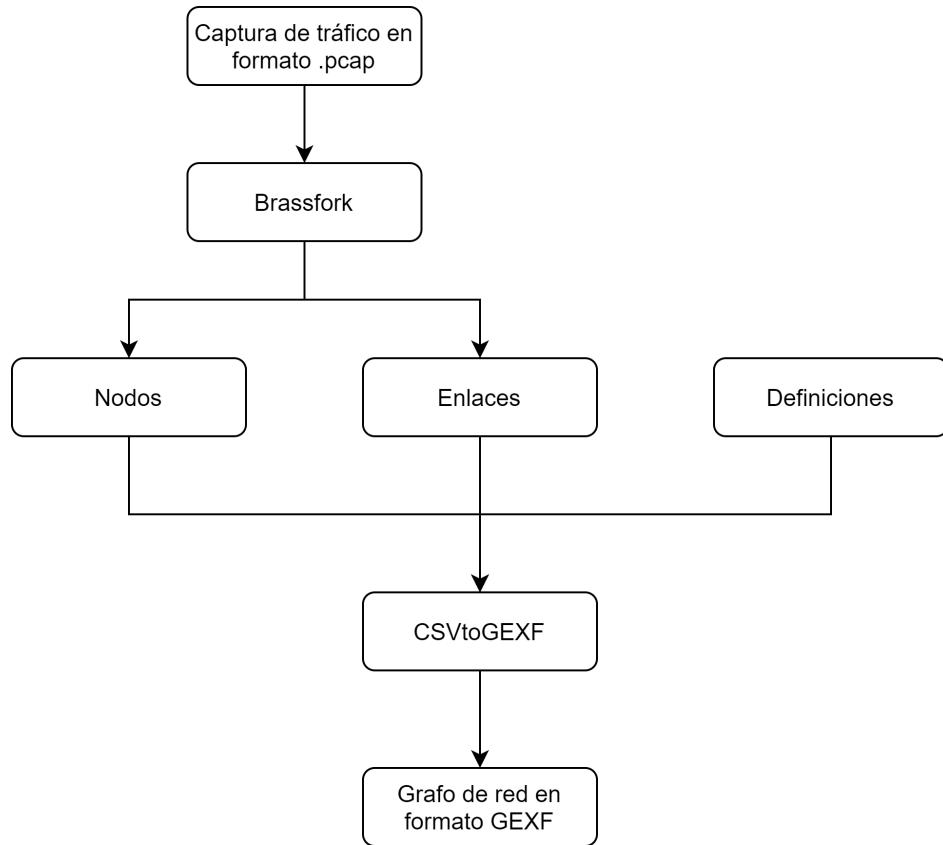


Figura 4.6: Diagrama de flujo de la creación del grafo de red

4.2. Desarrollo del script de instalación

Uno de los objetivos del proyecto es que pueda ser utilizado como herramienta para las empresas que quieran mejorar sus CMDB a través del análisis de tráfico. Para que la herramienta sea accesible, se ha creado un script de instalación que automáticamente instala todas las dependencias y programas necesarios, así como ejecuta procesos para configurar todo el sistema antes de que pueda ser utilizado. Las librerías y programas necesarios en el proyecto se detallan en capítulo 1.3.

El script de instalación está desarrollado para sistemas basados en Ubuntu, ya que se instala Zeek para Ubuntu. Si se quiere utilizar en otros sistemas basados en GNU/Linux, se debe instalar Zeek manualmente previamente a ejecutar el instalador. El instalador está escrito en lenguaje Bash, el lenguaje de comandos para shell de Linux, para que sea instalable en dichos sistemas. El diagrama de flujo del funcionamiento del instalador se refleja en la siguiente figura:

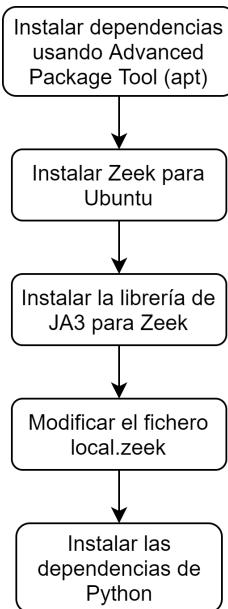


Figura 4.7: Diagrama de flujo de la instalación

4.3. Uso de Docker. Desarrollo del archivo Dockerfile

Docker es un proyecto de código abierto que se utiliza para abstraer la aplicación del sistema operativo anfitrión. Esto quiere decir que Docker funcionará como un entorno de virtualización independiente del sistema operativo sobre el que esté funcionando.

Esta característica es interesante para nuestro proyecto, ya que tiene la limitación actual de poder instalarse únicamente en el Sistema Operativo Ubuntu, con lo cual otros Sistemas Operativos como puede ser Windows, Mac OS y otros basados en Linux no podrían ejecutar la herramienta. Añadiendo Docker al proyecto, se podría ejecutar Active-Mapper en cualquier sistema operativo, ya que estaría funcionando sobre Docker, y dicho Docker estaría basado en Ubuntu 18.04. Esto, se podría comparar con una “pequeña máquina virtual”, al que llamaremos contenedor, cuyo único objetivo es ser capaz de ejecutar la herramienta que hemos desarrollado (ver Figura 4.8).

Para crear la imagen de Docker se crea un archivo Dockerfile, indicando las instrucciones previas al lanzamiento del contenedor para que todo funcione correctamente. Estas instrucciones incluyen la imagen base que se va a utilizar para el contenedor (en nuestro caso Ubuntu 18.04), la creación de los usuarios, la copia de archivos necesarios para el funcionamiento de Active-Mapper y por último el puerto y comando inicial que se va a ejecutar en el contenedor. Dicho Dockerfile está detallado en el Anexo E.

Una vez se tiene la imagen del contenedor con Active-Mapper instalado, se debe lanzar el contenedor con la opción “-it”, indicando que quieres establecer una sesión

interactiva con el contenedor, y permitiendo la ejecución de la herramienta dentro de este.

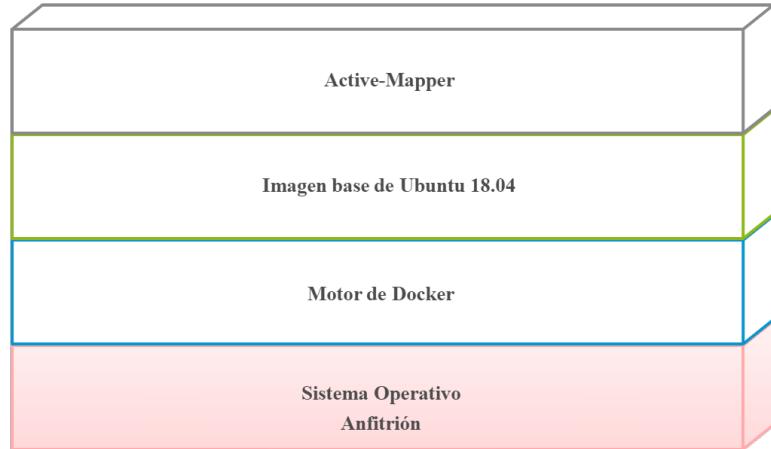


Figura 4.8: Arquitectura de Active-Mapper sobre Docker

Capítulo 5

Evaluación y mejoras

5.1. Efectividad

5.1.1. Evaluación utilizando dataset

A la hora de evaluar la herramienta, se ha procurado comprobar que los resultados obtenidos son correctos, y que lo detectado en los activos, es decir, su Sistema Operativo, el Software, el User-Agent y la dirección IP son correctos. Para ello, se han ido realizando a lo largo del desarrollo del trabajo distintas pruebas usando capturas de tráfico de la web de NETRESEC [5], en las que hay mucho tráfico de distinto tipo e incluso muestras de malware.

Para comprobar si funciona la herramienta correctamente, se ha utilizado NetworkMiner [3], una herramienta profesional de análisis de tráfico que es capaz de extraer, a partir de una captura de tráfico, las direcciones IP de la red, el sistema operativo, la huella JA3 y el posible User-Agent. La diferencia con nuestra herramienta es que esta tiene una GUI (*Graphical User Interface o Interfaz Gráfica de Usuario*), y que para exportar los datos a formato JSON es necesario comprar una licencia de dicho software.

Las capturas de tráfico que se van a utilizar para generar un dataset están extraídas de ejercicios de análisis de tráfico de la página web de Malware Traffic Analysis [7]. Estas capturas son muy diversas, algunas con varios ordenadores, con móviles, sistemas operativos de todo tipo, servidores, servicios muy diversos, etc. y en todas las capturas hay un mínimo de 2 dispositivos y un máximo de 8. El dataset que se ha creado es de 20 capturas de tráfico de las que se ha evaluado la red privada (lo que equivaldría a la red empresarial o red monitorizada) y la detección de los activos de esta con respecto a NetworkMiner. Los porcentajes que aparecen en la tabla son con respecto al total de activos dentro de la red monitorizada, ya que queremos comprobar el grado de fiabilidad de ambas herramientas.

Captura	NetworkMiner						Active-Mapper					
	IP	Sistema Operativo	Servicios	Software	JA3	User-Agents	IP	Sistema Operativo	Servicios	Software	JA3	User-Agents
C1	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %	67 %	100 %	67 %	100 %
C2	100 %	67 %	100 %	80 %	46 %	100 %	67 %	100 %	67 %	100 %	75 %	100 %
C3	100 %	50 %	100 %	100 %	67 %	100 %	50 %	100 %	50 %	100 %	50 %	100 %
C4	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %	67 %	100 %
C5	100 %	100 %	100 %	100 %	100 %	83 %	100 %	100 %	100 %	100 %	67 %	100 %
C6	100 %	100 %	100 %	100 %	100 %	70 %	100 %	100 %	100 %	100 %	80 %	100 %
C7	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %	50 %	100 %	100 %	100 %
C8	100 %	100 %	100 %	100 %	100 %	67 %	100 %	100 %	100 %	100 %	75 %	100 %
C9	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %	50 %	100 %	67 %	100 %
C10	100 %	100 %	100 %	100 %	100 %	60 %	100 %	100 %	100 %	100 %	100 %	100 %
C11	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %	50 %	100 %	100 %	100 %
C12	100 %	100 %	100 %	100 %	100 %	83 %	100 %	100 %	100 %	100 %	67 %	100 %
C13	100 %	100 %	100 %	100 %	100 %	50 %	100 %	100 %	100 %	100 %	100 %	100 %
C14	100 %	67 %	100 %	83 %	100 %	57 %	100 %	100 %	67 %	100 %	100 %	100 %
C15	100 %	62,50 %	100 %	94 %	41 %	100 %	75 %	100 %	75 %	100 %	67 %	100 %
C16	100 %	100 %	100 %	100 %	100 %	83 %	100 %	100 %	100 %	100 %	75 %	100 %
C17	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %	70 %	100 %
C18	100 %	100 %	100 %	100 %	100 %	36 %	100 %	100 %	50 %	100 %	40 %	100 %
C19	100 %	100 %	100 %	100 %	100 %	23 %	100 %	100 %	50 %	100 %	100 %	100 %
C20	100 %	57 %	100 %	100 %	100 %	28 %	100 %	100 %	43 %	100 %	87,50 %	100 %
MEDIAS	100 %	90 %	100 %	99 %	97 %	70 %	100 %	100 %	76 %	100 %	78 %	100 %

Tabla 5.1: Tabla comparativa de resultados entre la herramienta NetworkMiner y la herramienta Active-Mapper

En la Tabla 5.1 se pueden ver los resultados de la herramienta NetworkMiner junto a la herramienta desarrollada en este TFG, Active-Mapper. De esta tabla se pueden sacar varias conclusiones:

1. Tanto las direcciones IP como los Servicios de red y los hashes JA3 se realizan y almacenan de forma correcta en casi el 100% de las ocasiones en ambas herramientas.
2. La técnica para detectar el Sistema Operativo es más precisa en NetworkMiner que en Active-Mapper (+14%). Esto ocurre porque NetworkMiner utiliza p0f junto con Satori y Ettercap, dos herramientas adicionales de Passive OS Fingerprinting, para dar una aproximación más fiable acerca del Sistema Operativo.
3. El Software que detecta NetworkMiner es más preciso (+21% con respecto a Active-Mapper) ya que guarda todos los User-Agents y software que encuentra, por ejemplo software de RDP y de autenticación con Kerberos. Active-Mapper podría detectarlo, pero no es suficiente con el archivo software.log, sino que habría que añadirlo manualmente con otros logs adicionales de Zeek.
4. La herramienta Active-Mapper extrae los User-Agents de los JA3 usando la base de datos de ja3er [2], una fuente de información mucho más extensa que la que posee NetworkMiner, que es una base de datos fija que no se actualiza automáticamente ni varía hasta que hay una nueva actualización de software. Esto hace que Active-Mapper sea capaz de predecir un 30% más de User-Agents que NetworkMiner.

5.1.2. Evaluación en una red real

Tras evaluarlo con un dataset, se va a comprobar la fiabilidad de Active-Mapper con una red real. Para ello se ha utilizado una captura de tráfico del Cyber Research Center de West Point, la academia militar de Estados Unidos (USMA). Las capturas de tráfico y el escenario de red fueron creados por la National Security Agency (NSA) para la competición CDX de 2009 [1]. A diferencia de otros datasets, estos 12GB de capturas que proporciona la NSA, vienen junto a un escenario de red con distintos aspectos detallados sobre esta y sus equipos, de los que se van a comprobar cuántos de ellos han sido identificados correctamente por Active-Mapper.

El tamaño total de capturas que se ha analizado es de 500 MB, capturado durante 10 horas ininterrumpidas. Se han detectado un total de más de 30.000 direcciones IP únicas en la captura de tráfico y el resultado de activos identificados se puede comprobar en la Figura 5.1.

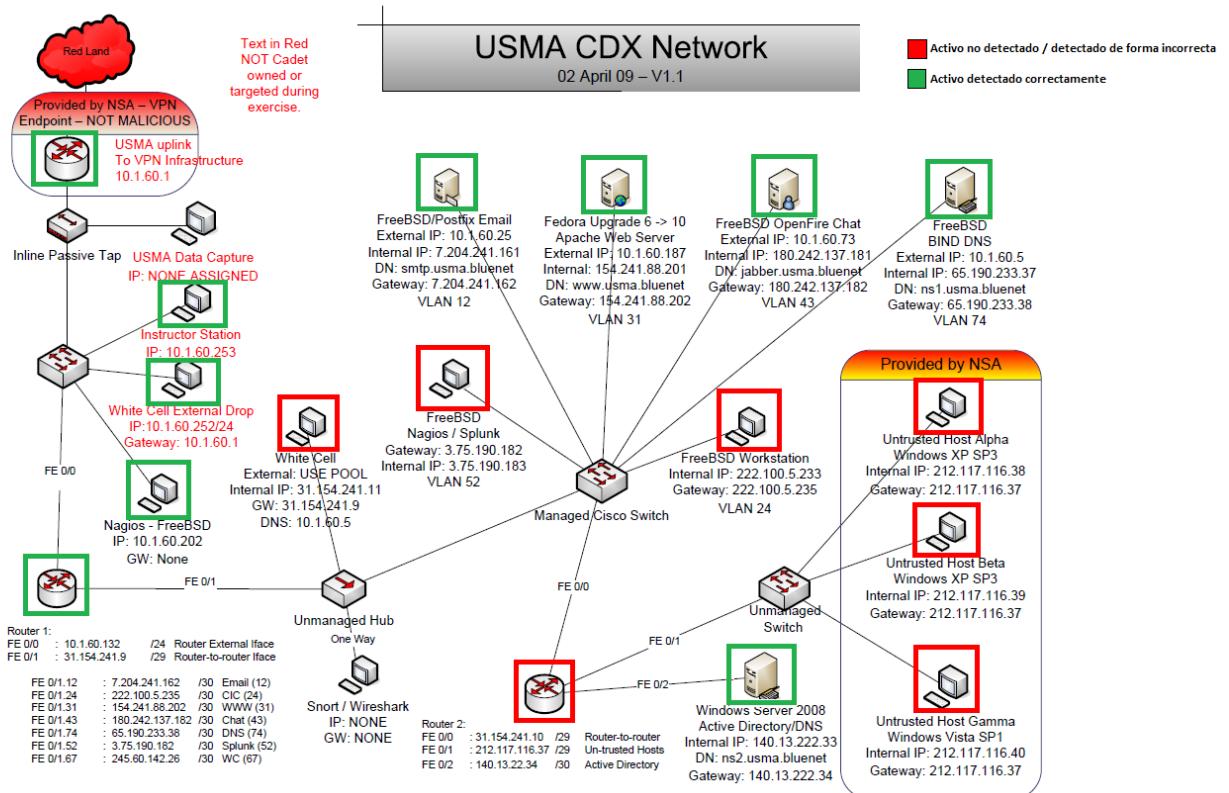


Figura 5.1: Resultados de la red analizada de USMA

De estos resultados se pueden extraer dos conclusiones:

1. De los 17 activos que tienen dirección de red en la Figura 5.1 se han detectado correctamente 10 de ellos, lo que supone un 59 % de efectividad de la herramienta en esta red concreta.
2. Estos datos reflejan dos problemas de la herramienta y de la detección pasiva de red en general: Depende de dónde captures en la red, vas a tener mayor o menor visibilidad de esta, y si un activo de red no genera tráfico, a todos los efectos es como si no existiera. Es por ello que se debe combinar con una CMDB ya creada anteriormente o con herramientas de escaneo activo que interfieran lo menos posible en la red. Además, es buena práctica capturar en varios segmentos de red al mismo tiempo, para aumentar la visibilidad.

5.2. Rendimiento

Las pruebas de rendimiento de Active-Mapper se han realizado en un Docker lanzado sobre un ordenador con Sistema Operativo Windows 10, con 8 GB de RAM y un procesador Intel i5-7300HQ a 3,00 GHz, que posee 4 núcleos y 16 hilos de ejecución, de los cuales solo se utiliza uno.

Para hacer una estimación de la velocidad de la herramienta se ha tomado una de las capturas de tráfico de 100MB de tamaño de la red de USMA [1] y se ha dividido en distintas capturas de menor tamaño. Se ha decidido dividir con un paso de 3 MB para tener una buena aproximación a la hora de realizar gráficas. Además, se han evaluado de forma separada el rendimiento de Zeek, p0f y de Active-Mapper, ya que p0f y Zeek son herramientas que no dependen de nuestra programación, sino que están implementadas previamente, y nosotros no las modificamos. También se excluyen de las gráficas el uso de brassfork y de CSVtoGEXF ya que son herramientas que tienen que procesar muchos datos y son bastante lentas, además de opcionales de usar cuando se usa Active-Mapper.

Dentro de Active-Mapper, también se ha dividido el tiempo total en distintos tiempos, ya que cada uno de esos subprocessos tienen un tiempo de procesado distinto, tal y como se puede observar en la Figura 5.2. De esta gráfica se destaca que el tiempo que más crecimiento sufre con respecto al tamaño es la validación del JSON Schema. También llama la atención que el tiempo de escritura, de búsqueda de JA3, la creación del diccionario y la obtención de los atributos únicos de cada activo tienen un tiempo despreciable conforme aumenta el tamaño de ficheros y el número de activos.

También se ha decidido evaluar el tiempo total dedicado por Zeek, el tiempo dedicado por p0f y el tiempo total dedicado por Active-Mapper, para saber qué parte del tiempo total se destina a cada una de las tres herramientas. Esto se detalla en la Figura 5.3. En esta gráfica se puede observar que el crecimiento de Active-Mapper es prácticamente a la misma velocidad que Zeek, y que p0f escala mucho mejor que ambas herramientas. Sin embargo, si se ejecuta Active-Mapper sin la validación del JSON Schema, se puede observar que escala a la misma velocidad aproximada que p0f, lo que es un punto a favor en capturas de gran tamaño y con muchos activos.

Uno de los aspectos que merece la pena mencionar es pico de tiempos que se produce entre las capturas de 46 MB y 49 MB de tamaño. Esto, a priori, podría parecer que se ha llegado al punto de inflexión en el cual la herramienta deja de ser estable por motivos de memoria utilizada, pero realmente esto se produce porque en la captura de 46 MB se han detectado 74 activos y en la de 49 MB se han detectado 6315 activos, haciendo que añadir el número de activos nuevos sea mucho más costoso y que la validación del total

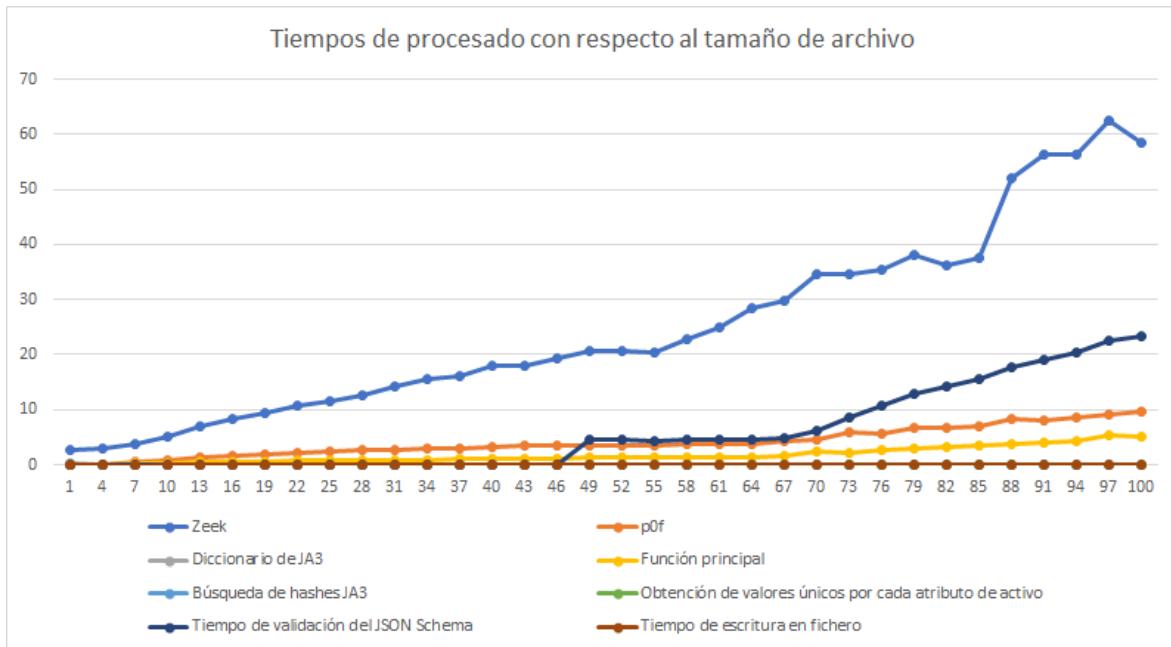


Figura 5.2: Gráfica de tiempos de cada uno de los procedimientos de Active-Mapper

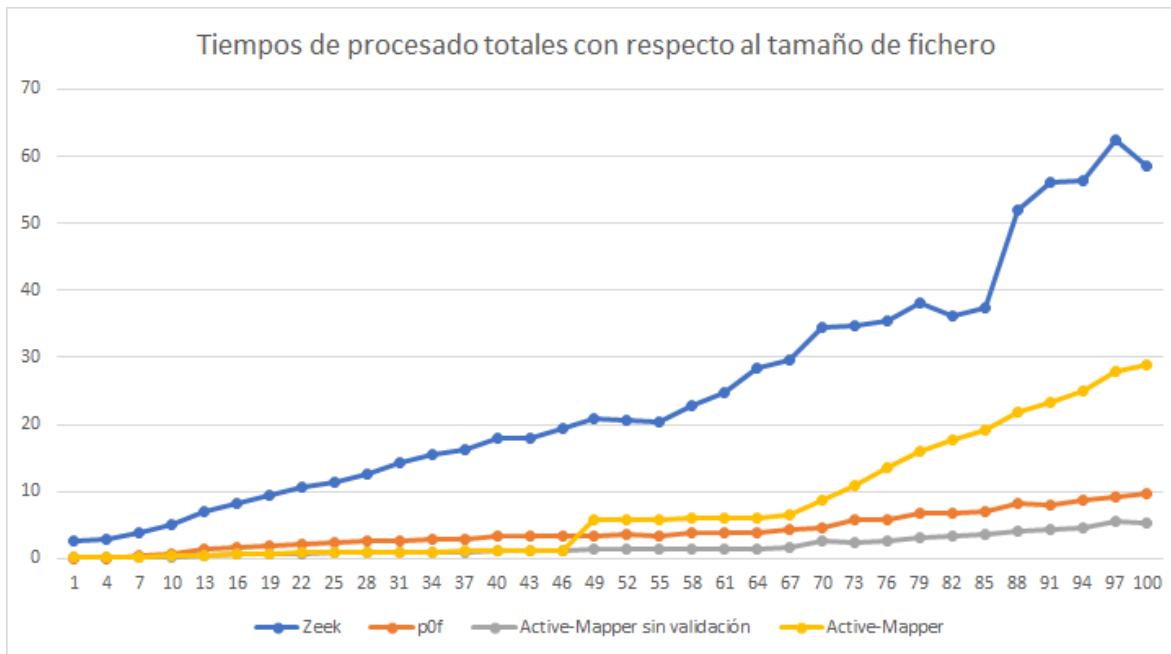


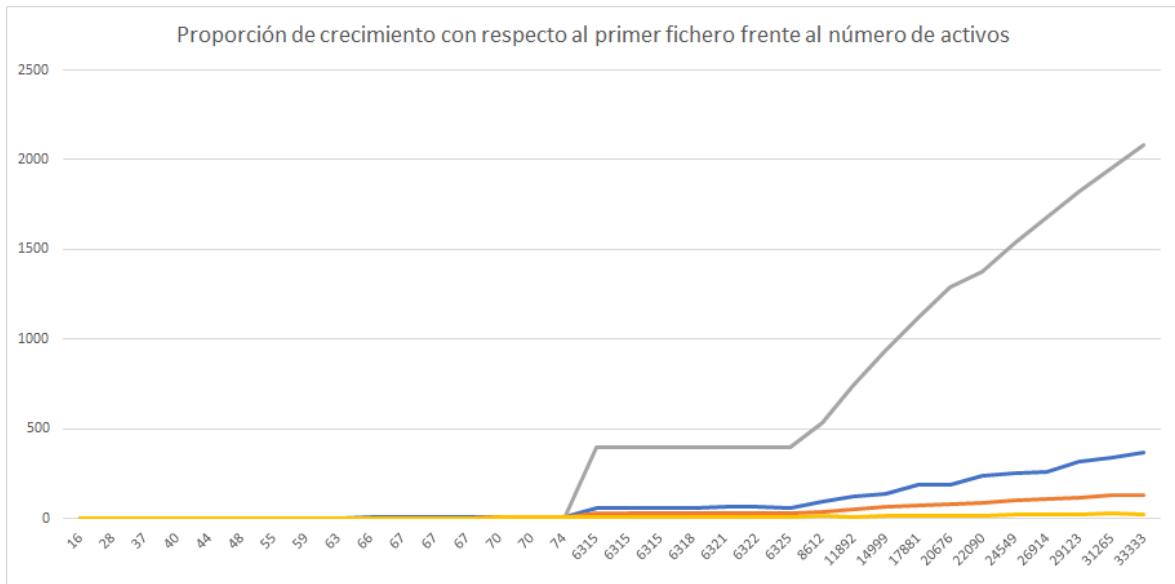
Figura 5.3: Gráfica de tiempos de Zeek, p0f y Active-Mapper con y sin validación del JSON Schema

de activos sea mucho más lenta, degradando la velocidad de la herramienta si se desea validar. En Active-Mapper, uno de los aspectos clave de rendimiento es por tanto el número de activos únicos detectados en la red. Por esto, también se ha querido analizar el rendimiento con respecto al número de activos, y no solo con respecto al tamaño del fichero. También destaca que el crecimiento de tiempos sin añadir la validación es mucho menor que con validación del JSON Schema, y que en ese caso, no se produce

el salto en tiempo, sino que es una curva creciente, pero no tan pronunciada, indicando que la validación escala mucho peor que las propias búsquedas. Para analizar esto, se han comprobado distintos parámetros como el tiempo medio de procesado por activo (ver Figura 5.4) y la proporción de crecimiento con respecto a la primera captura analizada (ver Figura 5.5).



Figura 5.4: Tiempo de procesado medio por cada activo frente al número de activos en la captura



que conforme aumente el número de activos, el tiempo de procesado para cada uno de ellos va a ser menor. Esto indica que la herramienta es más rápida por activo conforme aumenta el número de estos, como se puede ver en la Figura 5.4. Además, el crecimiento temporal es mucho menos pronunciado con la última versión del código, la cual sustituye las búsquedas en Python usando índices por búsquedas en diccionario, un proceso que es mucho más rápido a medida que el tamaño de la lista o diccionario aumenta, tal y como se define en la complejidad temporal en la documentación de Python [6]. Además, se puede observar que el tiempo crece en una mucha menor proporción si no se valida el JSON Schema.

Es por esto, que la herramienta es buena tanto para capturas de tráfico con pocos activos, en las que se va a tardar poco tiempo por ser un número pequeño de estos como para capturas de tráfico con una gran cantidad de activos. Como conclusión, el tiempo de procesado de la herramienta dependerá tanto del número de activos como del número de parámetros que haya descubierto Zeek y haya anotado en sus archivos log.

5.3. Futuras mejoras de la herramienta

5.3.1. Mejoras para la detección

- Añadir Satori y Ettercap para mejorar la detección del Sistema Operativo.
- Añadir el archivo DNS.log para ver el nombre de dominio del activo.
- Añadir huellas generadas por Cisco Mercury [14] y convertidas a JA3 para detectar no solo posibles Sistemas Operativos, sino posibles procesos de aplicaciones corriendo sobre los equipos finales. Esto podría ser capaz de detectar, con un determinado grado de fiabilidad, aplicaciones que usen SSL/TLS para comunicarse más allá de User-Agents de HTTPS.

5.3.2. Otras mejoras

Como otras futuras mejoras de la herramienta se propone:

- Crear una web para el archivo GEXF en vez de un archivo externo que se deba abrir en GEPHI. Para ello, hay librerías de JavaScript como D3.js que podrían realizar esta función.
- Mejorar el rendimiento en archivos PCAP de mayor tamaño y con más activos por red. Se ha pensado para ello mejorar los algoritmos de búsqueda y ordenación con el uso de sets de Python en lugar de diccionarios para mejorar el rendimiento.

- Crear una interfaz gráfica de usuario en vez de usar la herramienta desde ventana de comandos.
- Añadir la posibilidad de usar captura en tiempo real de una interfaz de red en vez de una captura realizada previamente.
- Añadir la posibilidad de unión de varios pcap en el mismo informe.
- Añadir soporte para archivos pcap-ng, aprovechando que Zeek tiene soporte para estos.
- Añadir opción de multi-procesador y multi-hilo para mejorar la velocidad de la herramienta, dividiendo los pcap en archivos más pequeños. Habría que tener especial atención a la hora de realizar las búsquedas y modificar los activos, ya que se podrían producir condiciones de carrera.

Capítulo 6

Conclusiones

En este TFG se ha desarrollado una herramienta capaz de construir una CMDB usando una captura de tráfico de red, de manera que se realiza de forma pasiva, sin interferir en la red ni ocupar ancho de banda. En esta CMDB se añaden distintos activos descubiertos, así como sus características de dirección IP, Software, Servicios de red, Sistema Operativo, huellas JA3 y posibles User-Agents de dichos activos. Esta CMDB de red puede ser utilizada de manera complementaria a una CMDB profesional y gracias a usar el formato JSON es fácilmente integrable en otros sistemas.

En el proyecto se ha analizado la herramienta Zeek, software que disecciona una captura de tráficos en distintos archivos de registro de red o logs, viendo su funcionamiento, su arquitectura y su programación.

Tras el análisis de Zeek y ver su potencial, se ha desarrollado la herramienta que es capaz de crear la CMDB a partir de estos logs, y se han añadido funcionalidades externas a Zeek para mejorar por ejemplo la detección pasiva del Sistema Operativo. También se ha desarrollado dentro de la herramienta una exportación del pcap a GEXF, el formato de representación de grafos de GEPHI. Por último, se ha realizado un script de instalación de la herramienta en sistemas Ubuntu, para que sea más fácil de instalar y utilizar la herramienta.

En cuanto a los resultados, son bastante satisfactorios, ya que detecta prácticamente todo lo que se esperaba. Hay margen de mejora, como se ha comentado en el apartado de mejoras, pero es una primera aproximación a lo que sería una herramienta profesional.

Como opinión personal, este trabajo ha supuesto un reto tanto de programación en un lenguaje nuevo como es Python, como de entender el funcionamiento de Zeek a bajo nivel, un IPS/IDS profesional con un gran número de posibilidades. Además, ha servido como experiencia para aprender a analizar un software, escribir una memoria de un trabajo, desarrollar una herramienta de seguridad y corregir los errores de esta.

Bibliografía

- [1] *Cyber Research Center - Data Sets — United States Military Academy West Point.* <https://www.westpoint.edu/centers-and-research/cyber-research-center/data-sets>
- [2] *JA3er. JA3 Lookup.* <https://ja3er.com/>
- [3] *NetworkMiner - The NSM and Network Forensics Analysis Tool.* <https://www.netresec.com/?page=NetworkMiner>
- [4] *Passive OS Fingerprinting - NETRESEC Blog.* <https://www.netresec.com/index.ashx?page=Blog&month=2011-11&post=Passive-OS-Fingerprinting>
- [5] *Public PCAP files for download.* <https://www.netresec.com/?page=PcapFiles>
- [6] *Time Complexity - Python Wiki.* <https://wiki.python.org/moin/TimeComplexity>
- [7] *Traffic Analysis Exercises.* <http://malware-traffic-analysis.net/training-exercises.html>
- [8] *Wireshark - Documentation.* <https://www.wireshark.org/docs/>
- [9] *The Zeek Project.* <https://zeek.org/>
- [10] The Growth in Connected IoT Devices Is Expected to Generate 79.4ZB of Data in 2025, According to a New IDC Forecast. In: *IDC* (2019), Jun. <https://www.idc.com/getdoc.jsp?containerId=prUS45213219>
- [11] BASTIAN, Mathieu ; HEYMANN, Sébastien ; JACOMY, Mathieu: *Gephi: An Open Source Software for Exploring and Manipulating Networks.* <http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154>. Version: 2009
- [12] BRAY, Tim: The JavaScript Object Notation (JSON) Data Interchange Format / RFC Editor. Version: December 2017. <https://www.rfc-editor.org/rfc/rfc8259.txt>. RFC Editor, December 2017 (8259). – RFC. – 1–16 S.. – ISSN 2070–1721

- [13] BROTHERSTON, Lee: *SquareLemon Blog*. <https://blog.squarelemon.com/tls-fingerprinting>
- [14] CISCO: *cisco/mercury*. <https://github.com/cisco/mercury>. Version: March 2020
- [15] ECMA, ECMA: 404: The JSON Data Interchange Format. Oct. 2013. In: URL: <http://www.ecma-international.org/publications/files/ECMAST/ECMA-404.pdf>
- [16] HINTEREGGER, Abraham: *CSVtoGEXF*. <https://github.com/oerpli/CSVtoGEXF>, 2017
- [17] LEHTISALO, Mikko: *Brassfork*. <https://github.com/mikkolehtisalo/brassfork>, 2014
- [18] O'DONNELL, Glenn ; CASANOVA, Carlos: *The CMDB Imperative: How to Realize the Dream and Avoid the Nightmares: How to Realize the Dream and Avoid the Nightmares*. Pearson Education, 2009
- [19] SALESFORCE: *salesforce/ja3*. <https://github.com/salesforce/ja3/tree/master/zeek>. Version: Jan 2020
- [20] SINGH, Dhara: *Raspberry Pi hack puts NASA in security jam*. <https://www.cnet.com/news/raspberry-pi-hack-puts-nasa-in-security-jam/>. Version: Jun 2019
- [21] SOMMER, Robin: *Following the Packets: A Walk Through Bro's Internal Processing Pipeline by Robin Sommer*. <https://www.youtube.com/watch?v=fGgHgJEzgLc>
- [22] WRIGHT, Austin ; ANDREWS, Henry ; HUTTON, Ben ; DENNIS, Greg: JSON Schema: A Media Type for Describing JSON Documents / Internet Engineering Task Force. Version: September 2019. <https://datatracker.ietf.org/doc/html/draft-handrews-json-schema-02>. Internet Engineering Task Force, September 2019 (draft-handrews-json-schema-02). – Internet-Draft. – Work in Progress

Lista de Figuras

3.1. Arquitectura de Zeek [9]	10
3.2. Arquitectura detallada de Zeek [21]	11
4.1. Diagrama de flujo de la información	15
4.2. Diagrama de flujo del funcionamiento de Python	19
4.3. Diagrama de flujo de construcción de un activo	20
4.4. Diagrama de flujo de la creación de informes	21
4.5. Funcionamiento de la búsqueda por IP	22
4.6. Diagrama de flujo de la creación del grafo de red	23
4.7. Diagrama de flujo de la instalación	24
4.8. Arquitectura de Active-Mapper sobre Docker	25
5.1. Resultados de la red analizada de USMA	30
5.2. Gráfica de tiempos de cada uno de los procedimientos de Active-Mapper	32
5.3. Gráfica de tiempos de Zeek, p0f y Active-Mapper con y sin validación del JSON Schema	32
5.4. Tiempo de procesado medio por cada activo frente al número de activos en la captura	33
5.5. Crecimiento de Activos y tiempos con respecto a la primera captura frente al número de activos	33
D.1. Grafo sin etiquetas	60
D.2. Laboratorio de datos de GEPHI	60
D.3. Grafo con etiquetas de los nodos visibles	61
D.4. Grafo con distribución Yifan Hu aplicada	62
D.5. Nodos a analizar resaltados en color rojo	62
D.6. Comunicaciones que ha tenido un nodo concreto	63
D.7. Filtro por protocolo aplicado al grafo	64
D.8. Filtro de componente gigante aplicado al grafo	64
F.1. Grafo de red asociado a la captura analizada	69

Lista de Tablas

2.1. Parámetros usados para detectar el Sistema Operativo y sus valores, obtenida de la web de NETRESEC [4]	7
5.1. Tabla comparativa de resultados entre la herramienta NetworkMiner y la herramienta Active-Mapper	28

Anexos

Anexos A

Código de la disección de los logs de Zeek

```
1 import numpy as np
2 import json
3 from jsonschema import *
4 import requests
5 import os
6 import sys
7 import time
8
9 #Se define el constructor del tipo Active, con los campos que posee
#    ↪ cada activo de red
10 class Active:
11     def __init__(self, ip,services,software,os,ja3,useragents):
12         self.ip=ip
13         self.services=services
14         self.software=software
15         self.os=os
16         self.ja3=ja3
17         self.useragents=useragents
18
19 # JSON Schema de un activo de red
20 schema = {
21     "type" : "object",
22     "properties":{
23         "IP": {"type" : "string"},
24         "OS": {"type": ["string", "array"]},
25         "Services": {"type": ["string", "array"]},
26         "Software": {"type": ["string", "array"]},
27         "JA3_Fingerprint": {"type": ["string", "array"]},
28         "Possible User-Agent": {"type": ["string", "array"]}
29     },
30     "required": ["IP"]
31 }
32 # Funcion para validar el JSON Schema e incluirlo en una lista
```

```

33 def validate_and_add(item,schema,list):
34     validate(item, schema=schema)
35     list.append(item)
36
37 # Funcion para quedarse con los elementos unicos de una lista
38 def unique(list1):
39     # Inserta la lista en el set
40     list_set = set(list1)
41     # Convierte el set en una lista de nuevo
42     unique_list = (list(list_set))
43     final_list=[]
44     for x in unique_list:
45         final_list.append(str(np.unique(x))[1:-1].replace("''", ""))
46     return final_list
47
48 # Funcion para eliminar caracteres sobrantes repetidos
49 def cleanname(name):
50     name = str(name).replace(''''', '')
51     name = str(name).replace(''''', '')
52     name = str(name).replace(''', '')
53     name = str(name).replace("#", '')
54     name = str(name).replace(";", '')
55     return name
56
57 # Funcion para leer el archivo conn.log y extraer las direcciones IP
58 def readips(rutatolog):
59     reader=open(rutatolog, 'r')
60     listahosts = []
61     for line in reader:
62         jsonreader = json.loads(line)
63         listahosts.append(jsonreader['id.orig_h'])
64         listahosts.append(jsonreader['id.resp_h'])
65     listaips=unique(listahosts)
66     return listaips
67
68 # Funcion para leer el archivo JA3er.json y guardar un diccionario de
#   ↪ md5:User-Agent
69 def readja3(rutatolog):
70     d=dict()
71     data = json.load(open(rutatolog, 'r'))
72     for user in data:
73         d[user['md5']] = user['User-Agent']
74     return d
75
76 # Funcion para extraer los servicios de red encontrados en el fichero
#   ↪ de known services
77 def knownservices(ips,actives,rutatolog):
78     # Leemos el json de los known_services

```

```

79     reader = open(rutatolog, 'r')
80     lines=[]
81     for line in reader:
82         row = json.loads(line)
83         lines.append(row)
84
85     # Para cada una de las lineas, se busca si coincide la IP y se
86     # → registran los servicios
87     for line in lines:
88         actives[line['host']].services.append(line['service'][0])
89     return actives
90
91 # Funcion para extraer los software encontrados en el fichero software
92 def software(ips,actives,rutatolog):
93     reader = open(rutatolog, 'r')
94     lines = []
95     for line in reader:
96         row = json.loads(line)
97         lines.append(row)
98
99     for row in lines:
100         name=row['unparsed_version']
101         name=cleanname(name)
102         # Detectar algun SO gracias al software
103         if ("ubuntu" in str(np.unique(str(row['unparsed_version']))))or
104             ("Ubuntu" in str(np.unique(str(row['unparsed_version']))))
105             ) or ("Debian" in str(np.unique(str(row['unparsed_version'
106             ])))):
107                 actives[row['host']].os.append("Linux")
108             elif ("Windows" in str(np.unique(str(row['unparsed_version']))))
109                 :
110                     actives[row['host']].os.append("Windows")
111             if (len(name) < 100): # De nuevo para eliminar longitud excesiva
112                 actives[row['host']].software.append(name)
113
114     return actives
115
116 # Funcion para extraer los hashes JA3 y JA3S de clientes y servidores
117     # → SSL
118 def ja3reader(ips,actives,rutatolog):
119     reader = open(rutatolog, 'r')
120     lines=[]
121     for line in reader:
122         row = json.loads(line)
123         lines.append(row)
124
125     # Para cada una de las lineas, se busca si coincide la IP y se

```

```

        ↳ registran los hashes JA3 y JA3S
121
122     for row in lines:
123         if row['established']==True:
124             actives[row['id.orig_h']].ja3.append(row['ja3'])
125             actives[row['id.resp_h']].ja3.append(row['ja3s'])
126     return actives
127
128 # Funcion para extraer del fichero de p0f los Sistemas Operativos
129 def p0freder(ips,actives,rutatolog):
130     with open(rutatolog) as reader:
131         finallist=[]
132         for line in reader:
133             fields = line.split('|')
134             if fields[3] == 'subj=cli':
135                 os = []
136                 ip=fields[1].split('=')[1].split('/')[0]
137
138                 if 'os' in fields[4]:
139
140                     if '???' in fields[4]:
141                         os=[]
142                     else:
143                         actives[ip].os.append(fields[4].split('=')[1])
144             if fields[3] == 'subj=srv':
145                 os = []
146                 ip = fields[2].split('=')[1].split('/')[0]
147                 if 'os' in fields[4]:
148                     if '???' in fields[4]:
149                         os = []
150
151                 else:
152                     actives[ip].os.append(fields[4].split('=')[1])
153     return actives
154
155 # Funcion para realizar busqueda en la bbdd de ja3er y recuperar los
        ↳ User-Agents
156 def ja3lookup(ja3fing,d):
157     useragents=[]
158     for user in ja3fing:
159         if(user in d.keys()):
160             useragents.append(d[user])
161     return useragents
162
163 # Funcion para escribir data.json
164 def writedata(rutatolog,data):
165     writer=open(rutatolog+"data.json", 'w')
166     nlines=0

```

```

167     currentline=0
168     for line in data:
169         nlines=nlines+1
170
171     writer.write("[")
172     for line in data:
173         line=str(line).replace("'", "''")
174         line=str(line).replace("#", "'")
175         writer.write(line)
176         if(currentline!=nlines-1):
177             writer.write(",")
178         currentline=currentline+1
179     writer.write("]")
180
181 cwd=os.getcwd()
182 os.system("figlet Active-mapper")
183 print("Introduce the base directory where the pcap is located: ")
184 dir=sys.stdin.readline().strip()
185 print("Do you want to create the graph associated to the pcap? [y/n] (
    ↪ default n)")
186 decision=sys.stdin.readline().strip()
187 print("Do you want to validate the JSON Schema? [y/n] (default n)")
188 validate_decision=sys.stdin.readline().strip()
189 start_time = time.time()
190 os.system("export PATH=/opt/zeek/bin:$PATH && cd "+dir+" && zeek -Cr "+
    ↪ dir+"*.pcap local")
191 os.system("p0f -r "+dir+"*.pcap -o "+dir+"p0f.log > /dev/null")
192 print("Zeek and p0f finished in %s seconds" % (time.time() -
    ↪ start_time))
193
194 start_time = time.time()
195 d=readja3(cwd+"/JA3db.json");
196 listaips=readips(dir+'conn.log')
197 dictactivos=dict()
198 for ip in listaips: #Se crean todos los activos vacios con solo la IP
199     activo = Active(ip,[],[],[],[],[])
200     dictactivos[ip]=activo
201
202 dictactivos=knownservices(listaips,dictactivos,dir+'known_services.log',
    ↪ )
203 dictactivos=software(listaips,dictactivos,dir+'software.log')
204 dictactivos=ja3reader(listaips,dictactivos,dir+'ssl.log')
205 dictactivos=p0freader(listaips,dictactivos,dir+'p0f.log')
206 print("Finished reading and parsing files in %s seconds" % (time.time(
    ↪ () - start_time)))
207 start_time = time.time()
208 for activo in dictactivos.values():
209     users=[]

```

```

210     activo.ja3 = unique(activo.ja3)
211     activo.useragents=ja3lookup(activo.ja3,d)
212     print("Finished the ja3 lookup in %s seconds" % (time.time() -
213         ↪ start_time))
213
214     start_time = time.time()
215     for activo in dictactivos.values():
216         activo.os=unique(activo.os)
217         activo.services=unique(activo.services)
218         activo.software=unique(activo.software)
219
220     network=[]
221     print("Finished getting the unique data in %s seconds" % (time.time() -
222         ↪ - start_time))
222     start_time = time.time()
223
224     # Formar los activos y agregarlos a la CMDB en data.json
225     for activo in dictactivos.values():
226         active={
227             "IP":activo.ip,
228             "OS":activo.os,
229             "Services":activo.services,
230             "Software":activo.software,
231             "JA3_Fingerprint":activo.ja3,
232             "Possible User-Agent":activo.useragents
233         }
234         if activo.ip not in ['255.255.255.255','0.0.0.0','::']:
235             if validate_decision in ['y','Y']:
236                 validate_and_add(active,schema,network)
237             else:
238                 network.append(active)
239     print("Finished adding the JSON in %s seconds" % (time.time() -
240         ↪ start_time))
241
241     writedata(dir,network)
242     os.system("cat "+dir+"data.json > "+cwd+"/templates/data.json")
243     if(decision =='y'):
244         start_time = time.time()
245         # Uso de brassfork y CSVtoGEXF para obtener el grafo de GEPHI en
246         ↪ out.gexf
246         tmp = os.popen("ls "+dir+"*.pcap").read()
247         command ="./brassfork -nodes="+dir+"nodes.csv -edges="+dir+"edges.
248             ↪ csv"+" -in="+tmp
248         os.system("cd "+cwd+"&& "+command)
249         os.system("sed 's/,/"/g' "+dir+"nodes.csv"+" > "+dir+
249             ↪ finalnodes.csv")
250         os.system("sed 's/,/"/g' "+dir+"edges.csv"+" > "+dir+
250             ↪ finaledges.csv")

```

```

251     os.system("tr '[:upper:]' '[:lower:]' <" + dir + "finaledges.csv >" + dir
252         ↪ +"outputedges.csv")
252     os.system("tr '[:upper:]' '[:lower:]' <" + dir + "finalnodes.csv >" + dir
253         ↪ +"outputnodes.csv")
253     print("Finished using brassfork in %s seconds" % (time.time() -
254         ↪ start_time))
254
255     start_time = time.time()
256     os.system("python3 convCSVtoGEXF.py -n " + dir + "outputnodes.csv -e " +
257         ↪ dir + "outputedges.csv -d " + cwd + "/definitions.txt")
257     os.system("mv " + cwd + "/out.gexf " + dir)
258     os.system("rm " + dir + "*.csv")
259     print("Finished the GEXF creation in %s seconds" % (time.time() -
259         ↪ start_time))

```


Anexos B

Código de la página web del informe

```
1 from flask import Flask, render_template
2 from json2html import *
3 import json
4 import main_functions
5 app = Flask(__name__)
6 print("Servidor con tu informe creado!")
7 @app.route("/")
8 def index():
9     # Abrimos primero el archivo de la CMDB
10    with open('templates/data.json') as json_file:
11        data = json.load(json_file)
12    # Usando la libreria JSON2HTML, convertimos la CMDB en una tabla.
13    # → Despues agregamos la funcion de busqueda.
14    html=json2html.convert(json=data,table_attributes="id=\"myTable\""
15                           → class="header")
16    writer=open('templates/indextabla.html', 'w')
17    writer.write('<html>\n<head>\n')
18    writer.write('<meta name="viewport" content="width=device-width,
19                  → initial-scale=1"><style>* {box-sizing: border-box;}#myInput {
20                  → background-position: 10px 10px;background-repeat: no-repeat;
21                  → width: 100%;font-size: 16px;padding: 12px 20px 12px 40px;
22                  → border: 1px solid #ddd;margin-bottom: 12px;}#myTable {border-
23                  → collapse: collapse;width: 100%;border: 1px solid #ddd;font-
24                  → size: 18px;}#myTable th, #myTable td {text-align: left;
25                  → padding: 12px;}#myTable tr {border-bottom: 1px solid #ddd;}#
26                  → myTable tr.header, #myTable tr:hover {background-color: #
27                  → f1f1f1;}</style>')
28    writer.write('</head>')
29    writer.write('<body>')
30    writer.write('<input type="text" id="myInput" onkeyup="myFunction()"
31                  → placeholder="Busca una IP:" title="Busca una IP">')
32    writer.write(html)
33    writer.write('<script>function myFunction() {var input, filter,
```

```

    ↵ table, tr, td, i, txtValue;input = document.getElementById("myInput");filter = input.value.toUpperCase();table = document
    ↵ .getElementById("myTable");tr = table.getElementsByTagName("tr");for (i = 0; i < tr.length; i++) {td = tr[i].
    ↵ getElementsByTagName("td")[0];if (td) {txtValue = td.
    ↵ textContent || td.innerText;if (txtValue.toUpperCase().
    ↵ indexOf(filter) > -1) {tr[i].style.display = "";} else {tr[i]
    ↵ ].style.display = "none";}}}}</script>'')
22 writer.write('\n</body>\n</html>')
23 writer.close()
24 # Cargamos como template el informe HTML generado
25 return render_template('indextabla.html')
26
27 @app.route("/data.json")
28 def data():
29     # Cargamos la CMDB por si se quiere integrar con otro sistema
30     return render_template('data.json')
31
32 if __name__ == '__main__':
33     app.run(debug=False)

```

Anexos C

Código del ejecutable de instalación

```
1 #!/bin/sh
2 echo Welcome to the Active-mapper installation. This will require sudo
   ↪ privileges
3 dir=$PWD
4 sudo apt-get update
5 echo Installing Zeek and dependencies...
6 sudo apt-get install nano wget p0f flex bison libpcap-dev libssl-dev
   ↪ python-dev swig zlib1g-dev git figlet -y --fix-missing
7 sudo sh -c "echo 'deb http://download.opensuse.org/repositories/
   ↪ security:/zeek/xUbuntu_18.04/ /' > /etc/apt/sources.list.d/
   ↪ security:zeek.list"
8 wget -nv https://download.opensuse.org/repositories/security:zeek/
   ↪ xUbuntu_18.04/Release.key -O Release.key
9 sudo apt-key add - < Release.key
10 sudo apt-get update
11 sudo apt-get install zeek -y
12 sudo mkdir /opt/zeek/share/zeek/site/ja3/
13 cd && git clone https://github.com/salesforce/ja3.git
14 cd ja3/
15 sudo cp zeek/* /opt/zeek/share/zeek/site/ja3/
16 cd && rm -rf ja3/
17 cd $dir
18 wget https://ja3er.com/getAllUasJson -O JA3db.json
19 sudo chmod +x brassfork
20 echo '@load tuning/json-logs' | sudo tee -a /opt/zeek/share/zeek/site/
   ↪ local.zeek
21 echo '@load tuning/track-all-assets' | sudo tee -a /opt/zeek/share/zeek
   ↪ /site/local.zeek
22 echo '@load ./ja3' | sudo tee -a /opt/zeek/share/zeek/site/local.zeek
23 echo Configuring python env...
24 sudo apt install python3-pip -y
25 cd $dir
26 pip3 install -r requirements.txt
27 clear
28 echo 'Instalation finished! For running the program you have to write'
```

```
↪ in this console: python3 app.py'  
29 echo 'Hope you like it! - Created by Javier Ortega - Universidad de  
↪ Zaragoza'
```

Anexos D

Manejo de GEPHI para extraer información de red

GEPHI (conocida como *The Open Graph Viz Platform*) es una herramienta open-source de visualización y manejo de grafos y redes, escrita en Java y que funciona en sistemas Windows, Linux y Mac OS X. Esta herramienta se utiliza para análisis de datos, de redes y relaciones, de redes sociales, de biología y muchas otras aplicaciones. Fue desarrollado por estudiantes de la University of Technology of Compiègne de Francia, y se inició en 2009.

En nuestra aplicación, se usa GEPHI para ver la información de la red y la comunicación que ha habido entre nodos. Muchas veces, dependiendo de dónde se ha realizado la captura de tráfico (dónde se ha colocado el dispositivo de captura), se van a ver patrones en los grafos muy marcados como un NAT (muchos distintos tipos de tráfico saliente en el mismo dispositivo), endpoints (tráfico mayoritario HTTP y HTTPS saliente), servidores de aplicaciones (tráfico entrante de un protocolo concreto), y muchos otros. El estudio de extrapolar la información de nodos a equipos reales y crear un mapa de red se sale del objetivo del TFG, pero puede ser una ayuda para comprobar el tipo de comunicación que tienen los equipos que se han registrado en la CMDB creada por la herramienta.

Una vez abierto el grafo en GEPHI, lo que se puede ver son dos ventanas distintas como son el grafo de nodos y enlaces tal y como se observa en la Figura D.1, y el laboratorio de datos, donde se pueden ver los nodos y los enlaces con sus distintas características (capa de transporte, protocolo, etc), tal y como se puede comprobar en la Figura D.2.

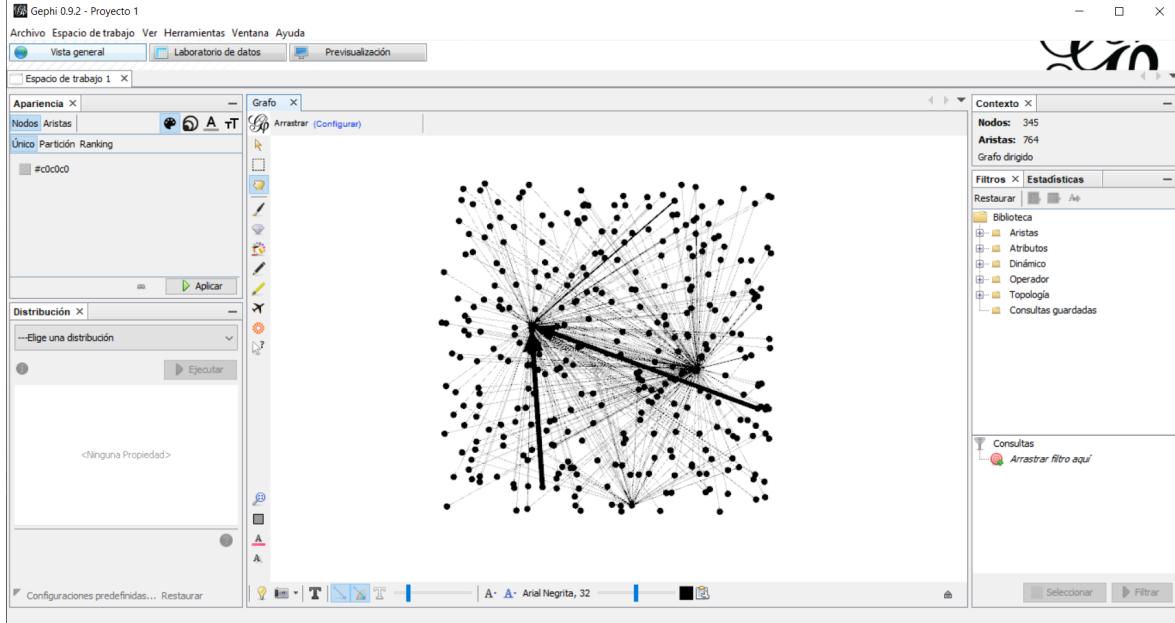


Figura D.1: Grafo sin etiquetas

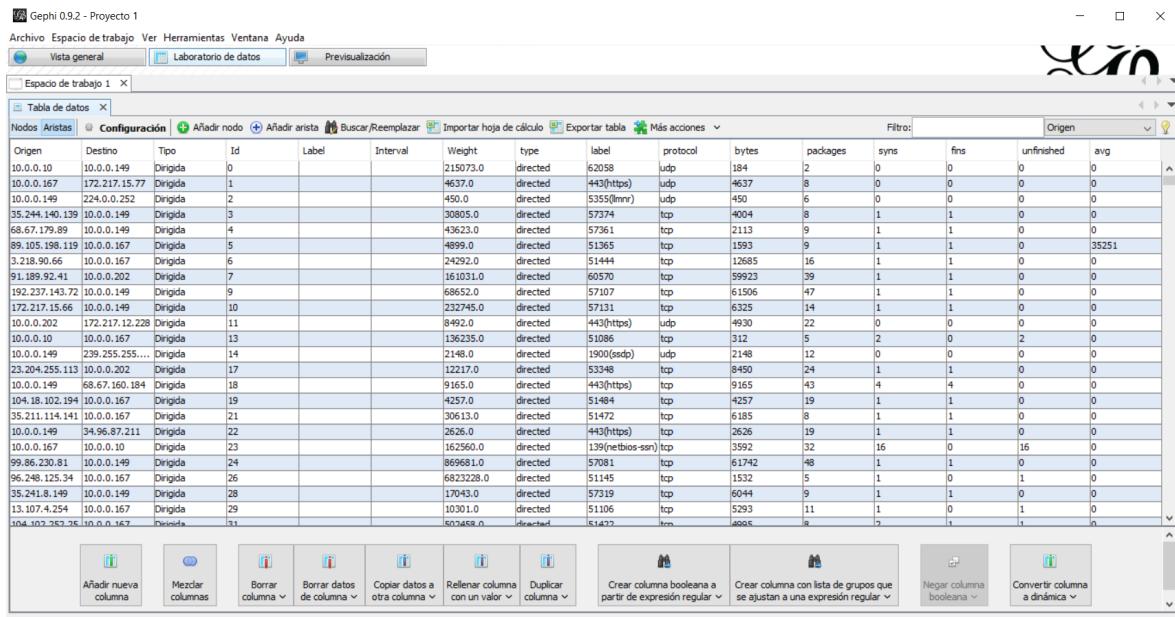


Figura D.2: Laboratorio de datos de GEPHI

Para mostrar las etiquetas de los nodos, es decir, las direcciones IP, se debe seleccionar la letra **T** de la parte inferior de la ventana y ajustar con la barra que se encuentra

a su derecha el tamaño de las etiquetas de los nodos. Aunque se hayan activado las etiquetas, con los nodos mal ordenados, es muy complicado de ver algo de utilidad en el grafo tal y como se observa en la Figura D.3. Para esto, se usan distribuciones de nodos. Una de las distribuciones más utilizadas es la Yifan Hu, que agrupa por nodos directamente relacionados y separa los no relacionados. Para aplicar una de las distribuciones que tiene GEPHI, se debe hacer en la pestaña de Distribuciones que se encuentra a la izquierda de la ventana, seleccionar la adecuada y presionar Ejecutar. Tras ello, se aplicará y se reordenarán los nodos, quedando como en la Figura D.4.

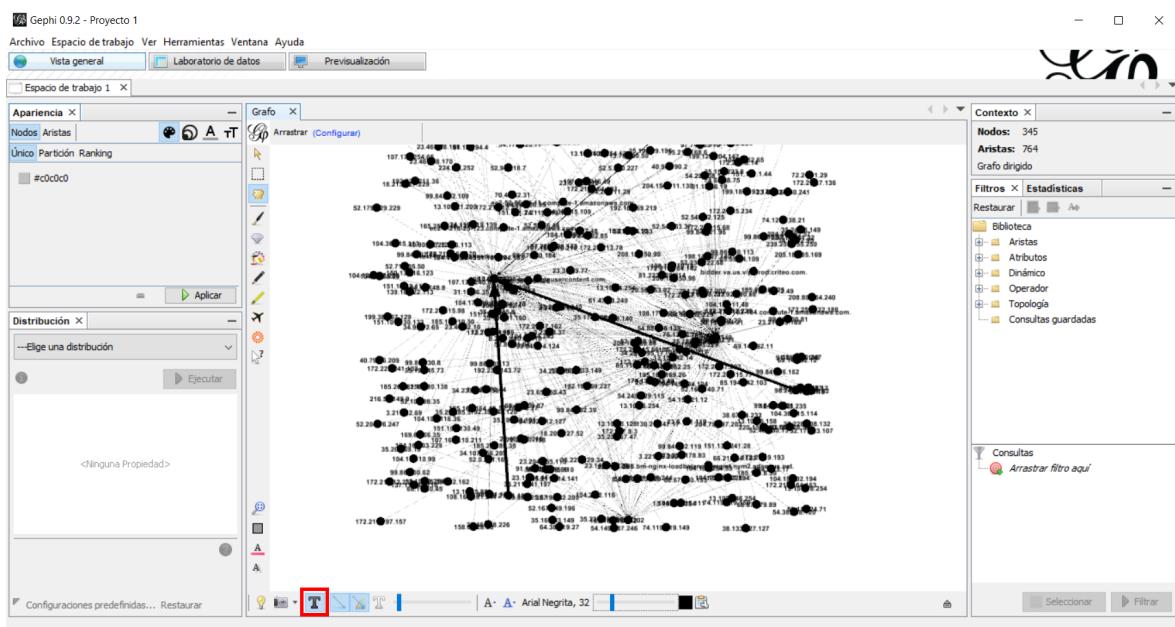


Figura D.3: Grafo con etiquetas de los nodos visibles

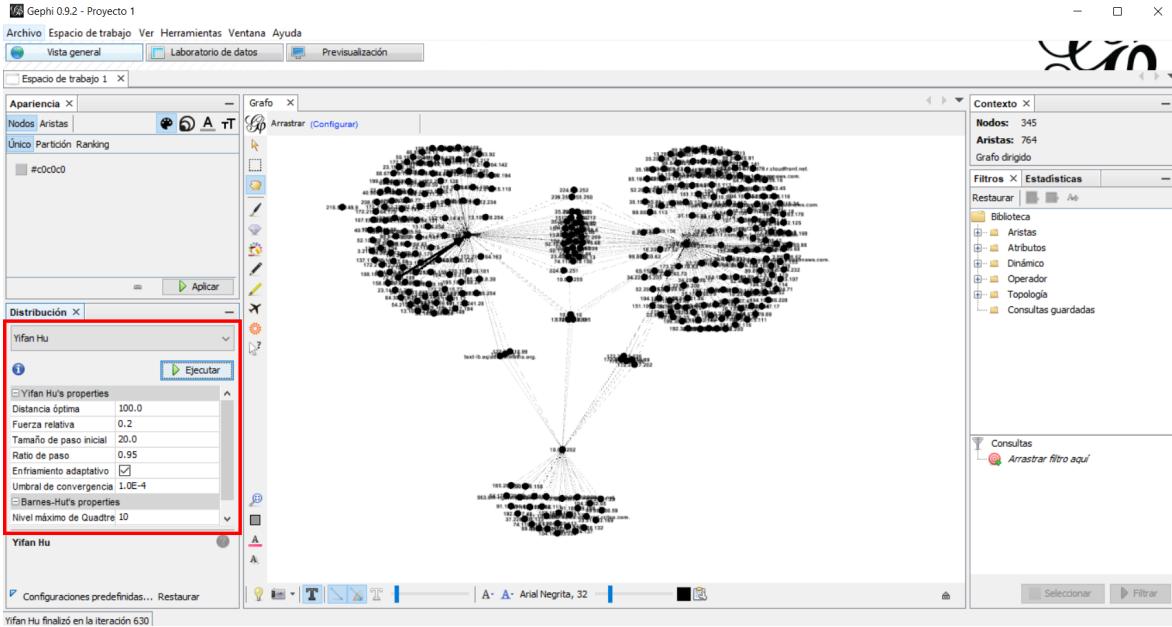


Figura D.4: Grafo con distribución Yifan Hu aplicada

Para resaltar los nodos que nos interesa analizar, los seleccionamos en el Laboratorio de datos y tras eso seleccionamos la opción de Resaltar en la vista de grafo. Después, una vez en la vista del grafo, elegimos la herramienta de modificar nodos y escogemos un color distinto, en nuestro los nodos de la red 10.0.0.0/24 resaltados en color rojo, como se ve en la Figura D.5.

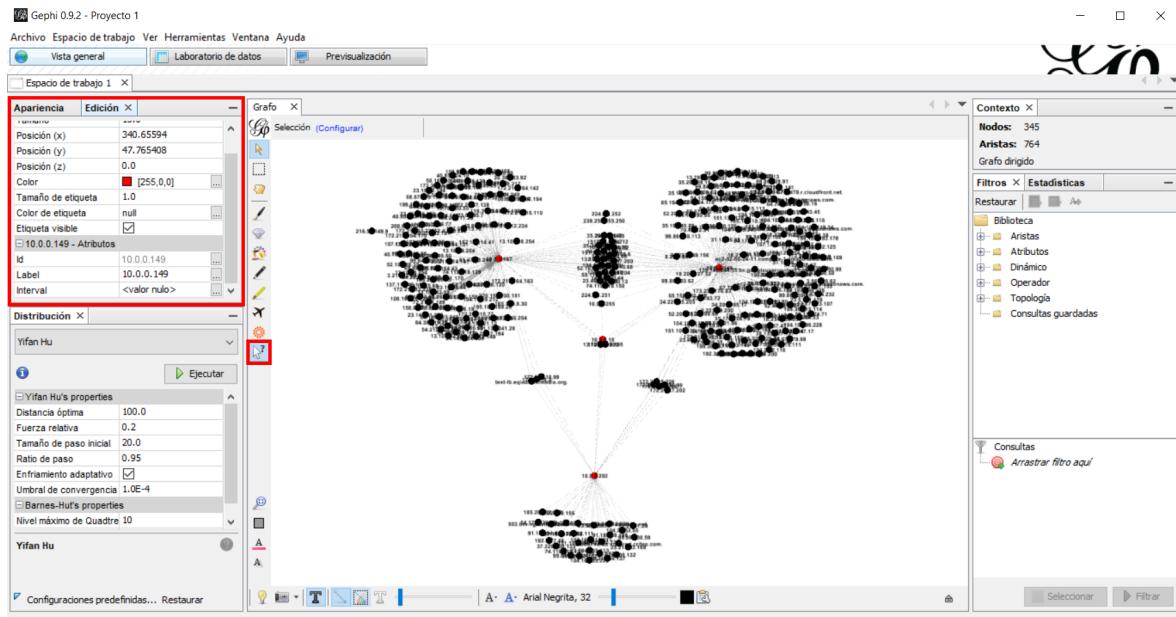


Figura D.5: Nodos a analizar resaltados en color rojo

Una de las mejores utilidades del grafo de red es el poder ver los enlaces entre nodos como protocolos. Para ello, se debe ir a Apariencia, seleccionar Aristas, Partición y filtrar por Etiqueta (*label*). Tras ejecutar Aplicar se van a generar automáticamente colores por cada protocolo de red y se van a observar en el grafo. Esto es útil por ejemplo para ver si se tiene un cliente o un servidor HTTP, si ha habido conexiones SSH, y otros muchos ejemplos. Como se puede observar en la Figura D.6, en el grafo hay una gran cantidad de conexiones TLS/SSL en el puerto 443, así como HTTP o IMAPS, cada una con un color distinto. El nodo que sale representado en la figura ha tenido comunicaciones con el exterior (flechas que parten desde el nodo) a través de HTTPS, HTTP e IMAPS, y ha tenido comunicaciones hacia él usando NETBIOS-SSN (puerto 139).

Combinando esta funcionalidad con el informe de activos creado, se puede ver que el equipo con la IP 10.0.0.167 es un Windows 10 con distintos servicios de red implementados, e incluso se ha podido extraer distintos User-Agents de él.

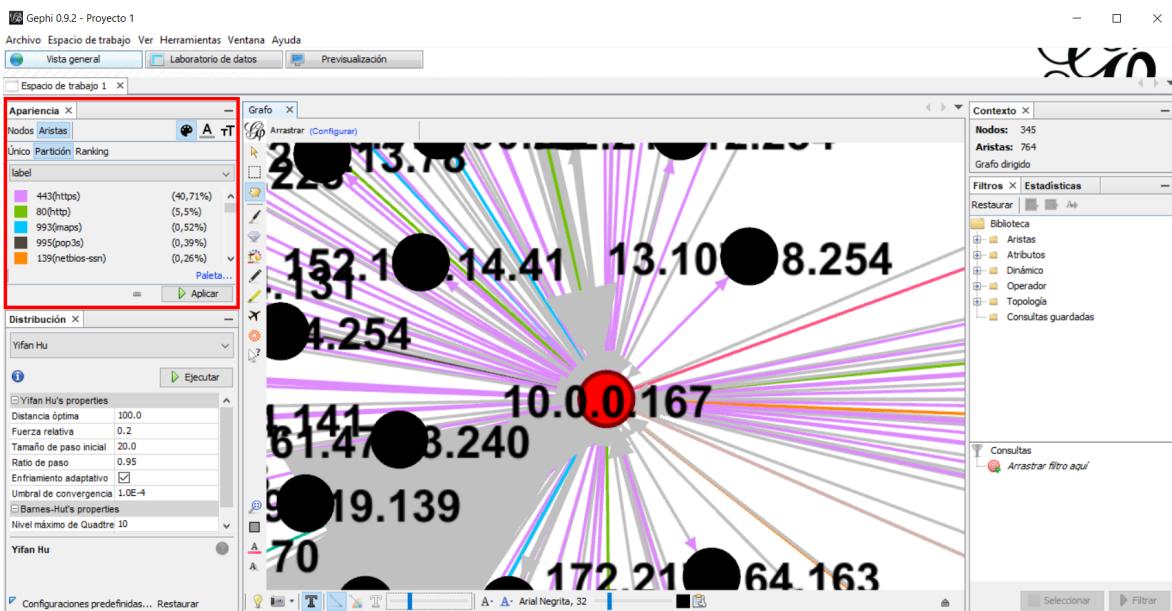


Figura D.6: Comunicaciones que ha tenido un nodo concreto

Otra de las funcionalidades más útiles de GEPHI es añadir filtros a los enlaces. Para ello, en la ventana de la derecha se selecciona el filtro y se arrastra a la parte inferior. Se pueden crear combinación de filtros como puede ser de unión o intersección utilizando lógica. En la Figura D.7 se ha aplicado el filtro de Unión y dos subfiltros de igualdad de etiqueta (protocolo) para los protocolos de HTTPS y NETBIOS-SSN, como se puede ver en el grafo.

Por último, una de las funcionalidades más útiles de GEPHI es el filtro de componente gigante. Este, si se aplica, elimina los valores “atípicos”, o valores que tienen

menos relaciones con el resto o se encuentran más aislados. En la figura D.8 se observa que aplicando este filtro se ha eliminado una de las partes del grafo, aunque en este grafo no sea lo más útil, ya que esos nodos también son importantes, en otros grafos es imprescindible realizar esta operación para eliminar nodos no útiles.

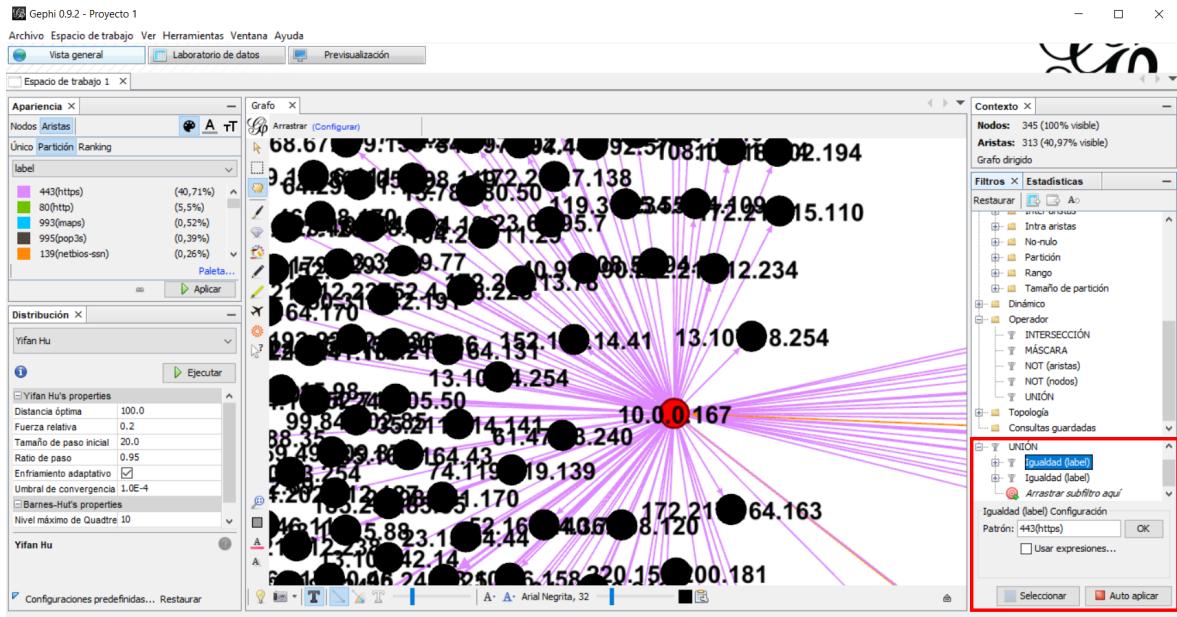


Figura D.7: Filtro por protocolo aplicado al grafo

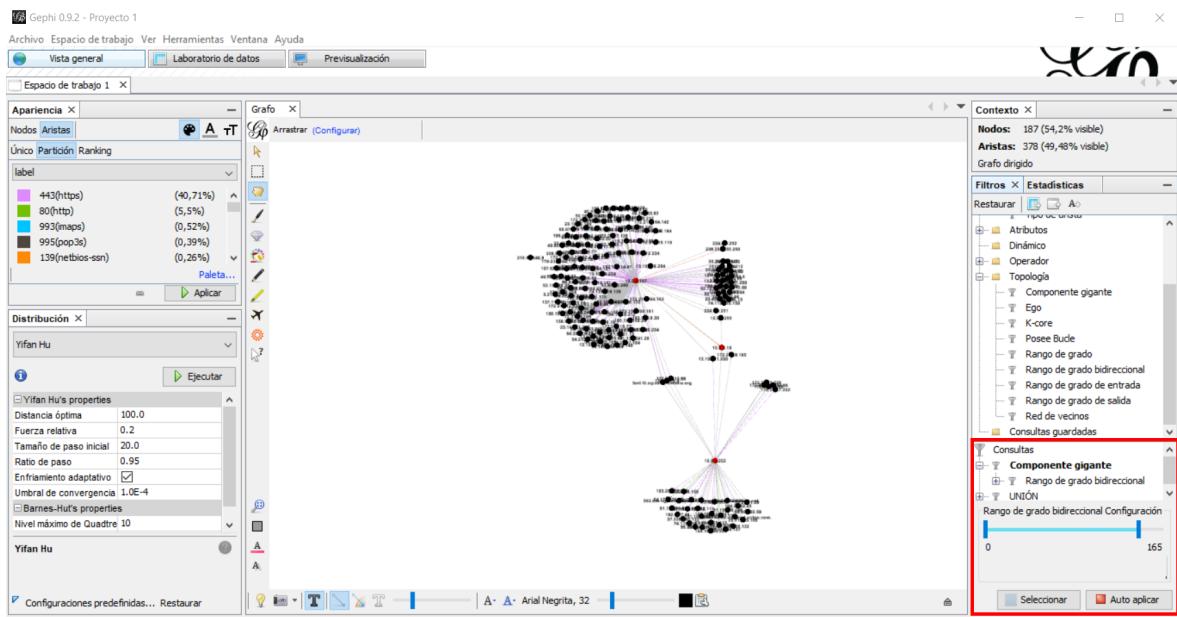


Figura D.8: Filtro de componente gigante aplicado al grafo

Anexos E

Código del Dockerfile

```
1 FROM ubuntu:18.04
2 RUN echo 'debconf debconf/frontend select Noninteractive' | debconf-set
3     ↪ -selections && \
4 apt-get install -y -q
5 RUN mkdir /etc/dma/
6 RUN echo 'user@example.org|smtp.example.org:password' | tee -a /etc/dma
7     ↪ /auth.conf && \
8 echo 'MAILNAME /etc/mailname' | tee -a /etc/dma/dma.conf && \
9 echo 'example@example.org' | tee -a /etc/mailname
10 RUN apt-get update -y && \
11     apt-get upgrade -y && \
12     apt-get install git sudo gnupg -y
13
14 RUN apt-get install -y dma --option=Dpkg::Options::=--force-confdef
15
16 RUN useradd --create-home --shell /bin/bash ubuntu && \
17     echo 'ubuntu:ubuntu' | chpasswd
18 RUN usermod -aG sudo ubuntu
19 ADD TFG/ /home/ubuntu/
20
21 RUN sudo sh /home/ubuntu/install.sh
22
23 RUN pip3 install -r /home/ubuntu/requirements.txt
24
25 RUN sudo chmod +x /home/ubuntu/brassfork && \
26     chown -R ubuntu:ubuntu /home/ubuntu/
27 RUN sudo mv /JA3db.json /home/ubuntu/
28 EXPOSE 5000
29
30 ENTRYPOINT ["/bin/bash"]
```


Anexos F

Ejemplo de resultados

F.1. Archivo data.json

```
1 [{"IP": "185.174.175.14", "OS": [], "Services": ["SSL"], "Software": []
  ↪ [], "JA3_Fingerprint": ["80b3a14bcc8598a1f3bbe83e71f735f"], "
  ↪ Possible User-Agent": []}, {"IP": "184.72.249.110", "OS": [], "
  ↪ Services": ["HTTP"], "Software": ["Cowboy"], "JA3_Fingerprint": []
  ↪ [], "Possible User-Agent": []}, {"IP": "216.58.193.68", "OS": [], "
  ↪ Services": ["SSL"], "Software": [], "JA3_Fingerprint": [
  ↪ f9a66afdd1f499d415ca470974ec00c8], "Possible User-Agent": []}, {""
  ↪ IP": "10.0.14.129", "OS": ["Windows 7 or 8", "Windows"], "
  ↪ Services": [], "Software": ["Mozilla/5.0 (Windows NT 6.1 Win64
  ↪ x64 Trident/7.0 rv:11.0) like Gecko", "Microsoft NCSI"], "
  ↪ JA3_Fingerprint": ["4d7a28d6f2263ed61de88ca66eb011e3"], "Possible
  ↪ User-Agent": ["Tofsee (from abuse.ch)"]}, {"IP": "173.223.52.18",
  ↪ "OS": [], "Services": ["HTTP"], "Software": [], "JA3_Fingerprint": []
  ↪ [], "Possible User-Agent": []}, {"IP": "10.0.14.3", "OS": [
  ↪ Windows 7 or 8"], "Services": ["SMB", "KRB", "DCE_RPC", "KRB_TCP"
  ↪ , "NTLM", "NTP", "GSSAPI"], "Software": [], "JA3_Fingerprint": [],
  ↪ "Possible User-Agent": []}, {"IP": "239.255.255.250", "OS": [], "
  ↪ Services": [], "Software": [], "JA3_Fingerprint": [], "Possible
  ↪ User-Agent": []}, {"IP": "185.43.223.6", "OS": [], "Services": [
  ↪ HTTP"], "Software": ["nginx/1.10.2", "PHP/5.4.45"], "
  ↪ JA3_Fingerprint": [], "Possible User-Agent": []}, {"IP": "
  ↪ 111.90.144.30", "OS": [], "Services": ["HTTP"], "Software": [
  ↪ PleskLin", "nginx"], "JA3_Fingerprint": [], "Possible User-Agent": []
  ↪ []}, {"IP": "224.0.0.252", "OS": [], "Services": [], "Software": [
  ↪ ], "JA3_Fingerprint": [], "Possible User-Agent": []}, {"IP": "
  ↪ 10.0.14.1", "OS": [], "Services": ["DHCP"], "Software": [],
  ↪ JA3_Fingerprint": [], "Possible User-Agent": []}, {"IP": "
  ↪ 10.0.14.255", "OS": [], "Services": [], "Software": [],
  ↪ JA3_Fingerprint": [], "Possible User-Agent": []}]}
```

F.2. Informe HTML

Busca una IP:

IP	OS	Services	Software	JA3_Fingerprint	Possible User-Agent
185.174.175.14		• SSL		• 80b3a14bcc8598a1f3bbe83e71f735f	
184.72.249.110		• HTTP	• Cowboy		
216.58.193.68		• SSL		• f9a66afdd1f499d415ca470974ec00c8	
10.0.14.129	• Windows 7 or 8 • Windows		• Mozilla/5.0 (Windows NT 6.1 Win64 x64 Trident/7.0 rv:11.0) like Gecko • Microsoft NCSI	• 4d7a28d6f2263ed61de88ca66eb011e3	• Tofsee (from abuse.ch)
173.223.52.18		• HTTP			
10.0.14.3	• Windows 7 or 8		• SMB • KRB • DCE_RPC • KRB_TCP • NTLM • NTP • GSSAPI		
239.255.255.250					
185.43.223.6		• HTTP		• nginx/1.10.2 • PHP/5.4.45	
111.90.144.30		• HTTP		• PleskLin • nginx	
224.0.0.252					
10.0.14.1		• DHCP			
10.0.14.255					

F.3. Grafo de red

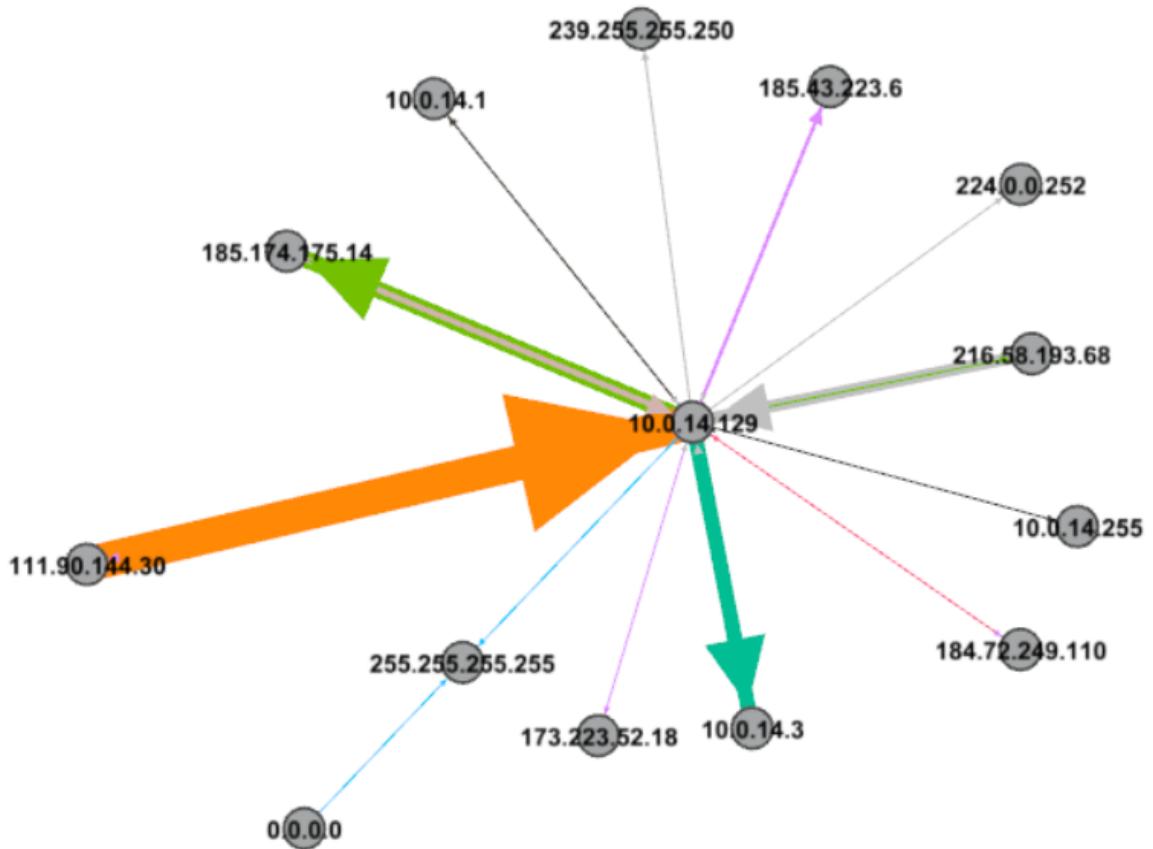


Figura F.1: Grafo de red asociado a la captura analizada