TRABAJO FIN DE GRADO

INGENIERÍA EN INFORMÁTICA

Implementación en tiempo real de sistemas de identificación de tráfico de red

Subtitulo del proyecto

Autor
Álvaro Maximino Linares Herrera

Directores
Jesús Esteban Díaz Verdejo

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, 3 de septiembre de 2017

Implementación en tiempo real de sistemas de identificación de tráfico de red

Subtitulo del proyecto

Autor
Álvaro Maximino Linares Herrera

Directores
Jesús Esteban Díaz Verdejo

Implementación en tiempo real de sistemas de identificación de tráfico de red

Álvaro Maximino Linares Herrera

Palabras clave: inspección profunda de paquetes, bro, flujos, identificación, clasificación, red, tráfico

Resumen

La identificación de tráfico en red es realmente importante para aplicaciones de ingeniería de tráfico y de seguridad.

En este trabajo se tratará la creación de un programa para un NMS (Network Monitoring System), en este caso se usará BRO, mediante el cual se pueda resolver el emparejamiento de flujos. BRO consiste en un NMS que funciona mediante el terminal en Linux o Mac, una de las peculiaridades de este programa es que para la creación de scripts que nos permitan extender la funcionalidad de la que dispone, tendremos que usar BRO como lenguaje de programación. Es un lenguaje de scripting, el cual está orientado a eventos, que se lanzan cuando ocurre algo relacionado con el control y análisis de redes, es un lenguaje que para los que vienen de C++, Java o Python, no debe de suponer un gran reto, más allá de acostumbrarse a sus sintaxis. Es un lenguaje potente que al estar orientado a redes nos permite obtener mucha información de los flujos que tenemos en la red o en el archivo que vayamos a analizar. En este trabajo mediante implementaciones offline se verificará la eficacia de esta técnica de clasificación de tráfico.

Project Title: Project Subtitle

First name, Family name (student)

Keywords: Keyword1, Keyword2, Keyword3, ...

Abstract

Write here the abstract in English.

Yo, **Álvaro Maximino Linares Herrera**, alumno de la titulación TITULACIÓN de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 76669401M, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Álvaro Maximino Linares Herrera

Granada a 3 de septiembre de 2017.

D. **Jesús Esteban Díaz Verdejo**, Profesor del Área de XXXX del Departamento YYYY de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado ***Implementación en tiempo real de sistemas de identificación de tráfico de red***, ha sido realizado bajo su supervisión por **Álvaro Maximino Linares Herrera**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 3 de septiembre de 2017.

Los directores:

Jesús Esteban Díaz Verdejo **Nombre Apellido1 Apellido2 (tutor2)**

Agradecimientos

Poner aquí agradecimientos...

Índice general

Capítulo 1

Introducción

1.1. Motivación y antecedentes

Dada la gran expansión de Internet desde los años 90, la seguridad se ha convertido en algo fundamental. Hoy en día el uso de Internet se ha generalizado tanto que ha llegado a todas las áreas, desde finanzas hasta el envío de pruebas médicas, pasando por compras, redes sociales y demás, por lo tanto se tratan temas de carácter muy privado y personal.

Internet se organiza mediante un modelo de capas. Hay varios modelos de capas, como son el modelo TCP/IP y el modelo OSI. [1]

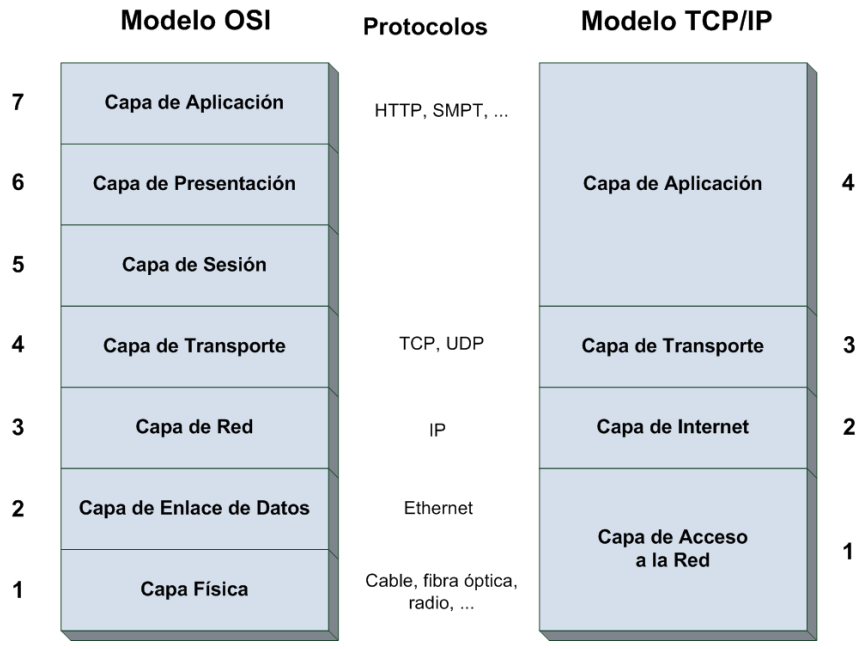

Figura 1.1: Modelos de capas de Internet.

Dentro del modelo TCP/IP, que es el que se seguirá en este proyecto, es importante tener claro las siguientes capas, pues forma parte de los conocimientos básicos para lograr entender el emparejamiento de flujos.

- **Capa de aplicación**

  En la capa de aplicación se encontrarán las aplicaciones de red y sus protocolos. Los protocolos que se encuentran en esta capa son HTTP, SMTP y FTP.

  - El protocolo *HTTP* permite la solicitud y transferencia de documentos web.
  - El protocolo *SMTP* permite la transferencia de mensajes de correo electrónico.
  - El protocolo *FTP* permite la transferencia de archivos entre dos terminales.

  A parte de estos protocolos también corresponde a esta capa el **DNS**, el cual se encarga de traducir los nombres de los sitios web.

  Para una información más extensa sobre la capa de aplicación consulte [2]

- **Capa de transporte**

  La capa de transporte es la encargada de transportar los mensajes de la capa de aplicación. Los protocolos de esta capa son **TCP y UDP**. La principal diferencia y será algo que siempre se debe de tener presente cuando se trabaja con redes es que *TCP* está orientado a garantizar la conexión, mientras que *UDP* no garantiza la conexión.

  Para ampliar información sobre esta capa consulte [3]

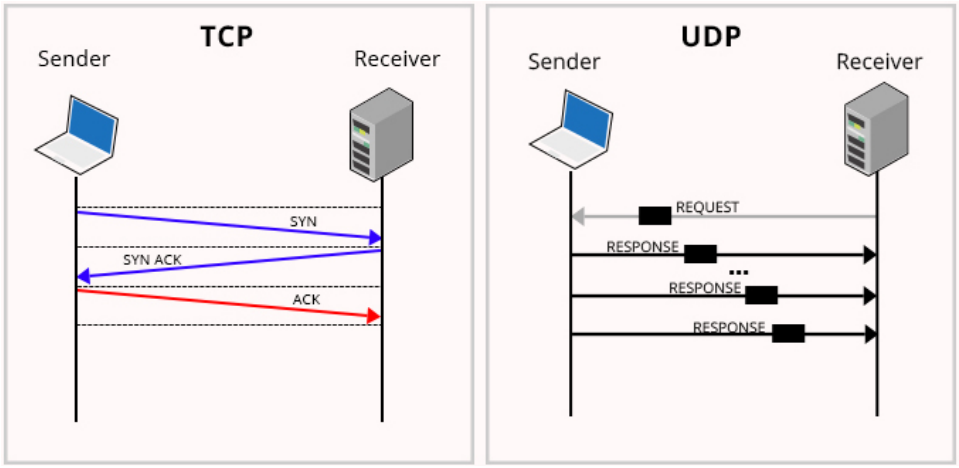

Figura 1.2: Diagrama de conexión de TCP y UDP.

- **Capa de red**

  En la capa de red es donde se encuentra el protocolo **IP**, el cual es el encargado de especificar el formato de envío entre los routers y los hosts y también el protocolo **ICMP**, el cual es un protocolo de control.

  En esencia, esta capa es la que se encarga de transportar los *datagramas* de un *host* a otro.

  Para ampliar sus conocimientos sobre la capa de red consulte [4]

En Internet la información que se envía se divide en paquetes. Estos paquetes tienen una IP de origen y otra de destino, así como un puerto de origen y un puerto de destino. Estos paquetes, que contienen parte de la información que se envía por la red, son unidos de nuevo una vez que llegan a su destino, de forma que entonces se dispondrá de la información completa. Dependiendo del tipo de protocolo se tolerarán perdidas de paquetes o no, para ilustrar esta idea, en una llamada mediante *Skype* se tolera perder algunos paquetes, pues no se notará mucha diferencia. Si se pierden muchos paquetes la imagen se verá borrosa y la voz no se escuchará bien. Ahora en un correo electrónico, no se tolerará perder paquetes, pues entonces no se obtendrá la información completa. Cuando varios paquetes tienen el mismo origen y el mismo destino se denomina flujo. [5]

De esta forma se llega al emparejamiento de flujos, el cual es muy útil para los NMS, sistemas de monitoreo de red, *Network Monitoring System*, como podría ser BRO [6]. Gracias al uso de estos programas el administrador

de sistemas podrá prevenir ataques, como por ejemplo, los de denegación de servicio, DDoS, *Distributed Denial of Service* [7]. Un administrador de sistemas deberá de garantizar que el recurso que facilita y gestiona esté operativo siempre, por lo tanto analizará el tráfico para evitar los posibles ataques, virus, etc.

Existen otras técnicas de clasificación de tráfico, como se puede ver en el artículo del departamento [8], estas técnicas son:

- Clasificación basada en los puertos de la capa de transporte. [9]
- Clasificación basada en el contenido del paquete. [10]
- Clasificación basada en la aplicación de técnicas de aprendizaje automático sobre estadísticas de tráfico. [11]

Hoy en día, las dos primeras técnicas mencionadas no son muy eficaces, pues por ejemplo:

- La primera técnica puede ser burlada fácilmente, haciendo que las conexiones pasen por el puerto 80.
- En la segunda técnica, al tener que acceder a los paquetes, solo bastará con que estos estén encriptados para que no se pueda acceder a dicho paquete. Y también se debe de tener en cuenta que al acceder a los paquetes de una red que no es del propietario de la información es un delito y no respeta la privacidad.

Ahora bien, la tercera técnica si es eficaz y fiable, pero su principal inconveniente es que necesita cierto tiempo para aprender a clasificar de una forma correcta el tráfico.

Por lo tanto es necesario un nuevo método de clasificación, que sea funcional en cualquier circunstancia y sea eficiente desde el principio. Por ello, el método de emparejamiento planteado por el departamento [8] es muy atractivo al ser necesario saber solo las IP's y los puertos, ya que se respeta la seguridad, pues no es necesario acceder a los paquetes para conocer estos detalles. Incluso no sería necesario ser un proveedor de servicios de Internet, ISP, para conocer estos datos.

1.2. Objetivos

En este trabajo se hará uso del NMS mencionado anteriormente, llamado Bro [6], el cual cuenta con su propio lenguaje de scripting. Dicho lenguaje es de fácil uso y comprensión, además de que está completamente orientado a

trabajar con conexiones, teniendo distintos tipos de datos que son realmente útiles. Dicho esto el objetivo del trabajo es desarrollar un módulo para Bro. En este módulo irá implementada la técnica de emparejamiento de flujos, de forma que pueda ser usada en redes fuera de un entorno de laboratorio.

- Para ello será necesario conocer como Bro gestiona los flujos, desde su nacimiento a su muerte.
- También hará falta gestionar las entradas y salidas, aunque al principio se hará uso de un archivo *pcap* de prueba para controlar si todo funciona bien, pudiendo pasar luego a analizar el tráfico de una red online.
- Implementar la función de emparejamiento de flujos.
- Será necesario realizar pruebas al módulo que se pretende desarrollar, de modo que se detecten los errores y se subsanen.

Por último, y fuera de los objetivos más técnicos, el desarrollo de esta memoria también se considera un objetivo, que se tendrá en cuenta dentro de la temporización.

1.3. Metodología

Para realizar este trabajo se establecen una serie de tareas:

- Lectura del artículo del departamento. [8]
- Estudio del uso de Bro y de su lenguaje de scripting.
- Pruebas con el lenguaje propio de Bro.
- Desarrollo del módulo en Bro.
- Pruebas del módulo.
- Desarrollo de la memoria.

Cada tarea anterior se realimentará de la siguiente, el ejemplo más claro es el desarrollo del módulo, que irá cambiando por las pruebas, pues se detectarán los fallos, a su vez habrá que releer el artículo mientras se desarrolla el módulo para asegurarse de que todo se realiza de acuerdo al trabajo previo, etc.

En el siguiente diagrama de Gantt se puede ver una temporización de estas tareas.

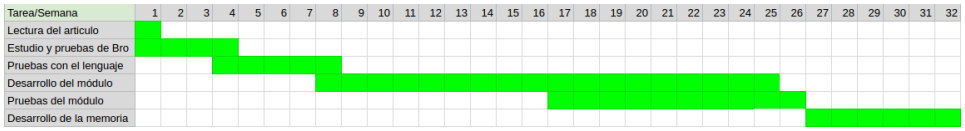

Figura 1.3: Temporización en diagrama de Gantt.

Este proyecto no necesita de ningún gasto, pues Bro [6] proporciona sus binarios de forma gratuita. El módulo será subido a GitHub [12] con licencia de software libre, por lo que cualquiera podrá usarlo o modificarlo en el futuro. El único *gasto* es el de un ordenador con Linux, o bien con Mac OS X, pues de momento no está disponible para Windows. [13]

1.4. Estructura de la memoria

Esta memoria se organizará de la siguiente forma:

- En el capítulo 2 se hablará de todos los fundamentos teóricos y tecnológicos sobre los que se basa el proyecto.
- En el capítulo 3 se contará cómo se pretende resolver el problema expuesto.
- En el capítulo 4 se encontrará detallado cómo se han implementado los diferentes módulos.
- En el capítulo 5 se realizarán las pruebas, para comprobar que todo funciona como estaba previsto, tanto a nivel funcional como a nivel de aplicación.
- En el último capítulo se hablará de las conclusiones recogidas a lo largo de este proyecto y las posibles opciones que tiene para seguir trabajando sobre él.

Capítulo 2

Estado del arte

Aquí se encuentra la parte teórica y tecnológica del proyecto. Se hablará de la identificación de tráfico y de las distintas técnicas que existen para ello. También se hablará de Bro [6], el analizador de tráfico que se usará para llevar a cabo el desarrollo del módulo. Se introducirá en que consiste, cómo gestiona los eventos y sus funcionalidades básicas, así como la posibilidad de ampliar estas.

Por último se presentará de forma teórica en que consiste el emparejamiento de flujos y de qué forma se podría usar para identificar el tráfico.

2.1. Identificación de tráfico

Hay que tener claro que identificar el tráfico no es clasificarlo. Identificar el tráfico consiste en analizarlo y obtener patrones comunes con los cuales se pueda ordenar el tráfico de forma posterior. Algunos de los patrones que se pueden tener en cuenta a la hora de identificar tráfico son los puertos, las IP's o la clase a la que pertenece.

Por lo tanto la identificación del tráfico de la red es esencial para realizar una clasificación posterior. Dicha clasificación se podrá analizar en busca de amenazas o intrusos, a nivel de seguridad y componentes lentos o defectuosos que afecten al rendimiento del servicio. Para poder identificar el tráfico lo deseable es un método que sea rápido y proteja la privacidad de los distintos usuarios.

2.1.1. Técnicas de identificación de tráfico

Existen varias técnicas de identificación de tráfico, como ya se explicó anteriormente. Estas técnicas son el paso previo a la clasificación del tráfico. Podría considerarse que son el paso intermedio a la clasificación, teniendo

en cuenta que el tráfico es capturado, identificado y clasificado. El tráfico, hasta ahora, se puede identificar de la siguiente forma.

- Por los puertos de la capa de transporte, establecido por *IANA*. [9]
- Por el contenido del paquete o *DPI*. [10]
- Por la aplicación de técnicas de aprendizaje automático sobre estadísticas de tráfico, *machine learning*. [11]

En los siguientes apartados se verán estas técnicas de una forma más detallada.

2.1.2. Identificación de tráfico basada en flujos

Dentro de este ámbito se encuentran dos de las tres técnicas que se han presentado anteriormente.

- Identificación basada en los puertos de la capa de transporte. [9]

  Esta técnica consiste en identificar el tráfico dependiendo de los puertos de la capa de transporte, fue diseñada por IANA [14]. Esto se debe a que IANA es la organización que se encarga de asignar los protocolos de Internet, algunos nombres de dominios y coordinar la asignación de direcciones IP.

  Por lo tanto, IANA es quien asigna el número de puerto a los distintos protocolos, haciendo que se pueda identificar el tráfico por el número de puerto al que se envía. Este sistema tiene el inconveniente de que ahora mismo, se están desarrollando multitud de aplicaciones web. Estas aplicaciones se conectan mediante el puerto 80, es decir, mediante el protocolo HTTP. Pero esto es solo teoría, pues se puede hacer que cualquier aplicación envíe información por el puerto 80, sin que sea HTTP, lo cual daría como resultado una mala identificación y posterior clasificación.

- Identificación basada en la aplicación de técnicas de aprendizaje automático sobre estadísticas de tráfico [11].

  Esta técnica de identificación es muy sencilla. Consta de algoritmos de aprendizaje automático, los cuales se dedican a realizar análisis estadísticos. De esta forma los algoritmos aprenden que características tienen los flujos y los identifican en función de estas. Por lo tanto cuanto más tiempo pase, los patrones serán mejores y se tendrán más aciertos a la hora de la identificación.

2.1.3. DPI

- Identificación basada en el contenido del paquete o DPI. [10]

  Esta técnica de identificación consiste en analizar los paquetes, más allá de su cabecera. La Inspección Profunda de Paquetes, DPI por sus siglas en inglés, *Deep Packet Inspection*, esta técnica lo que hace es analizar todos los paquetes en busca de cadenas que se repitan, además de las IP's y los puertos. Una vez que tiene una cadena, la usa para buscar paquetes que tienen ese patrón, de esta forma establece un patrón de identificación.

  Esta técnica es útil para buscar virus, intrusiones y demás. Suele ser usada por los proveedores de servicios de Internet, ISP, *Internet Service Provider* y grandes empresas. Se puede decir que es una técnica que no respeta la privacidad, pues entra dentro de la propia información del paquete. Aunque en el caso de una gran empresa si puede hacerse, pues si se está en la red propia no es delito mirar la información que contienen los paquetes.

  En algunos países, como Alemania, donde es ilegal el uso de *torrents*, las autoridades pueden pedir información al proveedor de servicios sobre uno de sus clientes, si cree que este está infringiendo la red, por lo que esta técnica es ideal para este tipo de cometidos.

  Además de para lo descrito, esta técnica también es ideal para saber si se está siendo atacado y por quien. [15]

2.2. BRO

Bro [6], es un analizador de tráfico de red de código abierto. Al ser de código abierto permite ser usado por quien lo desee sin necesidad de pagar licencias. Funciona sobre sistemas basados en Linux y Mac OS X. Una de sus principales características es la gran cantidad de información que devuelve con un solo escaneo. Otros monitores de red devuelven menos información, teniendo que ser el propio administrador el que analice después la información obtenida por estos. Por lo tanto tardará más en resolver los posibles problemas que encuentre en la red.

Figura 2.1: Logo de Bro.

No tiene interfaz gráfica, por lo que su gestión se realiza desde la linea de comandos. Cuando se analiza tráfico, Bro generará unos registros, los cuales están divididos en función de parámetros definidos por parte del equipo que lo ha desarrollado.

Bro incorpora la posibilidad de introducir funcionalidades nuevas, mediante la programación en su propio lenguaje de *scripting*, del mismo nombre. Esto se desarrollará más adelante y en la parte de implementación se podrá ver cómo se desarrolla y algunos ejemplos de código escrito en Bro.

El lenguaje de *scripting* está orientado a trabajar con eventos. Por lo tanto a la hora de añadir una nueva funcionalidad habrá que crearla a partir del uso de los eventos que puede gestionar el programa.

Bro está estructurado de forma que todos los flujos de paquetes que analiza son procesados por el motor de eventos. Este motor convierte los flujos de paquetes en eventos, de forma que es más sencillo trabajar con ellos. Una vez que los eventos son procesados se generan los registros correspondientes, los cuales podrán ser analizados posteriormente. [16]

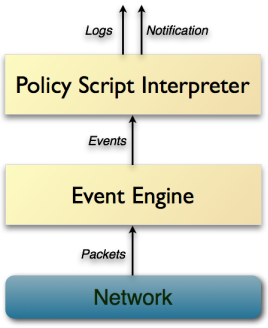

Figura 2.2: Arquitectura de Bro.

2.2.1. Funcionalidades básicas de BRO

La funcionalidad básica de Bro es la monitorización de la red en la que se ejecuta. Mientras que se encuentra en ejecución, genera registros o *logs* en texto plano que se podrán leer usando un editor de texto. Si se analiza un archivo *pcap* los *logs* no cambiarán. Sin embargo si se analiza tráfico a tiempo real, los registros se irán actualizando a medida que pase el tiempo. Algunos de los *logs* que se generarán son los siguientes.

- *dpd.log.* Consiste en un resumen de los protocolos encontrados en puertos que no son estándar.
- *dns.log.* Contendrá toda la actividad correspondiente al *DNS*
- *ftp.log.* Un registro de la actividad a nivel de sesión de *FTP.*
- *files.log.* Un resumen con los archivos transferidos a través de una red. Incluye protocolos *HTTP, FTP y SMTP.*
- *http.log.* Registro de toda la actividad *HTTP* con sus réplicas.
- *ssl.log.* Un registro de las sesiones *SSL*, incluidos los certificados que se utilizan.
- *weird.log.* En este *log* se guarda la información correspondiente a actividad inesperada o rara a nivel de protocolo. Al analizar gran cantidad de tráfico no es muy útil, pues ocurren muchas cosas raras, pero a pequeña escala es bastante interesante para detectar intrusiones y demás.
- *conn.log.* Aquí se puede ver la información correspondiente a conexiones *TCP, UDP e ICMP.*

Para más información sobre *logs* generados por una monitorización de Bro lea [17].

Bro dispone además de varios *frameworks* que extienden su funcionalidad. Con ellos se podrá crear *scripts* muy potentes. Algunas de las utilidades de estos *frameworks* son las siguientes.

- *Geolocalización.* Se podrá encontrar la localización geográfica de una IP.
- *Análisis de ficheros.* El monitor de red tiene la capacidad de trabajar con ficheros.
- *Framework de loggins.* Con este *framework* se podrá extender los archivos de registro que se generan.

- *NetControl*. Este *framework* permitirá tener un control muy amplio y diverso del tráfico.

Para mayor conocimiento de estos *frameworks* y otros lea [18].

Ahora con toda esta información de los registros, el administrador del sistema podrá determinar si existen amenazas en la red o si hay algún componente defectuoso. Para ello deberá de hacer uso de los eventos con los que Bro trabaja, para sacar mayor partido a todo el trabajo que se realice sobre la red.

2.2.2. Eventos y trazas

La programación en Bro no es secuencial, no se escribe un *script* y se espera que se ejecute en el orden en el que está escrito. La programación aquí esta orientada a eventos. Por lo tanto el orden depende de cuando se ejecute un determinado evento en el análisis.

Un evento se da cuando Bro detecta un "comportamiento", por ejemplo cuando detecta un paquete de respuesta *UDP*. Si se programa para que ese evento muestre un mensaje, este se mostrará cada vez que se detecte un paquete de respuesta *UDP*. Por lo tanto la programación deberá de estar pensada en función de los eventos disponibles.

Dentro de cada evento se podrá hacer lo que se desee, con la información que capture ese evento. Si el evento captura información de una conexión *TCP*, se tendrá que trabajar con esa información y las distintas variables globales que se hayan definido previamente.

Una traza es una captura del tráfico de una red. Con casi cualquier programa de diagnóstico se puede realizar capturas de red. En la propia web de Bro se encuentran algunos archivos de trazas de red, en formato *pcap*, de forma que se puedan realizar pruebas sobre ellos sin necesitar realizar una traza propia.

En el caso de Bro se podrá generar trazas de red mediante un *script* de *Python* [19], el cual se puede descargar desde el repositorio de *GitHub* de Bro. Con dicho script también se podrá trabajar sobre la traza, separando el tráfico entrante del saliente, desglosando en distintos registros el tipo de tráfico y demás.

2.2.3. Incorporación de funcionalidades

La incorporación de funcionalidades al monitoreo realizado por Bro es una característica muy llamativa. Gracias a esto se podrá realizar un análisis

muy personalizado usando un *script* creado por el administrador de redes. De esta forma podrá, por ejemplo, filtrar el tráfico de una determinada IP mientras se sigue analizando el tráfico de forma normal, con los registros que genera Bro de forma automática. Es una forma muy sencilla de comprobar si por ejemplo el servicio que administra está recibiendo demasiadas peticiones desde una misma IP. Lo cual sería un indicio de ataque de denegación de servicio.

A la hora de incorporar funcionalidades a Bro se puede hacer todo lo que se desee. Una búsqueda rápida por *GitHub* arrojará una gran cantidad de personas que contribuyen con una gran cantidad de nuevas funcionalidades [20]. Ahora lo ideal sería incorporar un módulo, de forma que si el resto de la comunidad lo desea pueda hacer uso de él de una forma sencilla.

2.3. Emparejamiento de flujos

La técnica de emparejamiento de flujos, fue planteada por el departamento de Teoría de la Señal, Telemática y Comunicaciones, *TSTC*, de la Universidad de Granada, en el año 2011 [21] [8].

En el emparejamiento de flujos, la idea de la que se parte es muy sencilla. Si dos paquetes comparten IP's de origen y destino, y puertos de origen y destino, se podrá decir que esos dos paquetes pertenecen a la misma clase. Una vez que están identificados como pertenecientes a la misma clase, lo que se debe de hacer es comparar los tiempos de esos paquetes, de forma que sean coherentes en el flujo. Si son muy lejanos en el tiempo, se podrá descartar que pertenezcan al mismo flujo. Esta idea se puede ver en [8] de una forma mucho más técnica.

En el mismo artículo [8] se encuentra la fórmula que se sigue para el emparejamiento.

$$F(x,y) = \begin{cases} G(x,y), & NIP(x,y) \geq 1 \\ -\infty, & \text{en otro caso} \end{cases}$$

Donde tenemos que G(x,y) se corresponde con la siguiente función:

$$G(x,y) = |NIP(x,y)–1|+1/(dp1(x,y)+k1)+1/(dp2(x,y)+k1)+1/(dt(x,y)+k2)$$

Las variables de la función son las siguientes:

- x: Primer paquete a comparar.

- *y*: Segundo paquete a comparar.
- *NIP(x,y)*: Es el número de paquetes analizados con las mismas características.
- *dp1(x,y)*: Se corresponde con los puertos de origen de los dos paquetes.
- *dp2(x,y)*: Serán los puertos de destino de los dos paquetes.
- *k1*: Es una variable definida previamente.
- *k2*: Es otra variable definida antes del comienzo del análisis. En trabajos anteriores esta variable y la anterior suelen estar definidas entre 1 y 10000.
- *dt*: Es la diferencia de tiempo existente entre los tiempos de inicio de los paquetes o *timestamps*.

Con esta fórmula se obtendrá como resultado un número, el cual será comparado con un umbral que se define previamente. Si el umbral definido es pequeño tendremos más flujos emparejados. Si el umbral es mayor serán menos los flujos que serán emparejados al no cumplir los requisitos.

El emparejamiento de flujos, por si mismo, sólo identifica el tráfico. Por lo tanto se necesitará otra técnica para clasificar el tráfico.

Capítulo 3

Diseño y arquitectura del sistema

En este capítulo se contará cómo se va a estructurar el módulo. Se explicará que funcionalidades debe de tener para que sea funcional. También se verá cómo gestiona Bro los flujos.

Todo esto será contado desde la vista del diseño, por lo tanto no se verá código, aunque se verán un par de tipos de datos interesantes. También se verá la arquitectura que seguirá el sistema.

3.1. Arquitectura del sistema

Para describir la arquitectura del sistema hay que tener en cuenta la arquitectura propia de Bro. Este monitor de red es un software modular. Esto quiere decir que esta compuesto de diferentes módulos que al ser ejecutados funcionan como un único sistema.

Por lo tanto la arquitectura de este problema se debe de acoplar a la arquitectura propia de Bro. Esto quiere decir, que se construirá un módulo. Entre los requisitos del módulo a desarrollar se encuentra que sea ligero y eficiente. Por lo tanto se tendrá que usar los distintos eventos que proporciona Bro para crearlo de la forma deseada.

Se prescindirá del uso de los *frameworks* descritos en el capítulo anterior, pues su uso no aporta nada que no se pueda realizar exclusivamente con los eventos destinados a gestionar el tráfico de la capa de transporte. Con el uso de estos eventos y alguna función auxiliar para el cálculo del umbral de emparejamiento de forma que se realicen menos cálculos dentro de los propios eventos. Con este esquema será suficiente para alcanzar los objetivos propuestos.

Entonces, a modo de resumen, la arquitectura final del sistema que se plantea será un módulo. Este estará compuesto por los distintos eventos de gestión del tráfico de la capa de transporte y una función auxiliar para el cálculo del umbral de emparejamiento.

3.2. Módulo y funciones

Dado que una mala programación del módulo puede dejar al monitor de red colgado con una traza simple. Se tratará de que el módulo contenga solamente lo justo para llevar a cabo la identificación del tráfico. De esta forma se obtendrá un módulo cuya ejecución será óptima.

Las funcionalidades que se espera que tenga este módulo en esencia son dos, la detección y almacenado del tráfico y la aplicación de la fórmula, vista en el anterior capítulo, a los distintos flujos que se han detectado. De una forma más amplia las funciones del módulo serán las siguientes.

- *Función que aplique la fórmula de emparejamiento.*

  A esta función se le pasará dos flujos, de forma que se aplique la fórmula y se discierna si pueden ser emparejados o no.

- *Funciones que detecten el tráfico.*

  Esto se hará con los eventos de Bro. Los eventos detectarán el tipo de tráfico que se está analizando y aplicarán la función anterior.

  Tras el uso de esta fórmula se almacenará o no el flujo que está siendo analizado. Por lo tanto será necesario el uso de algún tipo de contenedor para este cometido.

Lo que se espera es capturar el tráfico de la capa de transporte. Dicho tráfico se corresponde a los protocolos *TCP y UDP*. Por lo tanto será necesario ver que tipo de eventos son los que controlan el tráfico de estos dos protocolos. Esto se verá de forma más amplia en la siguiente sección.

Se debe de tener en cuenta que siempre se podrá extender la funcionalidad del módulo. Pero de momento no resulta interesante. La posible extensión corresponde a posibles trabajos futuros.

Para realizar este módulo es necesario conocer como gestiona los flujos Bro y de que forma se mantendrán los que son emparejados y los que están activos.

3.3. Gestión de flujos

La gestión de flujos en Bro pasa completamente por eventos. Por lo tanto se tendrá que crear variables globales para el almacenamiento de los flujos que sean emparejables y los que estén activos.

El *nacimiento de un flujo* es controlado por un evento. Por lo tanto cuando se detecta un nuevo flujo se lanza un evento. Este evento se tendrá que controlar de forma que si se tiene ya un flujo activo con las mismas características, se compare y se almacene. Si por el contrario no se tiene ningún flujo con esas características se tendrá que almacenar directamente en el contenedor de flujos activos.

La *muerte de un flujo* también es controlada por eventos. Ahora lo importante es si es interesante a nivel del análisis seguir almacenando los flujos aunque estos hayan muerto. Si no se quiere tener almacenados flujos muertos se tendrá que eliminar de la estructura en la que está almacenado. Si se quiere seguir trabajando con ellos habrá que mantenerlos guardados en la estructura. Obviamente si el flujo que va a morir está emparejado con otro se borrará solo del contenedor de flujos activos. Manteniéndose en el contenedor de flujos emparejados. De lo contrario se perderá información. Todo esto habrá que decidirlo en el evento que gestiona la muerte.

Estos dos comportamientos de los flujos están controlados por eventos genéricos. Con un único evento se detecta que el flujo ha nacido o ha muerto, independientemente del tipo de protocolo al que pertenezca. No pasará esto con los distintos estados de los flujos que serán detectados. Pues no es lo mismo detectar un ACK de un flujo TCP que una respuesta de un flujo UDP. Serán tratados en eventos distintos y tendrán que ser gestionados con eventos distintos, cada uno destinado a un protocolo distinto.

A continuación se verán las estructuras de datos necesarias para llevar a cabo el desarrollo del módulo.

3.4. Estructuras de datos

Las principales estructuras de datos que se necesitan para el desarrollo de este trabajo serán dos vectores, o semejantes para el almacenamiento de los flujos activos y los emparejados. Aunque esto se podrá ver mejor en la implementación.

Dichas estructuras de almacenamiento, deberán de ser capaces de estar ordenados por las IP's y los puertos. De esta forma se consigue que el acceso a los datos sea mucho más rápido. Incluso se podría prescindir de bucles,

los cuales pueden acabar siendo un problema en cuanto a rendimiento si se llega a almacenar muchos flujos.

Bro proporciona cierto tipos de datos muy interesantes y los cuales, además, incluyen mucha más información. Algunos de estos tipos de datos son.

- *connection.*

  Este tipo de dato es el flujo en si. Por lo tanto será de vital comprender-

  lo para poder trabajar como se desea. Para obtener más información sobre este tipo de dato lea [25].

- *time.*

  Este tipo de dato también es interesante. Aunque en otros lengua-

  jes se puede obtener, en Bro es un tipo de dato por si mismo. Por lo tanto se podrá operar sobre él desde el principio, siendo una gran ventaja a la hora de calcular el tiempo de inicio de los flujos. Para leer más [24].

Estos dos tipos serán los más utilizados e interesantes, junto con los contenedores de los flujos. Hay más tipos de datos, incluso algunos extienden la información disponible sobre los flujos [26].

Capítulo 4

Implementación

Ahora vamos a proceder a explicar cómo se ha solucionado los problemas propuestos, siendo estos explicados a través del programa que se ha realizado, quedando así más clara la explicación.

Lo primero será mostrar que estructura se ha utilizado para almacenar los flujos.

```
1
2 ## Tablas para guardar los flujos que son emparejados
3 global collection: table[addr, addr, port, port] of vector of
      connection &synchronized;
4 global collection_added: table[addr, addr, port, port] of vector of
      connection;
5
6 ## El umbral: "Comparar la constante 'k', que es el umbral que fijare
      con el resultado que devuelve la funcion,
7 ## si es mas grande el resultado que 'k' se puede decir que los dos
      flujos son iguales, si es mas pequenio podemos decir que los dos
      flujos no son iguales"
8 ## resultado del umbral que calculamos
9 global umbral: double;
10
11 ## Definimos el umbral, de manera global para hacer las comparaciones
12 global k=10;
```

En este primer código tenemos que:

- Vamos a guardar los flujos en una tabla *collection* que tiene como índices dos direcciones IP y dos puertos, y a partir de estos indices tendremos un *vector de connection*.
- La tabla *collection_added* está destinada para guardar los flujos que son emparejados.

- El *umbral*, que será una variable global, y será donde almacenemos el resultado de la función de comparación.
- k es el umbral que definimos nosotros al principio, que será el que usemos como referencia para saber si dos flujos son emparejables o no.

```
## funcion para la comparacion de los flujos, c1 el flujo que esta el primero en el vector de la tabla y c2 para el flujo que es candidato a ser emparejado

function emparejamiento(c1: connection, c2: connection ):double {

  ## Aniadimos variables para comprobar en la tabla, sin hacer bucle
  local orig = c1$id$orig_h;
  local dest = c1$id$resp_h;
  local po = c1$id$orig_p;
  local pd = c1$id$resp_p;

  local Nip = |collection[orig,dest,po,pd]|; ## Variable para saber cuantas conexiones tenemos
  local Po1: count; ## Puerto origen del primer flujo
  local Po2: count; ## Puerto origen del segundo flujo
  local Pd1: count; ## Puerto destino del primer flujo
  local Pd2: count; ## Puerto destino del segundo flujo
  local k1 = 1;  ## Variable fija
  local k2 = 10; ## Variable fija
  local dt: double; ## Variable para la diferencia de los tiempos
  local resultado = 0.0; ## Lo iniciamos a 0

  print c1$uid;
  print c2$uid;

  print fmt("Numero de Nip en table: %d", Nip);
  informacion_coincidencia(c1,c2);
  print fmt("Tiempo de inicio del flujo: %s", |c1$start_time|);
  print fmt("Tiempo de inicio del flujo: %s", |c2$start_time|);
  ## Para dp1 y dp2 que son 1-norm usamos la "Manhattan norm" que dice lo siguiente: SAD(x1,x2) = sumatoria(x1i - x2i)
  ## k1 y k2 son dos variables que nosotros le ponemos de forma manual, en este caso las pondremos como locales con 1 y 10 respectivamente
  ## dt es la diferencia de tiempo entre los time stamp de los primeros flujos de los flujos
  ## el tipo time se supone que es como un double, por lo tanto podremos restarlos sin problemas
  ## para la comparacion de puertos primero tendremos que hacer uso de la funcion  port_to_count [https://www.bro.org/sphinx/scripts/base/bif/bro.bif.bro.html#id-port_to_count]
  ## la cual nos pasa el puerto, que recordamos que va tambien con un string en el cual se nos dice que tipo es, a un
  ## valor numerico que si podremos restar sin problemas
  ## La funcion quedaria asi: (Nip-1)+(1/(dp1+k1))+(1/(dp2+k1))+(1/(dt+k2))
  Po1=port_to_count(c1$id$orig_p);
  Pd1=port_to_count(c1$id$resp_p);
  Po2=port_to_count(c2$id$orig_p);
  Pd2=port_to_count(c2$id$resp_p);
  ## local t1: double;
  ## local t2: double;
  ## t1 = time_to_double(c1$start_time);
```

```
43   ## t2 = time_to_double(c2$start_time);
44
45   dt=(|c1$start_time| - |c2$start_time|);
46
47   ## print fmt("Tiempo paquete 1: %s", t1);
48   ## print fmt("Tiempo paquete 2: %s", t2);
49   print fmt("Diferencia de tiempo: %s", dt);
50
51   resultado=(Nip-1)+(1/((Po1-Po2)+k1))+(1/((Pd1-Pd2)+k1))+(1/(dt+k2));
52
53  return resultado;
54
55 }
```

En este trozo de código se muestra la función que se usa para devolver el resultado de la función de comparación. El resultado siempre es comparado con el umbral que hemos definido antes de lanzar el programa, y una vez que se usa esta función quiere decir que son *comparables*, pero no sabemos todavía si son *emparejables*. Si son emparejables lo sabremos solamente después de que hayamos ejecutado la función y se realice la comparación con el umbral. Para que quede más claro aquí esta el código de la comparación con el umbral.

```
1  . . .
2
3          umbral=emparejamiento(cl,c);
4      print fmt("connection_finished");
5      if(umbral>k){
6        ## Si el umbral calculado es mayor que el umbral de comparacion
             lo aniadimos
7        print fmt("Si son emparejables TCP"); ## Mostramos TCP para
             saber en que evento se han calculado
8        collection[orig,dest,po,pd][|collection[orig,dest,po,pd]|] = c;
9        informacion_coincidencia(cl, c);
10       if( [orig,dest,po,pd] !in collection_added ){
11
12         collection_added[orig,dest,po,pd]=vector(c);
13         print fmt("Aniadimos una nueva conexion al vector de
               coincidencias");
14       } else {
15
16         ## Si ya esta, lo aniadimos
17         collection_added[orig,dest,po,pd][|collection_added[orig,dest,
               po,pd]|] = c;
18         print fmt("Ya esta en vector de coincidencias y lo aniadimos")
               ;
19       }
20
21     } else{
22       ## Si el umbral calculado es menor que el umbral de comparacion
             no lo aniadimos
23       print fmt("No son emparejables TCP");
24
25     }
26
27 . . .
```

En este ejemplo se calcularía para las conexiones de tipo *TCP*.

La comparación se dará cuando Bro reconozca un flujo de un determinado tipo de protocolo, en nuestro caso será mediante los eventos destinados a reconocer los tipos de protocolos de la *capa de transporte*. Dichos eventos son:

- connection_established, el cual se lanza cuando Bro localiza un SYN-ACK que corresponda a un handshake de TCP.
- connection_finished, el cual lanza Bro cuando una conexión TCP finaliza de forma normal.
- udp_request, el cual se lanza cuando Bro localiza un flujo que corresponde con UDP lanzado desde el origen.
- udp_reply, que es lanzado por Bro cuando localiza una respuesta a un flujo del anterior evento.
- icmp_echo_request, lanzado por Bro cuando localiza un flujo de tipo ICMP de echo.
- icmp_echo_reply, el cual es lanzado por Bro cuando localiza una respuesta a los flujos del evento anterior.

Dentro de estos eventos descritos se realiza la comprobación de que dos flujos se pueden comparar, ya que tienen los mismos IP's y puertos de origen y destino. Una vez en la función de comparación se almacenarán dichos puertos, pasándolos al tipo count para su uso en la función, se establecerán dos constantes, se calculará la diferencia de tiempo y en una variable se almacenará el número de flujos que hay con esos datos. Cuando realicemos esto, aplicaremos la función de cálculo y podremos decir si los flujos son emparejables.

Para mostrar la información de los flujos tenemos dos opciones, de las cuales mostramos el código a continuación:

```
1
2 ## Creo funcion auxiliar para ver la informacion del flujos que son
      coincidentes
3
4 function informacion_coincidencia(c: connection, p: connection){
5     print fmt("Informacion del primer flujo  IPo: %s , Po: %s , IPd: %
          s , Pd: %s ", c$id$orig_h, c$id$orig_p, c$id$resp_h,
          c$id$resp_p);
```

```
6       print fmt("Informacion del flujo coincidente  IPo: %s , Po: %s ,
            IPd: %s , Pd: %s ", p$id$orig_h, p$id$orig_p, p$id$resp_h,
            p$id$resp_p);
7   }
8
9   ## Funcion auxiliar para mostrar la informacion de un solo flujo
10
11  function informacion_flujo(c: connection){
12      print fmt("Informacion del flujo aniadido IPo: %s , Po: %s , IPd:
            %s , Pd: %s, uid: %s ", c$id$orig_h, c$id$orig_p, c$id$resp_h,
             c$id$resp_p, c$uid);
13  }
```

La función informacion_flujo sirve para mostrar la información del flujo relativa a las IP's y a los puertos que contiene, así como el *UID* del flujo, que es el identificador único que se le asigna automáticamente a cada flujo.

Por su parte la función informacion_coincidencia sirve para mostrar la infor-

mación de dos flujos a la vez, mostrando sus IP's y sus puertos. Esta función será usada para mostrar la información de los dos flujos que son emparejables, siendo el primero de ellos el que se usa para comparar siempre y el segundo es el nuevo flujo que se ha detectado. De esta forma tan sencilla se puede mostrar toda la información interesante sin tener que recorrer al final de la ejecución del programa todo la *tabla de connection*.

Para eliminar los flujos que, por su *timestamp* estén a punto de morir, Bro

nos da la opción mediante un evento de gestionar que hacer antes de que sean eliminados de la memoria, dicho evento es:

- connection_state_removed, evento que se lanza cuando un flujo activo va a ser eliminado de la memoria.

Cabe destacar que el borrado no es todavía operativo, por lo que en el script puede que este evento esté comentado.

```
1
2   ## Generated when a connections internal state is about to be
        removed from memory. Bro generates this event reliably
3   ## once for every connection when it is about to delete the internal
        state. As such, the event is well-suited for
4   ## script-level cleanup that needs to be performed for every
        connection.
5   ## This event is generated not only for TCP sessions but also for UDP
        and ICMP flows.
6
7   event connection_state_remove(c: connection){
```

```
8
9     local orig = c$id$orig_h;
10    local dest = c$id$resp_h;
11    local po = c$id$orig_p;
12    local pd = c$id$resp_p;
13    local coleccion = collection[orig,dest,po,pd];
14    local tama=|coleccion|;
15
16
17    for(j in coleccion){
18      if(j+1 >= tama){
19        if(tama==1){
20          collection[orig,dest,po,pd]=vector();
21        }
22        if(j+1==tama){
23          delete collection[orig,dest,po,pd][tama]; ## Aqui esta el
                error
24        }
25        break;
26      } else {
27        collection[orig,dest,po,pd][j]=coleccion[j+1];
28      }
29    }
30
31    print fmt("Terminamos copia y borrado...");
32
33  }
```

En este evento lo que tratamos de hacer es, antes de eliminarlo de la memoria, ver si existe algún flujo que pueda ocupar su lugar siendo el flujo referencia a la hora de comparar flujos. Básicamente lo que hace es comprobar si el tamaño es mayor que 1. Si es 1 solamente tendremos que borrar el vector contenido en la posición de la tabla, y si tenemos que es mayor, tendremos que borrar el primero y mover todos una posición hacia atrás. Nos interesa que este evento funcione, pues si los *timestamps* de dos flujos tienen mucha diferencia los daremos como que no son comparables, cuando si se van actualizando, tendremos más posibilidades de que la clasificación sea correcta.

Para añadir flujos a la tabla de conexiones, tenemos un evento dedicado

a ello:

- new_connection, Bro lanza este evento cada vez que detecta un flujo que era desconocido.

```
1  ## Cada vez que entra un nuevo flujo compruebo que si esta en la tabla
2  ## Este evento se lanza con cada nueva conexion de un flujo que no sea
       conocido, pero al borrarlo de memoria ser  desconocido aunque ya
       lo tengamos en la tabla
```

```
3  ## Generated for every new connection. This event is raised with the
       first packet of a previously unknown connection. Bro uses a flow-
       based definition of connection here that includes not only
       TCP sessions but also UDP and ICMP flows.
4
5  event new_connection(c: connection){
6
7    local orig = c$id$orig_h;
8    local dest = c$id$resp_h;
9    local po = c$id$orig_p;
10   local pd = c$id$resp_p;
11   print fmt("new_connection");
12
13   if( [orig,dest,po,pd] !in collection ){
14
15     ## Si no estan los valores clave del flujo lo creamos
16     collection[orig,dest,po,pd]=vector(c);
17     informacion_flujo(c);
18     print fmt("A adimos una nueva conexion");
19
20   } else {
21
22     ## Si ya esta, lo a adimos
23     collection[orig,dest,po,pd][|collection[orig,dest,po,pd]|] = c;
24     informacion_flujo(c);
25     print fmt("Ya esta y la a adimos");
26
27   }
28
29 }
```

Lo que se realiza en este evento es ver si en la tabla global tenemos ya almacenado un flujo de esas características. Si no tenemos uno igual creamos en la tabla para los valores de las IP's origen, destino y puertos de origen y destino un nuevo vector que contenga solamente este nuevo flujo. En caso de que si tengamos los valores ya almacenados solamente tendremos que añadir al final del vector el nuevo flujo. En ambos casos mostraremos la información del flujo.

Por último usamos dos eventos en el programa para ver cuando se lanza

y cuando finaliza el programa:

- bro_init, evento que se lanza cuando iniciamos Bro.
- bro_done, evento que se lanza cuando Bro va a terminar.

```
1  ## Evento que se lanza cuando se inicia BRO
2
3  event bro_init(){
```

```
4
5     print fmt("Hora de inicio: %s", current_time());
6
7  }
8
9  ## Evento que se genera cuando BRO va a tenerminar
10
11 event bro_done(){
12
13
14    print fmt("Hora de finalizacion: %s", current_time());
15
16 }
```

En el primer evento, que es el que se lanza cuando Bro se ejecuta, mostramos la hora de inicio. En el segundo evento, que solamente se ejecuta cuando Bro va a finalizar, mostramos la hora de finalización.

Capítulo 5

Evaluación y pruebas

Capítulo 6

Conclusiones y trabajo futuro

Bibliografía

[1] James F. Kurose and Keith W. Rose. *Redes de computadores un enfoque descendente*. Pearson, 2010.

[2] James F. Kurose and Keith W. Rose. *Redes de computadores un enfoque descendente*, chapter 2. Pearson, 2010.

[3] James F. Kurose and Keith W. Rose. *Redes de computadores un enfoque descendente*, chapter 3. Pearson, 2010.

[4] James F. Kurose and Keith W. Rose. *Redes de computadores un enfoque descendente*, chapter 4. Pearson, 2010.

[5] James F. Kurose and Keith W. Rose. *Redes de computadores un enfoque descendente*, chapter 1. Pearson, 2010.

[6] Bro Team. Bro indice, . URL `https://www.bro.org`.

[7] James F. Kurose and Keith W. Rose. *Redes de computadores un enfoque descendente*, page 55. Pearson, 2010.

[8] José Camacho, Pablo Padilla, Pedro García-Teodoro, and Jesús Díaz-Verdejo. A generalizable dynamic flow pairing method for traffic classification. 2013.

[9] Internet Assigned Numbers Authority (IANA). Service name and transport protocol port number registry. URL `https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml`.

[10] Christian Fuchs. Implications of deep packet inspection (dpi) internet surveillance for society. URL `http://fuchs.uti.at/wp-content/uploads/DPI.pdf`.

[11] Thuy T.T. Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. URL `http://ieeexplore.ieee.org/document/4738466/?reload=true`.

[12] Álvaro Maximino Linares Herrera. Repositorio del trabajo con bro. URL https://github.com/Lynares/bro-flows.

[13] Bro Team. Descarga de bro, . URL https://www.bro.org/download/index.html.

[14] Kim Davies. Una introducción a iana. URL https://www.iana.org/about/presentations/davies-atlarge-iana101-paper-080929-es.pdf.

[15] Dr. Thomas Porter. The perils of deep packet inspection. URL https://www.symantec.com/connect/articles/perils-deep-packet-inspection.

[16] Bro Team. Arquitectura de bro, . URL https://www.bro.org/sphinx/intro/index.html#architecture.

[17] Bro Team. Logs de bro, . URL https://www.bro.org/sphinx/script-reference/log-files.html.

[18] Bro Team. Frameworks de bro, . URL https://www.bro.org/sphinx/frameworks/index.html.

[19] Bro Team. Trazas en bro, . URL https://www.bro.org/sphinx/components/trace-summary/README.html.

[20] securitykitten. Finding beacons with bro. URL https://gist.github.com/securitykitten/a7edcee0932c556d5e26.

[21] José Camacho, Pablo Padilla, F. Javier Salcedo-Campos, Pedro Garcıa-Teodoro, and Jesús Dıaz-Verdejo. Pair-wise similarity criteria for flows identification in p2p/non-p2p traffic classification. 2011.

[22] Bro Team. Web de bro, . URL https://www.bro.org/sphinx/intro/index.html.

[23] Bro Team. Instalación de bro, . URL https://www.bro.org/sphinx/install/index.html.

[24] Bro Team. Tipo time, . URL https://www.bro.org/sphinx/script-reference/types.html?highlight=time#type-time.

[25] Bro Team. Tipo cennection, . URL https://www.bro.org/sphinx/scripts/base/init-bare.bro.html#type-connection.

[26] Bro Team. Tipo conn, . URL https://www.bro.org/sphinx/scripts/base/protocols/conn/main.bro.html#type-Conn::Info.

[27] Bro Team. Función get_port_transport_proto, . URL https://www.bro.org/sphinx/scripts/base/bif/bro.bif.bro.html#id-get_port_transport_proto.

[28] Bro Team. Analizadores de protocolos, . URL https://www.bro.org/sphinx/script-reference/proto-analyzers.html.